Denotational Semantics of Mobility in Unifying Theories of Programming (UTP)

Gerard Ekembe Ngondi

University of York

Computer Science

November 2016

Abstract

UTP promotes the unification of programming theories and has been used successfully for giving denotational semantics to Imperative Programming, CSP process algebra, and the Circus family of programming languages, amongst others.

In this thesis, we present an extension of UTP-CSP (the UTP semantics for CSP) with the concept of mobility. Mobility is concerned with the movement of an entity from one location (the source) to another (the target). We deal with two forms of mobility:

- Channel mobility, concerned with the movement of links between processes, models networks with a dynamic topology; and
- Strong process mobility, which requires to suspend a running process first, and then move both its code and its state upon suspension, and finally resume the process on the target upon reception.

Concerning channel mobility:

- We model channels as concrete entities in CSP, and show that it does not affect the underlying CSP semantics.
- A requirement is that a process may not own a channel prior to receiving it. In CSP, the set of channels owned by a process (called its interface) is static by definition. We argue that making the interface variable introduces a paradox. We resolve this by introducing a new concept: the capability of a process, and show how it relates to the interface.

We then define channel mobility as the operation that changes the interface of a process, but not its capability. We also provide a functional link between static CSP and its mobile version.

Concerning strong mobility, we provide:

- The first extension of CSP with jump features, using the concept of continuations.
- A novel semantics for the generic interrupt (a parallel-based interrupt operator), using the concept of Bulk Synchronous Parallelism.

We then define strong mobility as a specific interrupt operator in which the interrupt routine migrates the suspended program.

Contents

List of Figures

This thesis is gratefully dedicated to my father

EKEMBE SAMUEL

and to my mother

NGONDI YAKA HERMINE JULIETTE

Acknowledgements

I wish to express my sincere thanks to Pr. Jim Woodcock for his helpful supervision and for the great interest he has shown for my studies. The completion of this work has been aided considerably by his many suggestions.

I would also like to thank my examiners Dr. Jeremy Jacob and Pr. Shengchao Qin for their professional, rigorous, and helpful feedback.

I wish to express my gratitude to my colleagues for the many useful discussions we have had. A special thanks goes to Victor for his patient listening.

Finally, I wish to thank my family whose continuous moral support and prayers have withheld me throughout these past years of research. My thanks also to friends of old for their encouragements, and new friends, for providing me with an environment suitable for my studies.

Author's Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Some parts of this thesis have been submitted in conferences proceedings; where items were published jointly with collaborators, the author of this thesis is responsible for the material presented here.

- G.N. Ekembe, *Unifying Theories of Mobile Channels*, Refine'15, pp. 24-39, 2016. doi:10.4204/EPTCS.209.3
- G.N. Ekembe, J. Woodcock, *UTP Semantics of Reactive Processes with Continuations*, UTP'16, (submitted).

Chapter 1

Introduction

This document contains results of a research started in October 2012 about the concept of *mobility* in Computer Science. Two questions must enter into consideration when talking of movement: what moves? and in which space? Computer Science is filled with mobile objects; e.g. data, moving from one program/computer to another in a computer system; given a state space, programs, moving from one state to another. Mobility captures the movement of a given entity, from one location to another.

The need for a theory of mobility has been greatly motivated by the growing importance of computer networks, notably the Internet. For industrials, it is expected that such a theory could help them to increase network performance. For the programmer, it is a paradigm that may help him to better understand and program aspects of a system, especially those that can be expressed using mobility.

Three forms of mobility may be considered:

- channel mobility: (references to) channels move in a virtual space of linked processes.
- weak (process) mobility: only the 'code' of a process moves, perhaps together with some initialisation data.
- strong (process) mobility: here a process's execution is suspended. The process's code is then migrated or moved together with its execution state to another (computational) location where its execution is resumed from the place it had been suspended.

Tang & Woodcock [126] have defined denotational semantics for weak mobility based on UTP [66]. We plan to extend their work with the other two forms of mobility cited above.

Objective. Define the formal semantics of both channel mobility and strong (process) mobility, on the basis of UTP-CSP [66, Chap. 8], [35]. This objective may be broken down into two objectives, one for channel mobility, and the other for strong mobility.

State of the art: Channel mobility. There are not many works on denotational semantics for channel mobility in general and even less on the basis of CSP [65], [106], [115].

[48], [122], and [62] are based on category theory and are not suitable for a direct reasoning about channel mobility.

[28, 55, 56, 58, 123, 14] all contain work on extending with channel mobility FOCUS [27], a semantics framework for data flow networks.

Some works attempt to build a traces model for a language that may be viewed as a subset of CSP, although they do not firmly establish a relation with CSP: They are [96, 18], and [67].

Roscoe proposes a CSP operational semantics to the pi-calculus in [108]; and in [109], Roscoe proposes a closed-world semantics, an attempt at a denotational semantics of channel mobility in CSP.

In [116, 132], an extension of CSP||B with channel mobility is proposed, but the semantics are quite limited since the only channels that may be moved are those between a CSP controller and a B machine, and not for example, channels between any two CSP controllers.

Finally, [138] proposes a CSP semantics for the channel mobility mechanism of the programming language occam-pi; however, the result contains many implementation details, not suitable for abstract reasoning hence.

General remarks concerning the aforementioned works are these: they are limited to a traces model; the languages proposed may all be viewed as a subset of CSP.

We may classify them according to whether they use so called *dynamic alphabets*, or they use *name generation* (as in the pi-calculus). For the latter, the resulting semantics are quite complex and difficult to relate to their static counterpart in CSP. For the former, we may argue that no model is well-founded, depending on how they answer the following questions: what makes a channel *new*? And, what entitles a process of *using* a new channel?

In [56, 58], they consider that every process (or component) has *by default* the set of all possible channels as their interface; then each component restricts what channel may be used at what time. We may then ask the question: if the process already *has* the channel, is it not just a matter of ensuring that only a subset of it may be used, so that there is not, *stricto sensu*, channel mobility?

None of the CSP-based works cited above deals adequately with the requirement of making channels *concrete*. Such a *requirement* is imposed by the explicit manipulation of channels. However, whilst channels are defined to be concrete in both [18] and [67], neither discuss how concrete channels affects their proposed semantics, knowing that channels are *not* concrete entities in CSP.

An operator that makes the semantics of channel mobility more complex is hiding. None of the CSP-based models cited above actually has the hiding operator either. [58] defines a form of hiding in which only the channels that are in the initial interface (the interface provided at the time of the definition) may be hidden. This reduces the scale of the systems that such languages may model. Mobile Telecommunications Networks and the Internet are examples of systems in which the number of channels may increase silently with the size of the network. If one may object that such systems are too large, we may take the example of

a single router. Initially, its routing table is limited to a few nodes; but after some time, the size of the routing table may grow exponentially. If each line of the routing table is modelled as a channel linking the router to some network, we have another example of a mobile system for which the hiding of new channels may not be modelled. In fact, the hiding operator poses difficulty even in the context of operational semantics, as can be testified by the papers [20] and [111].

Finally, to the best of our knowledge, no work on channel mobility (in both operational and denotational semantics) actually defines functional links between theories of static and mobile processes.

State of the art: Process mobility. As already stated, Tang & Woodcock have provided a denotational semantics for weak mobility on the basis of UTP, in [125, 126]. That is the only work on denotational semantics for weak mobility found by the author. In the realm of operational semantics we may cite HO-pi [112], which extends the pi-calculus with weak mobility.

There is almost no work on semantics for strong mobility, except for [131], which proposes a denotational semantics in the context of Object orientation. The semantics are based on the programming language Join Voyager, and not on traditional mathematical domains as is usually the case. An advantage of the approach in [131] is the availability of already implemented functionalities, most especially *interrupt mechanisms*. An inconvenient is that the semantics are not abstract enough.

In UTP-CSP, the definition of strong process mobility requires building the features for continuations for UTP-CSP processes. It is also necessary to define adequate interrupt operators. This second requirement is partly met by existing semantics for the catastrophic interrupt operator $\triangle_i$ in [84]. However, without action prefix $(a \rightarrow P)$ —including communication—, $\triangle_i$ is obsolete since an interrupt may occur only when the running process is in a waiting state.

Another difficulty (with providing semantics for strong mobility in a CSP-based model) is that all parallel-based semantics of interrupt are based on timed semantics. This may be a problem since one may have to deal with three levels of abstraction simultaneously in the same UTP theory: continuations (implementation level), (untimed) UTP-CSP, and time (but only for the interrupt operator?). Since time is not necessary for the understanding of strong mobility, it seems cumbersome to introduce time even if only to abstract away from it.

Finally and not the least, it is difficult to save the state of the executing process upon interrupt. K.Wei [135] could not achieve this; and Huang et al. [69, 70] achieve this only at the cost of violating many healthiness conditions for CSP processes, notably by saving the state into the trace. Also, Huang et al. define interrupts for a sequential language only. We are not aware of any other significant work on interrupt in the context of UTP.

Motivations. We are interested in extending UTP as part of a general effort for the unification of theories of programming. In the context of this thesis, we are concerned with

mobility as the proposed extension. Since mobility is inherently distributed, the UTP theory of CSP processes (or UTP-CSP) may serve as our basis. Notwithstanding the difficulties, we are willing to use exclusively the formulation based on reactive processes (and not reactive designs), and also untimed UTP-CSP, as we believe that it should be possible to express a parallel version of the interrupt operator in an untimed model. Concerning channel mobility in particular, we are interested in refusals and divergences models in addition to traces.

Contributions: Channel mobility. In general, we propose a solution to all of the problems expressed above regarding channel mobility. In detail:

- The introduction and formal characterisation of a new concept called the *capability* of a process. Thus we:
  - formalise the requirement that channels need to be concrete entities;
  - establish the relation between concrete channels and the traditional notion of an *interface* in CSP;
  - argue that name generation models such [18] and [108], including even the pi-calculus [86] must rely on a static *capability* and a dynamic interface in order to be qualified of characterising channel mobility;
- A formal semantics of channel mobility in UTP-CSP applicable also to all existing CSP-based (and CSP-like) models;
- Functional links between static UTP-CSP and UTP-CSP extended with channel mobility (or simply mobile CSP);
  - we suggest that a similar link may be build between CCS and the pi-calculus and propose a way for doing so. Our hypothesis is that if we algebraically characterise both the notions of process capability and of process interface in the algebraic framework defined by (an algebraic framework similar to the one defined by) de Simone [120], then the only difference between CCS and the pi-calculus would be, for the second, an operation that changes the interface;
- A model of a buffer using mobile channels, and its transformation into a static version;
- A model for a mobile communications network, comparable to the one defined in [86];
- A new semantics for the hiding operator in which the set of silent channels may grow;
- A new semantics for the renaming operator in which names may be renamed as they are acquired;
- A new way of interpreting the hiding operator that dissolves the distinction between *internal* vs. *external* mobility through a subtle separation of concerns: hiding hides, and the process communicates.

- On the basis of all the previous results, we argue that name generation such as developed for the pi-calculus is unnecessary in the context of denotational semantics; we further argue that such may be the case in the definition of the pi-calculus itself.

Contributions: Process mobility. We have solved a number of problems related to the definition of strong mobility in (untimed) UTP-CSP. In paricular:

- Continuations semantics for UTP-CSP;
- A parallel-based semantics for the generic interrupt operator that may provide a basis for defining a canonical form for all interrupt operators;
  - untimed UTP-CSP semantics for the generic interrupt, without having to transform *first* the timed semantics such as provided by K. Wei [135]. In particular our definition allows saving the state of the executing process when the interrupt occurs. The semantics are based on the concept of Bulk Synchronous Parallelism (or parallel-by-multiple merge [66, Chap. 7]).
- Denotational semantics for strong mobility in UTP-CSP as a particular form of interrupt operator (or family thereof) in which the interrupt routine sends the code of the running process together with its interrupt state (including its next continuation) to a remote location.
  - Although not defined in this thesis, the semantics for strong mobility may serve as the basis for defining a local version of the operator in which the interrupted process is resumed locally instead.

This thesis is structured as follows.

Chapter 2 introduces UTP, notably UTP concepts whose knowledge is necessary for understanding most of the materials in this thesis.

Chapter 3 contains a review of the literature on denotational approaches to the semantics of channel mobility. It also contains such works that discuss the extension of CSP with channel mobility even when operational semantics are involved. In particular, we provide some additions to existing works in our discussions, especially:

- in Section 3.1.2, we add a paragraph on design decisions;
- in Section 3.2.1 we discuss a simpler formulation of the model which has notably highlighted some unresolved questions in the original semantics;
- in Section 3.2.2 we propose a way towards simplifying existing semantics (by means of abstraction);
- in Section 3.2.5 we propose a separation of the traces model that may make reasoning easier;

- in Section 3.2.6 we make a suggestion towards simplifying the original model. Every work presented is followed by a discussion of its channel mobility mechanism.

- Section 3.4 contains general remarks concerning the literature; of particular interest, we discuss the separation between hiding and communication, and argue that the distinction between internal and external mobility is not necessary; we argue that passing names is not enough for the characterisation of channel mobility, not even in the pi-calculus; we propose an axiomatisation of mobile calculi which answers the question of the *characterisation* of channels.

In Chapter 4 we discuss the UTP model for channel mobility. Therein, we formalise the concept of capability and show how it relates to the traditional concept of interface in CSP. We give the semantics of channel mobility and then define some operators, in particular dynamic hiding, and dynamic renaming. We then construct the link between static and mobile CSP. Amongst other examples, the chapter contains an illustration of both the semantics of channel mobility, and links to static CSP, of a buffer with mobile channels; and an application of channel mobility to model features of a mobile communications network.

Chapter 5 is about strong mobility. We first define continuations for UTP-CSP processes in Section 5.2. We then discuss the introduction of the state explicitly into the trace of CSP processes. In Section 5.4 we provide the semantics of the generic interrupt based on the concept of Bulk Synchronous Parallelism (or parallel-by-multiple merge as called in UTP). We then give the semantics of strong mobility, in Section 5.5.

Each chapter ends with a discussion of the results. Finally, in the Conclusion we summarise the thesis and the implications of our results, and also discuss future work.

Chapter 2

Unifying Theories of Programming (UTP)

2.1 Generalities

A program may be conceived of as a device that takes some data as input and returns some possibly different data as output. Output data results from the transformation of the input according to a set of rules or instructions that define the transformation. The data received as input is stored into a memory that may be accessed any time, and similarly for the output. A program variable is the representation of the memory cell that stores the data: at different times, it may hold different values. The value of a variable is also called its *state*, and the state of a program is the Cartesian product of the states of all of its variables. Hence, a program may be defined as a state transformer, that transforms any input state into an output state according to given transformation rules.

The effect of a program on its variables is also called the *behaviour* of the program, and may be described in a number of ways. For example, as a mathematical function, where the function parameters would stand for the input variables, the body of the function for the transformation rules, and the result returned by the function for the output variables. Another possibility is to consider an observer that observes the program's behaviour and writes statements (predicates) about his observations. Such a predicate would notably state the values of the program variables at a given observation time. In this case, input values correspond to values before the program starts, and output values to those at the end of the program. The program transformation itself may then be represented by the relation between those two (sets of) values. It is this latter approach that is adopted in UTP.

Let X denote a program, and x, y, z denote the (program) variables of X. Let $P(X)$ denote a predicate describing X. Then, to each program variable x of X may be associated a pair of logical variables $(\grave{x}, \grave{x}')$ such that $\grave{x}$ denotes input values and $\grave{x}'$ output values of X. If X is a deterministic program viz. such that every input always has the same output on every run of X, then X may be represented by some function, say f. That is, we may write $P(X) = \{(\grave{x}, \grave{x}') \mid \grave{x}' = f(\grave{x})\}$. If X is non-deterministic viz. the same input may

have different outputs on different runs of X, then X may be represented by a relation, say r. Then, $P(X) = \{(lx, lx') \mid (lx, lx') \in r\}$. Note that the difference between f and r is that f denotes a relation where every lx has a unique lx', whereas for r, every lx as at least one corresponding lx'.

In practice in UTP, the set notation is not much used, and the distinction between the notions of program, predicate and relation is left implicit. The distinction between program variable and predicate (or logical) variable is also obsolete, as a predicate is seen as a program itself.

Whilst a program defines the set of rules for computing variables, the elements of a program, if modified, may yield different programs. For example, by changing the order of application of the rules. That is, the same set of rules may help defining many distinct programs. A programming language defines the set of rules for constructing programs. Just like French and English languages may serve to denote the same objects, different programming languages may serve to write the same programs. Yet, different programming languages may also serve to define different classes of programs, or different programming models or theories. Below, we discuss how UTP does represent all those programming concepts.

UTP is a formal semantics framework for reasoning about programs, programming theories and the links 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. Relations are themselves represented as *alphabetised predicates*, i.e. predicates of the form $(\alpha P, P)$. αP is called the *alphabet* of the predicate P and determines what variables P may mention. αP may be partitioned into two subsets: $in\alpha P$, which represents the initial observations, and $out\alpha P$, which represents the final observations.

Programming languages and paradigms are formalised as UTP theories. A UTP *theory* is simply a collection of predicates, and consists of three elements: an *alphabet*, containing only those variables which 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 (i.e. healthy predicates are those that can be implemented).

In UTP, specifications, designs, programs, and implementations are all regarded as predicates (or equivalence classes thereof). They differ only in their level of abstraction, which may be captured formally thanks to a refinement relation.

The construction of a UTP theory is generally done as follows: starting from the more general theory of relations, characterise those relations that are of interest, and leave out the rest. Hence, UTP makes possible the study of new theories in isolation.

In section 2.2, we present the relational calculus, that defines all possible and imaginable relations. The corresponding language is too large as it may serve to define also relations that are not programs in the intended sense, though some programming constructs may already

be defined on its basis.

The theory of designs is then constructed by applying restrictions to the theory of relations by means of some healthiness conditions. Designs permit the definition of every programming theory of interest, and are the subject of section 2.3.

In general, the description of a programming concept (construct, language, paradigm) will be preceded by some informal explanation of its meaning. Because every program is a relation, UTP promotes reuse of existing theories, by embedding existing theories into new ones. Hence, most, if not all the constructs of sequential programming will be found across many other theories. We present, in section 2.4, how UTP theories can be linked together.

Thanks to UTP, many paradigms and language constructs have been defined. In section 2.5, we present the theory of Reactive Processes, for reasoning about reactive programs. There are many computation models describing reactive programs or simply a view of it, some formal, others informal. CSP [65] is a formal framework that permits reasoning about reactive programs. The UTP semantics of CSP [35], [66] are presented in section 2.5.1.

Finally, the semantics for continuations (for modelling control flow) are presented in section 2.6.

All the semantics presented below are denotational, which is the way that most UTP theories are presented. However, it is also possible to define both operational semantics and algebraic semantics, as illustrated in [66, Chaps. 5, 10].

2.2 Relational calculus

In this section we present the theory of alphabetised relations which serves as a basis for the definitions of all other UTP theories. First we define what a relation is.

Some definitions. A *relation* R between two sets A and B is a subset of their Cartesian product i.e. $R \subseteq A \times B$. A *binary relation* is a relation over $A \times A$. Let $\alpha P = in\alpha P \cup out\alpha P$ denote the alphabet of a predicate P, then P denotes an *alphabetised relation* over $in\alpha P \times out\alpha P$. A relation is said to be *homogeneous* if $in\alpha P = out\alpha P$, and *heterogeneous* otherwise.

All of the theories presented here use homogeneous relations only, unless otherwise stated. A presentation of heterogeneous relations may be found in [75].

Monotonicity, Idempotence. Let $\mathbf{A}$ and $\mathbf{B}$ denote sets of predicates. A *predicate transformer* from $\mathbf{A}$ to $\mathbf{B}$ is a function from $\mathbb{P}\,\mathbf{A}$ to $\mathbb{P}\,\mathbf{B}$. A predicate transformer f is *monotonic* if

$$P \Rightarrow Q \text{ then } f(P) \Rightarrow f(Q)$$

and it is *idempotent* if

$$f \circ f(P) = f(P)$$

In what follows, and in the rest of this thesis, we will use the terms relations, predicates, alphabetised relations and alphabetised predicates interchangeably. Some programming constructs are defined below.

$x :=_{\mathsf{A}} e$ denotes the assignment of an expression e to a variable x. The meaning of assignment is thus equality: that between x and e at the end of the assignment.

Definition 2.2.1 (Assignment [66, Def. 2.3.1])**.**

$$\begin{aligned} x :=_{\mathsf{A}} e &\mathrel{\widehat{=}} (x' = e \wedge y' = y \wedge .. \wedge z' = z) \\ \alpha(x := e) &\mathrel{\widehat{=}} \mathsf{A} \cup \mathsf{A}' \end{aligned}$$ □

II_A denotes the command that does nothing; it is equivalent to the assignment $x := x$.

Definition 2.2.2 (Skip [66, Def. 2.3.2])**.**

$$\begin{aligned} II_{\mathsf{A}} &\mathrel{\widehat{=}} (x' = x) \qquad \text{where } \mathsf{A} = \{x, x'\} \\ \alpha II_{\mathsf{A}} &\mathrel{\widehat{=}} \mathsf{A} \end{aligned}$$ □

$P \lhd b \rhd Q$ stands for 'if b then P else Q', where b is some testable condition. Formally, a *condition* is defined as a predicate not containing dashed variables.

Definition 2.2.3 (Conditional [66, Def. 2.1.1])**.**

$$\begin{aligned} P \lhd b \rhd Q &\mathrel{\widehat{=}} (b \wedge P) \vee (\neg\, b \wedge Q) \qquad \textit{if } \alpha b \subseteq \alpha P = \alpha Q \\ \alpha(P \lhd b \rhd Q) &\mathrel{\widehat{=}} \alpha P \end{aligned}$$ □

$P \fatsemi Q$ denotes a program that first executes P, and when P terminates, then it executes Q.

Definition 2.2.4 (Sequential composition [66, Def. 2.2.1])**.**

$$\begin{aligned} P(v') \fatsemi Q(v) &\mathrel{\widehat{=}} \exists\, v_0 \bullet P(v_0) \wedge Q(v_0) \qquad \textit{if } out\alpha P = in\alpha' Q = \{v'\} \\ in\alpha(P(v') \fatsemi Q(v)) &\mathrel{\widehat{=}} in\alpha P \\ out\alpha(P(v') \fatsemi Q(v)) &\mathrel{\widehat{=}} out\alpha Q \end{aligned}$$ □

var x denotes the declaration of a new program variable x, and **end** x its undeclaration.

Definition 2.2.5 (Variable declaration, undeclaration [66, Def. 2.9.1])**.** *Let* A *be an alphabet that includes* x *and* x'*. Then:*

$$\begin{aligned} \mathbf{var}\, x &\mathrel{\widehat{=}} \exists\, x \bullet II_{\mathsf{A}} & \alpha(\mathbf{var}\, x) &\mathrel{\widehat{=}} \mathsf{A} \setminus \{x\} \\ \mathbf{end}\, x &\mathrel{\widehat{=}} \exists\, x' \bullet II_{\mathsf{A}'} & \alpha(\mathbf{end}\, x) &\mathrel{\widehat{=}} \mathsf{A} \setminus \{x'\} \end{aligned}$$ □

The scope of a variable x lies between **var**x and **end**x; beyond, the variable is undefined and cannot be observed.

Definition 2.2.6 (Alphabet extension [66, Def. 2.9.5])**.** *Let $x, x' \notin \alpha P$, then:*

$$\begin{aligned} P_{+x} &\mathrel{\widehat{=}} P \wedge x' = x \\ \alpha P_{+x} &\mathrel{\widehat{=}} \alpha P \cup \{x, x'\} \end{aligned}$$

□

$P \sqcap Q$ denotes the program that executes either P or Q; the environment has no control over such a choice and no means of prediction either.

Definition 2.2.7 (Nondeterminism [66, Def. 2.4.1])**.**

$$\begin{aligned} P \sqcap Q &\mathrel{\widehat{=}} P \vee Q \qquad \textit{if } \alpha P = \alpha Q \\ \alpha(P \sqcap Q) &\mathrel{\widehat{=}} \alpha P \end{aligned}$$

□

The Complete lattice. The space of predicates used for describing programs is bounded. It has a bottom element which corresponds to the program whose behaviour is totally uncontrollable: it is the weakest predicate *true*, denoted by $\bot$, and called *abort*. The top element corresponds to the program whose behaviour is totally unobservable: it is the strongest predicate *false*, denoted by $\top$, and called *miracle*. They are characterised by the following laws:

L1	$[P \Rightarrow \bot]$	*for all P*
L2	$[\top \Rightarrow P]$	*for all P*

where $[Pred]$ denotes the universal closure of a given predicate $Pred$ i.e. for $\alpha Pred = \{x, y, x', y'\}$, $[Pred]$ denotes $\forall\, x, y, x', y' \bullet Pred$.

A mathematical space with an ordering that has the least upper bound (lub) and the greatest lower bound (glb) of all subsets of its element is known as a **complete lattice**. *The space of relations is a complete lattice.* This result is important for the definition of recursive programs.

Recursion. A recursive program is one that makes calls to itself. To model such a program it is necessary to define a predicate variable X. Defining X recursively is then similar to applying a function F to X such that X satisfies the following equation:

$$X = F(X)$$

Such an X is called a fixed point. The definition of recursion adopted in UTP is that of the weakest fixed point, μF, which represents the glb of all the fixed points of F. Formally,

$$\mu F \mathrel{\widehat{=}} \bigsqcap\{X \mid X = F(X)\}$$

However, this definition is unsatisfactory: using it, it is possible to model a terminating program that contains a non-terminating loop. For illustration, consider the predicate describing a while loop, denoted by $b * P$. A while loop defines the statement *while condition b is true, execute program P*. Formally,

$$b * P \mathrel{\widehat{=}} \mu X \bullet ((P \fatsemi X) \lhd b \rhd II)$$

A non-terminating loop may be obtained by replacing b and P by *true* and II respectively.

$$\begin{aligned} true * II &= \mu X \bullet (X) \\ &= true \end{aligned}$$

Now, consider the program described by the predicate $true \fatsemi Q$, where Q is a terminating non-recursive program. The following property is expected:

$$true \fatsemi Q = true$$

and in the opposite direction:

$$Q \fatsemi true = true$$

However, the previous results are not valid for any predicate Q. For example, take $Q = (x := e)$, then:

$$\begin{aligned} true * II \fatsemi (x := e) &= true \fatsemi (x := e) \\ &= true \fatsemi (x' = e) \\ &= true \wedge (x' = e) \\ &= (x' = e) \end{aligned}$$

In words, a non-terminating program terminates, which is a paradox. The latter will be eliminated with the theory of Designs, presented later on.

Floyd assertion and assumption. An *assertion* is the statement that a condition, c say, is *expected* to be true at the point at which it is written; otherwise, the program behaves in a totally unpredictable, chaotic way, i.e. like $\bot$. That is, the failure is caused by the programmer. An assertion captures the *intent* of the programmer, that something is meant to be true.

Definition 2.2.8 (Assertion [66, Def. 2.8.3]). $c_\perp \;\widehat{=}\; II \lhd c \rhd \perp$ □

On the other hand, an *assumption* is the statement that a condition is true at the point at which it is written; otherwise the program behaves in an impossible, miraculous way, i.e. like $\top$. That is, the failure is caused by the program. An assumption captures the *confidence* of the programmer, that something is true.

Definition 2.2.9 (Assumption [66, Def. 2.8.3]). $c^\top \;\widehat{=}\; II \lhd c \rhd \top$ □

Correctness

A relation may be used for representing programming concepts at different levels of abstraction. A *specification* is a statement of the form *if the program Q is started in a state satisfying pre and terminates, then its final state will satisfy post*. Such a specification is more readily expressed by a Hoare triple $pre\{Q\}post$.

Definition 2.2.10 (Hoare triple [66, Def. 2.8.1]). $p\{Q\}r \;\widehat{=}\; [Q \Rightarrow (p \Rightarrow r')]$ □

Both preconditions and postconditions are but conditions, hence, they may not mention dashed variables.

In practice, the same specification may be satisfied by many programs. It is therefore important of knowing if a program satisfies a given specification, in which case the program is said to be *correct* (w.r.t. the specification). Since both specifications and programs are predicates, correctness may be expressed using logical implication.

The gap between a specification and a corresponding implementation is often very big, so that many design steps may be used in between. Each design step may itself be represented by a predicate, each one less abstract than the previous one, until the final program is reached. This construction procedure is called a *stepwise refinement* and defines an ordering between programs, denoted by $\sqsubseteq$, and called the *refinement ordering*.
The correctness of a program P with respect to a specification S is denoted by $S \sqsubseteq P$ (read S is refined by P), and is defined as follows.

$$S \sqsubseteq P \textit{ iff } [P \Rightarrow S]$$

2.3 Designs

Eliminating the non-termination paradox mentioned previously is achieved by restricting the space of relations to those predicates that satisfy the zero laws:

$$true \fatsemi P = true = P \fatsemi true$$

That space may be further restricted by means of an ensemble of laws known as healthiness conditions. The resulting relations are called Designs.

Two variables ok and ok' are used to record respectively when a program has started and when it has terminated. Designs exclude the miraculous predicate *false* which does not satisfy the zero laws:

$$false \fatsemi P = false = P \fatsemi false$$

A design is described by a pair of predicates assumption-commitment, similar to specifications (precondition-postcondition) except that there is no restriction regarding the presence of dashed variables.

Definition 2.3.1 (Designs [66, Def. 3.1.1])**.** *Let P and Q be predicates not containing ok and ok'.*

$$P \vdash Q \mathrel{\widehat{=}} (ok \wedge P) \Rightarrow (ok' \wedge Q)$$

$P \vdash Q$ *expresses that* if the program starts in a state satisfying P, it will terminate in a state satisfying Q. □

Below we give the semantics of assignment.

Definition 2.3.2 (Assignment and Skip [66, Def. 3.1.3])**.**

$$\begin{aligned} x :=_{\mathsf{A}} e &\mathrel{\widehat{=}} true \vdash x' = e \wedge y' = y \wedge .. \wedge z' = z \\ II_D &\mathrel{\widehat{=}} true \vdash x' = x \wedge y' = y \wedge .. \wedge z' = z \end{aligned}$$ □

All other operators defined so far keep the same meaning on designs.

The implication ordering of the relational calculus has an equivalent order in the calculus of designs (or *refinement calculus*). That equivalence is established by the following theorem. The following formulation corresponds to Cavalcanti and Woodcock [37, Law 68]. Hoare and He [66, Th. 3.1.2] give an equivalent formulation without using the refinement symbol $\sqsubseteq$ however.

Theorem 2.3.3. $(P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2) \Leftrightarrow [P_1 \Rightarrow P_2] \wedge [P_1 \wedge Q_2 \Rightarrow Q_1]$

Proof. cf. [37, Law 68], [66, Th. 3.1.2] □

As stated earlier, taking into account the new definition of assignment, all programming operators defined in the relational calculus remain valid in the refinement calculus. So as relations form a complete lattice under implication ordering, *designs form a complete lattice under refinement ordering* [66, Th. 3.1.5]. Its bottom and top elements are denoted respectively by $\bot_D$ and $\top_D$, and defined below.

$$\begin{aligned} \bot_D &\mathrel{\widehat{=}} (false \vdash true) = true \\ \top_D &\mathrel{\widehat{=}} (true \vdash false) = \neg\, ok \end{aligned}$$

By virtue of their definitions, designs solve part of the paradox: they obey the left zero law. This fact is captured in the following law [66, Th. 3.1.2, L1]:

L3 $\mathit{true} \fatsemi (P \vdash Q) = \mathit{true}$

As defined so far predicate pairs do not yet satisfy the right zero law. In addition, further properties are desired from designs to ensure that they are actually implementable. These are captured in the following law, and are known as healthiness conditions.

Definition 2.3.4 (Design healthiness conditions [66, Def. 3.2.1])**.** *A predicate R is said to be **H1**, **H2**, **H3** or **H4** according to which of the following laws it satisfies:*

$$\begin{aligned} &\textbf{H1} && R = (ok \Rightarrow R) \\ &\textbf{H2} && R[\mathit{false}/ok'] \Rightarrow R[\mathit{true}/ok'] \\ &\textbf{H3} && R = R \fatsemi II_D \\ &\textbf{H4} && R \fatsemi \mathit{true} = \mathit{true} \end{aligned}$$ □

H1 states that no observation can be made before the program expressed by predicate R has started. ***H2*** states that non-termination cannot be a requirement.***H3*** states that the precondition of a design is a condition. This is to avoid irreversible constraints on dashed variables. For example, $x' = 2 \vdash \mathit{true}$ does not terminate if $x = 2$, but terminates otherwise. This is undesired according to the definition of designs. Effectively, if $x' = 2$ is unsatisfied, then the state of the program should not be observable. ***H4*** states that given any design $P \vdash Q$, for any initial values of the undashed variables that satisfy P, there exist final values for the dashed variables that satisfy Q. ***H4*** eliminates the miraculous predicate $\top_D$ and is called a *feasibility condition*. Furthermore, if a design does not satisfy ***H4***, there is no program that could ever implement it.

Theorem 2.3.5.

1. *A predicate is **H1** and **H2** iff it is a design.*
2. *A design $P \vdash Q$ is **H3** iff its assumption can be expressed as a condition.*
3. *$P \vdash Q$ satisfies **H4** iff $[\exists\, ok', x', .., z' \bullet (P \vdash Q)]$.*

Proof. cf. [66, Th. 3.2.2], [66, Th. 3.2.4], and [66, Th. 3.2.5] respectively. □

Summary. Both sections on relations and designs have served for the presentation of how programs and specifications are represented within UTP. In the former, a program was described by a single predicate giving the state of the observable variables of a program; more precisely, it described the relation between the initial and final values of each variable. The space of relations was later restricted because of the possible non-termination paradox. The new class thus formed, called designs, are predicate pairs which should satisfy a number of

healthiness conditions, namely ***H1***, ***H2***, ***H3*** and ***H4***. The operators so far defined are suitable for describing sequential programs only. Another UTP theory, the theory of reactive processes, permits the modelling of concurrency and communication, and will be presented in section 2.5. In the next section we describe how UTP theories may be linked (formally) together.

2.4 Linking theories

A UTP theory may be more concisely represented by a triple $(\mathsf{A}, \Sigma, \boldsymbol{HCond})$ where A is the alphabet of the theory, common to all the predicates; Σ is the union of two sets: a set of primitive predicates, and the set of operators; ***HCond*** is a monotonic and idempotent function.

A comment about ***HCond*** is necessary. In [66] and as indicated above, it is said that a UTP theory is characterised by a *set* of healthiness conditions, and not just a single function. It may hence turn out that the order of composition of given healthiness conditions matters, thus yielding different ***HCond*** functions. This would mean that two theories with the same set of distinct healthiness conditions may be considered as different if different composition orders yield different results. Notwithstanding the apparent ambiguity, it is the point of this section to show how any two theories may be related.

A UTP theory may be given a set theoretic characterisation of as the set

$$\{(\alpha P, P) \mid \alpha P \subseteq \mathsf{A} \bullet \boldsymbol{HCond}(P) = P\}$$

of all healthy predicates with a given alphabet. Thus, linking UTP theories amounts to building functions over them. It is common to use traditional functional notation to discuss such *linking functions*. They may also be viewed as particular *predicate transformers* that transform predicates of a theory into those of a distinct theory.

Let $\mathbf{S}$ and $\mathbf{T}$ denote characteristic sets of predicates of UTP theories, with $\mathbf{S}$ denoting the potentially (stronger or more expressive) theory, and $\mathbf{T}$ the potentially (weaker or less expressive) theory. Let $L : \mathbf{S} \to \mathbf{T}$ and $R : \mathbf{T} \to \mathbf{S}$ denote two linking functions.

A very common kind of link that may be built is that between a theory and any *subset* of it.

Definition 2.4.1 (Subset links, weakening, strengthening [66, Def. 4.1.2])**.** *Let* $\mathbf{T} \subseteq \mathbf{S}$*. Then,* $R : \mathbf{T} \to \mathbf{S}$ *is simply the identity function.* $L : \mathbf{S} \to \mathbf{T}$ *is called* weakening *if*

$$\forall\, X \in \mathbf{S} \bullet L(X) \sqsubseteq X$$

and strengthening *otherwise i.e. if*

$$\forall X \in \mathbf{S} \bullet X \sqsubseteq L(X) \qquad \square$$

The formal definition of a *link* is given hereafter.

Definition 2.4.2 (Link, retract [66, Def. 4.1.11])**.** *A function that is both weakening and idempotent is a* link*. A link that is monotonic is called a* retract*.* $\square$

Another form of link is one that maps a theory to itself, and is called a bijection. Bijections are most useful when one wants to show that two theories apparently dissimilar from the outset (e.g. they use different mathematics, hence resulting in different alphabets and healthiness conditions) are equally expressive.

Definition 2.4.3 (Bijection [66, §4.2])**.** *A function L is a* bijection *iff, $R = L^{-1}$, where L^{-1} exists, and the following identities hold for all P*

$$L \circ R(P) = P \ \wedge \ R \circ L(P) = P \qquad \square$$

For theories that have different expressiveness, the linking function is often not bijective. Hence, links must be either strengthened or weakened accordingly. The corresponding (L, R) pair of functions is called a Galois connection.

Definition 2.4.4 (Galois connection [66, Def. 4.2.1])**.** *Let $\mathbf{S}$ and $\mathbf{T}$ be two lattices, and let $L : \mathbf{S} \to \mathbf{T}$ and $R : \mathbf{T} \to \mathbf{S}$. The pair (L, R) is a* Galois connection *iff*

$$\forall X \in \mathbf{S}, Y \in \mathbf{T} \bullet R(Y) \sqsubseteq X \Leftrightarrow Y \sqsubseteq L(X) \qquad \square$$

A subset relation may be seen as a form of Galois connection; a bijection is a stronger relation than a Galois connection: not every bijection is a Galois connection.

2.5 Reactive Processes

The UTP theory of Reactive Processes concerns programs that may interact with their environment. Reactive programs are expressed as *processes*, i.e. predicates that allow characterising the intermediate states of a program, between initialisation and termination.

The interactions of a process are modelled as atomic *events*, i.e. actions without a duration. A process may engage in given events only, which are thus said to be *authorised*. Authorised events form a set called the *actions set* or *events alphabet* of the process, and denoted by $\mathcal{A}$ ($\mathcal{A}P$ for a process P).

Each occurrence of an event is recorded in order. The resulting sequence is called the *trace* of the process, denoted by the variable $tr : \mathcal{A}^*$. tr gives the trace at the beginning of (i.e. prior to) the current observation, and $tr' - tr$ gives the trace from start to termination.

A process or its environment may refuse to engage in an event. All the events that may be refused constitute the refusal set of the process, denoted by the variable $ref : \mathbb{P}\,\mathcal{A}$. *ref*

gives events which may be refused during the current observation and ref' give those that may be refused next.

The boolean variable $wait : \mathbb{B}$ is used to represent waiting states. $wait = true$ means that the predecessor is in a waiting state, i.e. the process is waiting for its predecessor to terminate, meanwhile it does nothing. When used in conjunction with the boolean variable $ok : \mathbb{B}$ it also permits representing termination.

The variable ok determines if the process is in a stable state (i.e. not making any progress). $ok = false$ means that the current process has not yet started . If $ok = true$ then the process has started and its predecessor is stable. If $ok' = true$ and $wait' = true$ then the process is stable but in an intermediate state. If $ok' = true$ and $wait' = false$ then the process has terminated. If $ok' = false$ then the process is in a non-stable state and the values of other variables are meaningless: the process *diverges*.

The variable name $\mathbf{o}$ (resp. $\mathbf{o}'$) is often used to denote all the variables in the set $\{ok, wait, tr, ref\}$, also called the *observational variables* of a process.

In summary, the alphabet of a reactive process consists of the following:

- $\mathcal{A}$, the set of authorised events ; $tr, tr' : \mathcal{A}^*$, 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.

The above alphabet alone is not enough to characterise reactive processes. Predicates with such an alphabet must also satisfy the following healthiness conditions.

$$\textbf{\textit{R1}} \quad P = P \wedge tr \leq tr'$$

$$\textbf{\textit{R2}} \quad P = \bigsqcap\{P[s, s \frown (tr' - tr)/tr, tr'] \mid s \in \mathcal{A}^*\}$$

$$\textbf{\textit{R3}} \quad P = (II_R \lhd wait \rhd P) \qquad \textbf{where}$$

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

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 behaves like II_R when its predecessor had not yet terminated.

Alternatively, the single healthiness condition $\boldsymbol{R}$ may be used for characterising reactive processes: $\boldsymbol{R} = \textbf{\textit{R1}} \circ \textbf{\textit{R2}} \circ \textbf{\textit{R3}}$.

A particular model of reactive processes is provided by the CSP process algebra [65], [106], whose semantics in UTP are presented subsequently.

2.5.1 CSP processes semantics

CSP processes are reactive processes that obey the following additional healthiness conditions:

$$\begin{array}{lll} \textbf{\textit{CSP1}} & P & = P \lhd ok \rhd tr \leq tr' \\ \textbf{\textit{CSP2}} & P & = P ;; J \qquad\qquad \textbf{where} \\ & J & \mathrel{\widehat{=}} (ok \Rightarrow ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge v' = v) \end{array}$$

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 may always terminate. It characterises the fact that divergence may never be enforced.

Alternatively, the single healthiness condition ***CSP*** may be used for characterising CSP processes: $\textbf{\textit{CSP}} = \boldsymbol{R} \circ \textbf{\textit{CSP1}} \circ \textbf{\textit{CSP2}}$.

We present the semantics of some CSP processes subsequently. Some definitions are similar to the ones presented in section 2.2, with some changes. For example, the definitions mention new alphabet elements, and certain healthiness conditions are applied directly, for e.g. assignment below.

Assignment. Denoted by $x := e$, is the process that sets the value of the variable x to e on termination, but does not modify the other variables. It does not interact with the environment, always terminates and never diverges [126, Def. 4].

Definition 2.5.1. $(x := e) \mathrel{\widehat{=}} \textbf{\textit{R3}} \circ \textbf{\textit{CSP1}}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge x' = e \wedge v' = v)$ □

II_R, a particular kind of assignment that leaves everything unchanged, has already been seen above.

Skip. Denoted by *SKIP*, is the process that refuses to engage in any event, terminates immediately and does not diverge. It is special kind of II_R [66, e.g. 8.2.3(3)].

Definition 2.5.2 (*SKIP*)**.** $SKIP \mathrel{\widehat{=}} \exists\, ref \bullet II_R$ □

Stop. Denoted by *STOP*, is the process that is unable to interact with its environment. It is always in a waiting state [66, e.g. 8.2.3(1)].

Definition 2.5.3 (*STOP*)**.** $STOP \mathrel{\widehat{=}} \boldsymbol{R}(wait := true)$ □

Chaos. Denoted by *CHAOS*, is the process with the most non-deterministic behaviour viz. the worst possible reactive process [66, e.g. 8.2.3(4)].

Definition 2.5.4 (*CHAOS*)**.** $CHAOS \mathrel{\widehat{=}} \boldsymbol{R}(true)$ □

Sequential composition. Denoted by $P \fatsemi Q$, is the process that first behaves like P, and if P terminates, then behaves like Q [66, e.g. 8.2.3(4)].

Definition 2.5.5. *Provided* $out\alpha P = in\alpha' Q = \{\mathbf{o}', v'\}$,

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

Substitution. Denoted by $P[e/x]$, is the behaviour of P when the variable x is replaced the expression e. It may also be denoted by $P(e)$. The effect of substitution is defined by the following law:

Definition 2.5.6. $(x := e \fatsemi P) \;=\; P[e/x]$ $\square$

Conditional choice. Denoted by $P \lhd b \rhd Q$, is the process that behaves either like P if the condition b is true or like Q otherwise.

Definition 2.5.7. $P \lhd b \rhd Q \;\widehat{=}\; (b \wedge P) \vee (\neg\, b \wedge Q)$ $\square$

Iteration. Denoted by $b * P$, is the process that behaves like P if the condition b is true, and then again when P terminates, and so on until b is false, and then terminates.

Definition 2.5.8. $b * P \;\widehat{=}\; \mu X \bullet ((P \fatsemi X) \lhd b \rhd SKIP)$ $\square$

Internal choice. Denoted by $P \sqcap Q$, is the process that behaves either like P or like Q, where the choice cannot be controlled by the environment.

Definition 2.5.9. $P \sqcap Q \;\widehat{=}\; P \vee Q \qquad$ *provided* $\alpha P = \alpha Q$ $\square$

External choice. Denoted by $P \,\Box\, Q$, is the process that behaves like either P or Q, where the choice is controlled by the environment [66, Def 8.2.7].

Definition 2.5.10. *Let* $\mathcal{A}P = \mathcal{A}Q$, $P \,\Box\, Q \;\widehat{=}\; \boldsymbol{CSP2}((P \wedge Q) \lhd STOP \rhd (P \vee Q))$ $\square$

External choice supposes that P and Q may offer the same set of events initially. The definition states that if no interaction is performed and termination has not occurred i.e. $STOP = true$ then the observation must be agreed by both P and Q. Otherwise the behaviour will be either that of P or Q depending on which one the environment chose to interact with. Notice that $STOP$ appears as a *condition*, and not as a *process*. This is simply thanks to the definition of conditional, and the fact that $STOP$ is *also* a predicate.

Prefix. Denoted by $a \rightarrow P$, is the process that engages in action a and then behaves like process P. First, consider the following predicate transformer [66, Th. 8.1.3], [84, Defs. 15, 16].

Definition 2.5.11 (Φ)**.** $\Phi \mathrel{\widehat{=}} \boldsymbol{R} \circ and_B = and_B \circ \boldsymbol{R}$
where $and_B \mathrel{\widehat{=}} B \wedge X$*, and* $B \mathrel{\widehat{=}} (tr' = tr \wedge wait') \vee tr < tr'$*, and* X *denotes any predicate of a given UTP theory.* □

and_B imposes the condition B on any predicate X: if X is waiting for an event to occur then it may not modify the trace, otherwise the only possible observation is trace expansion. Φ makes the predicate X into a reactive process and characterises the values of the other variables when $wait' = true$.

The occurrence of an event a is defined as the process denoted by $do_{\mathcal{A}}(a)$. It never refuses to engage in a whilst it is waiting for a to occur. Following its occurrence, a is recorded in the trace [66, e.g. 8.1.4)], [84, Def. 17].

Definition 2.5.12 (Do)**.** $do_{\mathcal{A}}(a) \mathrel{\widehat{=}} \Phi(a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle)$ □

a may be used as an abbreviation for $do_{\mathcal{A}}(a)$. The process denoted by $a \to SKIP$ first engages in a and then terminates successfully [84, Def. 19].

Definition 2.5.13 (Simple prefix)**.** $a \to SKIP \mathrel{\widehat{=}} \boldsymbol{CSP1}(ok' \wedge do_{\mathcal{A}}(a))$ □

The process denoted by $a \to P$ first engages in event a and then behaves like some process P [84, Def. 18)], [66, Def. 8.2.5].

Definition 2.5.14 (Prefix)**.** $a \to P \mathrel{\widehat{=}} a \to SKIP \fatsemi P$ □

Renaming. Denoted by $p.P$, is the process that behaves like P except that any event name a in P has been replaced by a name $p.a$ [66, Chap. 8], [65].

Definition 2.5.15 (Renaming)**.**

$$\begin{aligned} p.P &\mathrel{\widehat{=}} P[p.a \leftarrow a] \\ \mathcal{A}(p.P) &\mathrel{\widehat{=}} \{p.a \mid a \in \mathcal{A}(P)\} \end{aligned}$$ □

The definition above applies also if both a and $p.a$ stand for (ordered) lists of events.

Hiding. Denoted by $P \setminus X$, is the process that behaves like P but in which the events of the set X occur silently, without the participation or even the knowledge of the environment of P [126, Def. 18], [66, Def. 8.2.14].

Definition 2.5.16 (Hiding)**.**

$$\begin{aligned} \mathcal{A}(P \setminus X) &\mathrel{\widehat{=}} \mathcal{A}P - X \\ P \setminus X &\mathrel{\widehat{=}} \exists\, tra, refa \bullet \begin{pmatrix} P[tra, refa / tr', ref'] \wedge \\ tra = tr' \restriction (\mathcal{A}P - X) \wedge \\ refa = ref' \cup X \end{pmatrix} \fatsemi SKIP \end{aligned}$$ □

Communication. A particular type of interaction between processes is the passing or communication of messages. A *communication* is the sending or the reception of a message via a channel (the communication medium). A communication event is represented by a pair $ch.e$ where ch denotes the (name of the) channel used for the communication, and e any message that may be sent through ch. The set of all the channels that a process may use is called its *interface*; (in this thesis) it will be denoted by $\mathcal{I} \mathrel{\widehat{=}} \{ch \mid \exists\, m \bullet ch.m \in \mathcal{A}\}$.

The occurrence of a communication event $ch.e$ is just the process $do_{\mathcal{A}}(ch.e)$. As earlier $ch.e$ may be used as an abbreviation for $do_{\mathcal{A}}(ch.e)$. The definitions below correspond to a *synchronous model of communication.*

Definition 2.5.17 (Communication)**.** $ch.e \mathrel{\widehat{=}} do_{\mathcal{A}}(ch.e)$ □

The input prefix $ch?x \to P$ receives a message from ch, assigns it to the variable x and then behaves like process P.

Definition 2.5.18 (Input prefix [126, Def. 13])**.** $ch?x \to P \mathrel{\widehat{=}} \Box_{e \in Msg} ch.e \fatsemi (x := e) \fatsemi P$ □

The output prefix $ch!y \to P$ is the process that is willing to output the value of the variable y through channel ch first, and then behaves like P.

Definition 2.5.19 (Output prefix [126, Def. 14])**.** $ch!y \to P \mathrel{\widehat{=}} ch.y \fatsemi P$ □

Parallel composition. Denoted by $P \parallel Q$, is the process that behaves like both P and Q and terminates when both have terminated. P and Q may not share any variable other than the observational variables $(ok, wait, ...)$. P and Q modify separate copies of the shared observational variables which are then merged at the end using the merge predicate M, as defined below.

Definition 2.5.20 (Parallel composition [126, Def. 16])**.**

$$\begin{aligned}
\mathcal{A}(P \parallel Q) &\mathrel{\widehat{=}} \mathcal{A}P \cup \mathcal{A}Q \\
P \parallel Q &\mathrel{\widehat{=}} P(\mathbf{o}, 1.\mathbf{o}') \wedge Q(\mathbf{o}, 2.\mathbf{o}') \fatsemi M(1.\mathbf{o}, 2.\mathbf{o}, \mathbf{o}') \\
M &\mathrel{\widehat{=}} \begin{pmatrix} ok' = (1.ok \wedge 2.ok) \wedge \\ wait' = (1.wait \vee 2.wait) \wedge \\ ref' = (1.ref \cup 2.ref) \wedge \\ (tr' - tr) = \big((1.tr - tr) \parallel (2.tr - tr)\big) \end{pmatrix} \fatsemi SKIP
\end{aligned}$$

Upon termination the final trace is given by the trace merge function defined subsequently. 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: $\{a, b\} \notin E(s) \cap E(t)$, $\{c, d\} \in E(s) \cap E(t)$. The trace merge for parallel composition may be defined by case as follows:

$$s \parallel t \mathrel{\widehat{=}} t \parallel s \qquad \langle\rangle \parallel \langle\rangle \mathrel{\widehat{=}} \{\langle\rangle\} \qquad \langle\rangle \parallel \langle c\rangle \mathrel{\widehat{=}} \{\} \qquad \langle\rangle \parallel \langle a\rangle \mathrel{\widehat{=}} \{\langle a\rangle\}$$

$$\langle c\rangle \frown x \parallel \langle c\rangle \frown y \mathrel{\widehat{=}} \{\langle c\rangle \frown u \mid u \in x \parallel y\}$$
$$\langle a\rangle \frown x \parallel \langle c\rangle \frown y \mathrel{\widehat{=}} \{\langle a\rangle \frown u \mid u \in x \parallel \langle c\rangle \frown y\}$$
$$\langle c\rangle \frown x \parallel \langle d\rangle \frown y \mathrel{\widehat{=}} \{\}$$
$$\langle a\rangle \frown x \parallel \langle b\rangle \frown y \mathrel{\widehat{=}} \{\langle a\rangle \frown u \mid u \in x \parallel \langle b\rangle \frown y\} \cup \{\langle b\rangle \frown u \mid u \in x \parallel \langle a\rangle \frown y\} \qquad \square$$

Pipes, Chaining. A pipe [106], [65], [125], is a type of process that inputs on one channel usually called *left*, and outputs on another channel usually called *right*. The simplest kind of pipes is composed of only one process.

Example 2.5.21. *A simple pipe which outputs the square of any number it has input:*

$$Sq \mathrel{\widehat{=}} \mu X \bullet (left?x \to right!(p * p) \to X) \qquad \square$$

More complex pipes may be formed by connecting in a sequence a number of (simple) pipes such that the $j - th$ pipe feeds its inputs to the $(j+1) - th$ one: such a composition is called *chaining*, and is defined below.

Definition 2.5.22 (Chaining [125, Def. 8])**.**

$$P \gg Q \mathrel{\widehat{=}} (P[mid \leftarrow right] \parallel Q[mid \leftarrow left]) \setminus \{mid\} \qquad \square$$

Notation: $P \langle\!\langle\, right \leftrightarrow left \,\rangle\!\rangle\, Q$ could have been used instead of $P \gg Q$. This second notation is more useful for denoting more than one chain.

By indexing the pipes a chaining order may be defined.

Definition 2.5.23 (Indexed-chaining [125, Def. 9])**.**

$$(\gg i : 1 \mathinner{..} n \bullet P_i) \mathrel{\widehat{=}} \begin{cases} P_1 & n = 1 \\ (\gg i : 1 \mathinner{..} n-1 \bullet P_i) \gg P_n & n > 1 \end{cases}$$

$\square$

Two pipes may be chained such as to form a ring.

Definition 2.5.24 (Circular-chaining [125, Def. 10])**.**

$$P \ll\gg Q \mathrel{\widehat{=}} (P \begin{bmatrix} mid_1 \leftarrow right, \\ mid_2 \leftarrow left \end{bmatrix} \parallel Q \begin{bmatrix} mid_1 \leftarrow left, \\ mid_2 \leftarrow right \end{bmatrix}) \setminus \{mid_1, mid_2\} \qquad \square$$

Under the other notation, circular chaining may be denoted equivalently by $P \langle\!\langle\, right \leftrightarrow left, left \leftrightarrow right \,\rangle\!\rangle\, Q$, or by $P \langle\!\langle\, right, left \leftrightarrow left, right \,\rangle\!\rangle\, Q$.

Catastrophic interrupt. McEwan & Woodcock [84] have defined an interrupt operator called the catastrophic interrupt as a kind of sequential composition $P \triangle_i Q$ where control can pass from P to Q even when P is in a non-terminating (or intermediate) state, i.e. $P.wait' = true$. This is unlike normal sequential composition and is reflected in the following law, defined as a healthiness condition allowing interference, and denoted by ***I3***.

Definition 2.5.25 (Interference ***I3*** [84, Def. 26])**.** $\boldsymbol{I3}(Q) = Q \lhd wait \rhd II_R$ □

In [84], two treatments of interrupt, one called unconditional and the other conditional, are considered. We will present the unconditional interrupt only.

Definition 2.5.26 (Catastrophic interrupt [84, Defs. 27 to 34])**.**

$$\begin{aligned} P \triangle_i Q &\mathrel{\widehat{=}} \boldsymbol{R3} \circ \boldsymbol{CSP2}(P^{+i} \fatsemi i \triangle Q) \\ P^{+i} &\mathrel{\widehat{=}} P \wedge (i \notin ref' \lhd wait' \rhd P) \\ i \triangle Q &\mathrel{\widehat{=}} \boldsymbol{CSP1}(ok' \wedge force(i, Q)) \\ force(i, Q) &\mathrel{\widehat{=}} \boldsymbol{I3}(try(i, Q)) \\ try(i, Q) &\mathrel{\widehat{=}} ((i \notin ref' \wedge II_R) \lhd wait' \rhd tr' = tr \frown \langle i \rangle) \fatsemi Q \end{aligned}$$ □

In the notation $P \triangle_i Q$, P is called the *interruptible* process and Q is called the *interrupting* process. The definition states that when P is in a waiting state $P.wait' = true$, the interrupt event i might occur and when it does then Q will be executed. If, however, i does not occur, then P will resume its execution without Q being capable of interfering until the next waiting of P. In detail:

- P^{+i} is the process that behaves like P when P is not in a waiting state; otherwise, when P is waiting, it may 'witness' the occurrence of i. This is simply a specification format: P^{+i} states that i may occur when P is waiting, but does not prescribe how that is supposed to happen.
- $try(i, Q)$ is simply a syntactic sugar for $do(i) \rightarrow Q$. $force(i, Q)$ is the process that tries doing event i although its predecessor has not terminated, hence breaking ***R3***. $i \triangle Q$ simply enforces that the predecessor must, however, be stable i.e. $ok' = true$, when trying to enforce the occurrence of i.
- $P^{+i} \fatsemi i \triangle Q$ is the process that explicitly places P in an environment in which i may occur when $P.wait' = true$. Such a process is then made healthy.

Reactive Designs. The following theorem establishes that reactive processes may be expressed as designs.

Theorem 2.5.27 (Reactive designs)**.** *For every CSP process P:*

$$P = \mathbf{R}(\neg\, P[false/wait, false/ok'] \vdash P[false/wait, true/ok'])$$

Proof. cf. [37, Th. 12], [66, Th. 8.2.2]. □

The CSP processes defined so far may be called *static* since their interface does not change throughout their activation. For that reason the model for CSP processes presented so far will be referred to as the *static model*. As there is no explicit representation of *time*, the model may also be referred to as the *untimed* (static) CSP model.

We do not present timed CSP models. References to such models in UTP are [134], [118]. In the next section we present the UTP semantics for continuations.

2.6 Continuations

2.6.1 Steps and Assembly of Steps

Implementing a program consists of adding details related to the program's execution on a given platform: the result is called an *implementation*. A detail of particular importance relates to **control flow**, or the order of the execution of the instructions in the program. A device called the *program counter* normally computes and stores the value of the address of the next instruction to be executed. When executing a program, the processor always refers to the program counter.

In UTP, the program counter is represented by a variable, denoted by l, and referred to as the *control variable*. The set of possible values that l can take is called *continuations set* or simply continuations, and is denoted by αl (αlP, the continuations of a predicate P). The *instructions* of the program are represented by *steps*, which are themselves predicates. An implementation may consist of a 'single' step or of an assembly of such steps.

First, consider programs that may be represented as the sequential repetition of a single step. The value of l is tested before each repetition of the step and determines if the execution of the step starts, continues, or ends. Hence, l does also specify *termination*.

Definition 2.6.1 (Continuations and execution [66, Def. 6.1.1])**.**

$$P^* \mathrel{\widehat{=}} (l \in \alpha lP) * P$$

αlP denotes the set of continuations of P; $l \in \alpha lP$ denotes the control variable for its execution; and P^* denotes the execution of P, defined as a loop that iterates the step as long as l remains in αlP. □

For a step P, the value of l determines the start and termination of its execution. When l is outside the continuations of P, P is not even started. Although the behaviour of P in such case may be anything, it is safe to define that it does nothing, i.e. its behaviour is II. This is a sound assumption, considering the execution of P in conjunction with that of other steps.

Definition 2.6.2 (Step [66, Def. 6.1.5]). *A predicate P is a* step *if $l \in \alpha lP$ and*

$$P = P \lhd l \in \alpha lP \rhd II$$

As a consequence,

$$((l \notin \alpha lP)_\perp \fatsemi P) = (l \notin \alpha lP)_\perp \qquad \square$$

Definition 2.6.3 (Continuations of operators [66, Def. 6.1.6]).

$$\alpha\, l(P \,\mathbf{op}\, Q) \;\widehat{=}\; \alpha\, lP \cup \alpha\, lQ \qquad \textit{where } \mathbf{op} \in \{\fatsemi, \sqcap, \lhd b \rhd\} \qquad \square$$

The following theorem gives the closure property of some operators.

Theorem 2.6.4 (Step closure). *If P and Q are steps, then*

1. *$P \fatsemi Q$ is a step.*
2. *$P \sqcap Q$ and $P \lhd b \rhd Q$ are also steps whenever $\alpha lP = \alpha lQ$.*
3. *The set of steps is a complete lattice.*

Proof. cf. [66, Th. 6.1.7]. $\square$

By definition, a predicate is a step if l is in its alphabet. This means that a step may be arbitrarily complex. In particular, it can contain familiar programming notations, although it makes it necessary to specify the value of l.

The following theorem states that it is possible to determine the first action of the execution of a step.

Theorem 2.6.5. $P^* = P \fatsemi P^*$

Proof. cf. [66, Lemma 6.1.10]. $\square$

A step is executed exactly once if its execution is guaranteed to assign to l a value outside its continuations.

Theorem 2.6.6. $P^* = P \quad \textit{iff } P = P \fatsemi (l \notin \alpha lP)_\perp$

Proof. cf. [66, Lemma 6.1.10]. $\square$

Programs may occupy disjoint storage areas, in which case they are said to be disjoint. This means that two steps that have disjoint continuations are disjoint. It is possible to assemble them into a single program, by using the assembly operator defined below.

Definition 2.6.7 (Assembly [66, Def. 6.1.14]). *Let P and Q be disjoint steps, i.e. $\alpha lP \cap \alpha lQ = \{\}$.*

$$\begin{aligned} P \boxplus Q &\;\widehat{=}\; (P \lhd l \in \alpha lP \rhd Q) \lhd (l \in \alpha lP \cup \alpha lQ) \rhd II \\ \alpha l(P \boxplus Q) &\;\widehat{=}\; \alpha lP \cup \alpha lQ \end{aligned} \qquad \square$$

There are two known ways of implementing a program: compilation and interpretation. In what follows we present the former only.

2.6.2 Compilation

Compilation is the transformation of a program into a target program expressed in the machine code of the machine that is to execute it. Compilation preserves the meaning of the source program.

The semantics of the target language (or machine code) may equally be given in UTP. Precisely, each instruction in the target language may be given a meaning as a step.

A single instruction is a step with a single continuation given by the singleton set $\{m\}$.

Definition 2.6.8 (Single instruction [66, Def. 6.2.2])**.** *If INST is a machine code instruction then*

$$m : INST \mathrel{\widehat{=}} INST \lhd l = m \rhd II$$

is a single instruction. □

Single instructions may be assembled together using the assembly operator, denoted (in this thesis) by ⧉ (instead of ⟦ [66, Chap. 6]).

Definition 2.6.9 (Machine code block [66, Def. 6.2.3])**.** *A* machine code block *is a program expressed as an assembly of single instructions*

$$S_0 ⧉ S_1 ⧉ ... ⧉ S_n$$ □

Using the preceding definition, it is possible to enter a machine code block at any of its constituent continuation points. In practice, it is common to define a *normal starting point*, denoted by s, and a *normal finishing point*, denoted by f. They relate respectively to the first and last single instructions of the program. s is the value of l when control enters sequentially into the program; any other point should be entered by a jump. f is the value of l when control leaves sequentially through the last instruction. Respectively in each case, we will also talk about *normal start or entry* and *normal termination or exit*.

The assumption of normal entry is expressed by the predicate $(l = s)^\top$. The obligation to terminate normally is expressed by the assertion $(l = f)_\bot$. Machine code blocks that have these pre- and post-conditions are called *structured*.

Definition 2.6.10 (Structured block [66, Def. 6.2.4])**.** *A* structured block *is a program of the form*

$$(l = s)^\top ⨟ P^* ⨟ (l = f)_\bot$$

where P is a machine code block. The value of s is called its starting point *and the value of f its* finishing point. □

Let $\widehat{P}$ denote the target program into which a source program P has been compiled by a compiler. $\widehat{P}$ should have the same effect (or behaviour) as P, or better. $l \in \alpha\widehat{P}$ but $l \notin \alpha P$.

$$P \sqsubseteq (\mathbf{var}\, l \fatsemi \widehat{P} \fatsemi \mathbf{end}\, l)$$

Definition 2.6.11 (Target code [66, Def. 6.2.9])**.** *A program is in* target code *if it is expressed in the form*

$$\langle s, P, f\rangle \mathrel{\widehat{=}} \mathbf{var}\, l \fatsemi (l = s)^{\top} \fatsemi P^* \fatsemi (l = f)_{\perp} \fatsemi \mathbf{end}\, l$$

where P is a machine code block. An equivalent formulation is :

$$\langle s, P, f\rangle \mathrel{\widehat{=}} \mathbf{var}\, l := s \fatsemi P^* \fatsemi (l = f)_{\perp} \fatsemi \mathbf{end}\, l \qquad \square$$

According to the fundamental theorem of compilation [66, Th. 6.2.10], every program can be expressed in target code.

Theorem 2.6.12 (Fundamental theorem of compilation)**.** *Every program can be expressed in target code.*

Proof. cf. [66, Th. 6.2.10] $\square$

It is possible to combine low-level language features such as jumps and labels with high-level language features. Such a facility was provided by many early programming languages.

2.6.3 High-level language with jumps and labels

For the combination to be possible it is necessary to consider, in addition to s and f, other continuation points viz. those corresponding to entry and exit by a jump. For convenience, a special value denoted by $\mathbf{n}$ will denote either s or f, accordingly. $\alpha l_0 P$ will denote the set of all entry points; $\alpha l' P$ will denote the set of all exit points; none of them contains $\mathbf{n}$. If l takes its value in either of these sets, it will signify that the program has been entered or exit by a jump respectively, by opposition to normal entry and exit through $\mathbf{n}$.

Definition 2.6.13 (Blocks and proper blocks [66, Def. 6.4.2])**.** *Let S and F be sets of labels (continuation points), and $\mathbf{n} \notin S$, and $\mathbf{n} \notin F$.*

$$(P : S \Rightarrow F) \mathrel{\widehat{=}} P = (P \fatsemi (l \in F \cup \{\mathbf{n}\})_{\perp}) \lhd l \in S \cup \{\mathbf{n}\} \rhd II)$$

A program is a block *if it satisfies*

$$P : \alpha l_0 P \Rightarrow \alpha l' P$$

A block is called a proper block *if*

$$\alpha l_0 P \cap \alpha l' P = \{\} \qquad \square$$

The construction **label** s permits placing a label within the program at the point intended to be the destination of a jump. It may be entered normally or by a jump, but it always exits normally. The construction **jump** f permits jump-ing to the location indicated by the label f. It may be entered normally or by a jump, but it always exits by a jump.

Definition 2.6.14 (Labels and jumps [66, Def. 6.4.5])**.**

$$\mathbf{label}\, s \mathrel{\widehat=} (l := \mathbf{n}) \lhd l \in \{s, \mathbf{n}\} \rhd II \qquad \alpha l_0 \mathbf{label}\, s \mathrel{\widehat=} \{s\} \quad \alpha l' \mathbf{label}\, s \mathrel{\widehat=} \{\}$$
$$\mathbf{jump}\, f \mathrel{\widehat=} (l := f) \lhd l = \mathbf{n} \rhd II \qquad \alpha l_0 \mathbf{jump}\, f \mathrel{\widehat=} \{\} \quad \alpha l' \mathbf{jump}\, f \mathrel{\widehat=} \{f\} \qquad \square$$

The following theorem gives the permitted operators for blocks having the same alphabets of entry and exit points.

Theorem 2.6.15 (Block closure)**.** *The set of blocks* $\{P \mid P : S \Rightarrow F\}$ *is a complete lattice, and closed with respect to non-deterministic choice and conditional. The same applies to proper blocks.*

Proof. cf. [66, Th. 6.4.7]. $\square$

Before giving the closure for sequential composition, we first give its continuations. A sequential composition $P \fatsemi Q$ may be entered normally through $\mathbf{n}$, or by a jump. In the second case, the entry point may belong to either P or Q. Similarly, it may be exit normally through $\mathbf{n}$, or by a jump from either P or Q.

Definition 2.6.16 (Continuations for sequential composition [66, Def. 6.4.8])**.**

$$\begin{aligned} \alpha l_0(P \fatsemi Q) &\mathrel{\widehat=} \alpha l_0 P \cup \alpha l_0 Q \\ \alpha l'(P \fatsemi Q) &\mathrel{\widehat=} (\alpha l' P \setminus \alpha l_0 Q) \cup \alpha l' Q \end{aligned} \qquad \square$$

Theorem 2.6.17 (Sequential composition closure)**.** *If* $P : S \Rightarrow F$ *and* $Q : T \Rightarrow G$*, then:*

$$(P \fatsemi Q) : S \cup T \Rightarrow ((F \setminus T) \cup G)$$

Proof. cf. [66, Th. 6.4.9] $\square$

2.7 Final considerations

Number of programming constructs and paradigms have been formalised using UTP that have not been mentioned. Parts of the programming language *Circus* [140, 36, 133] have been formalised, and formalisation of other aspects of *Circus* are still ongoing. In the long run, we plan to extend *Circus* with the semantics presented in this thesis.

Pointers have been formalised in [37, 59]. We expect to be able to give a new encoding for pointers in UTP based on the proposed semantics for channel mobility (Chapter 4).

We have left out the presentation of UTP theories of CSP+time [118], [134]. Timed models bear a particular interest for us due to the different interrupt operators that have been expressed on their basis, notably in [135] and [70]. Although links between timed and untimed models have been established, it seems to be a rather circumvented approach that which would consist in defining timed CSP processes only to discard time later on via some encapsulation or hiding. An interesting paper [143] discusses the relation between the external choice operator $\Box$ and time. The discussions therein show that much can still be said about timed models in CSP. We discuss the case of the interrupt operators (in untimed CSP) in greater detail in §5.4.

The interplay between time and mobility is an interesting topic but timed models are already complex on their own, so we leave that out for future research and focus first on defining a most simple model as we possibly can.

Finally, Reactive Designs [35], [134], which provide a distinct formulation for reactive processes (including CSP processes), have not really been considered in this thesis. It may be the topic of further research than to recast our results in the UTP theory of Reactive Designs (both with and without time).

Chapter 3

Literature Review

3.1 Mobile Processes

3.1.1 Code Mobility

The description of mobility requires first the definition of what entity moves and in which space. In [49], Fugetta et al. provide a survey of mobile systems following three axes: mobile code technologies, used for implementation; design paradigms, which define architectural abstractions (abstraction from machines and implementations); and application domains. In particular, they present a "Virtual Machine for Code Mobility". *Code Mobility*, the key concept used therein for describing mobile systems, is defined as "*the capability to dynamically change the bindings between code fragments and the location where they are executed*". Systems that implement (use) code mobility are called *Mobile Code Systems* (MCSs). Their architecture is captured in *Figure3.1.*

The figure in the blue (first, top-right) circle represents the layered architecture of MCSs, in which the layers on top use services provided by lower layers. At the bottom is the machine or Host in which the computation is carried. Immediately above is the Core Operating System (COS), which provides basic services such as file system, memory management and process support; it does not provide communication nor distribution services. These are supported by the Network Operating System (NOS) through channels and communication protocols. Above the NOS is the Computational Environment (CE): it provides facilities or mechanisms for realising code mobility; e.g. Programming Languages for programming code mobility, also called Mobile Code Languages (MCLs), could belong to this layer. At the top of the architecture are the components hosted by the CE, which are of two kinds: Executing Units (EUs) whose structure is shown in the red (second, lower) circle, and resources (e.g. data, network devices). EUs represent sequential flows of computation. An EU is composed of a code segment, an execution state and a data space.

In MCSs the following entities may move: code, data and EUs. The space in which they move is the network of computers or more abstractly, the network of EUs. An EU is located relatively to its CE: *EUs move from one CE to another.*

Two forms of mobility are considered:

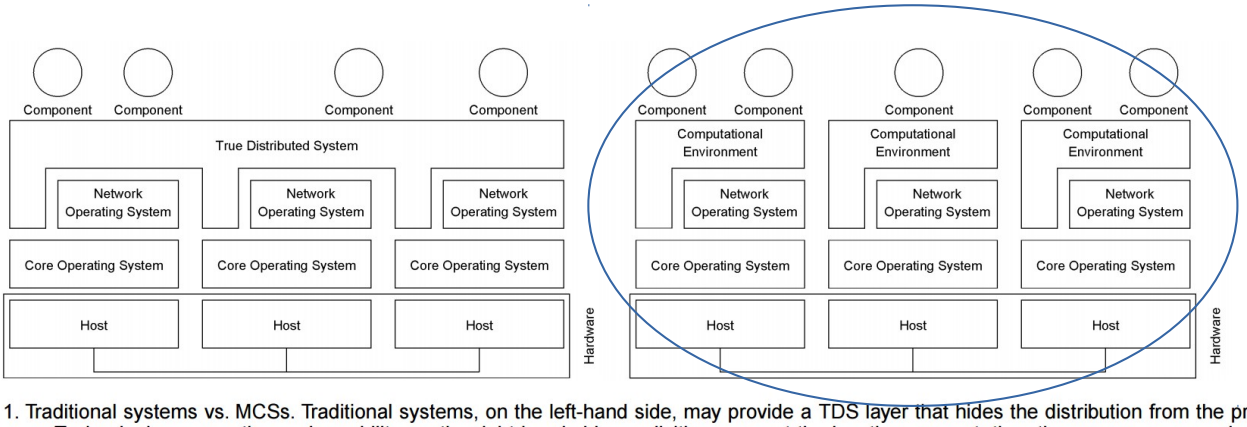

Fig. 1. Traditional systems vs. MCSs. Traditional systems, on the left-hand side, may provide a TDS layer that hides the distribution from the programmer. Technologies supporting code mobility, on the right hand side, explicitly represent the location concept, thus the programmer needs to specify *where*—i.e., in which CE—a computation has to take place.

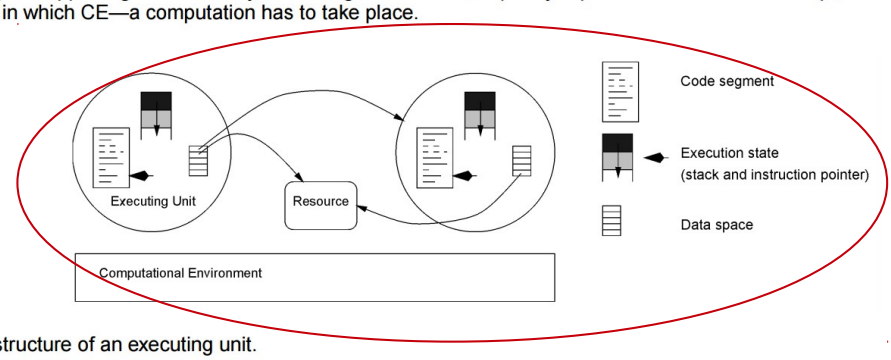

Fig. 2. The internal structure of an executing unit.

Figure 3.1: A Virtual Machine for Code Mobility ([49])

- *strong mobility*: both the code and the execution state of an EU may be moved from a CE to another CE; and
- *weak mobility*: only the code of an EU may be transferred. This code could be accompanied by some initialisation data, but the execution state cannot be communicated.

Furthermore, two mechanisms support strong mobility:

- *migration*: the EU suspends its execution, is transmitted to the destination CE where it resumes its execution. The copy of the EU in the original CE is deleted. If the EU 'autonomously' moves, the migration is said to be *proactive*; if the migration is triggered by another EU, the migration is called *reactive*.
- *remote cloning*: a copy of an EU is created at a remote CE but the original copy remains at its current CE.

For weak mobility, a distinction is established between *stand-alone code*, used for creating a new EU on the destination site; and *code fragment*, which needs to be linked in the context of already running code, e.g. Remote Procedure Call (RPC). If the code is executed as soon as it is received then this is called an immediate execution, otherwise it is a deferred execution.

A third form of mobility, not explicitly mentioned in [49] as such, is what can be called *data mobility/migration*. The migration of an EU has implications on its data space (defined

as the set of bindings to resources accessible by the EU). Because not every resource may be moved, e.g. a relatively big database, mobile data must be explicitly declared as such.

As an effort towards abstraction, Fugetta et al. have presented design paradigms meant to serve as architectural abstractions for reasoning about MCSs; these design solutions may allow the design of Distributed Mobile Code Applications (MCAs). They are:

- Code On Demand (COD): an EU running on a host can download and link-on-the-fly (dynamic linking) the code to perform a given task from a different (remote) component that acts as a code server. In short, the client owns the resources but lacks the code;
- Remote Evaluation (REV): here, the clients own the code while the server owns the resources. The server offers a unique service which is the execution of client code; and
- Mobile Agent (MA): a mobile agent is basically an EU, which, while in execution on a given CE, is able to migrate (autonomously) to a different CE where it resumes its execution seamlessly.

Discussion. [49] provides an informal description of mobile systems (that use code migration). There have been distinguished three types of mobility: strong mobility, weak mobility, and data mobility. Three design guides for modelling mobility have also been defined, namely: COD, REV and MA.

Much terminology is introduced, which may cloud understanding. The distinction between immediate and delayed execution, and that between stand-alone code and code fragment are not essential to the understanding of mobility: they all deal with the *time* when an EU is executed on the remote machine.

Also, the distinction between weak and strong mobility leaves room to ambiguity. Indeed, it is possible to suspend the execution of a EU and yet send only its code. Then, one may argue as in [15] that strong mobility may be translated into weak mobility.

One way of getting around the preceding ambiguity would be to classify mobility according to the type of entity concerned. Then, EU mobility must be distinguished from code mobility. The 'code' may be understood as a piece of text, with the particularity that it is executable, and code mobility is hence a form of data mobility. EU mobility implies code mobility, and depends on the migration mechanism, which notably suspends the EU and performs both code and state mobility.
EU mobility may further be partitioned according to (the precision of) the value of the execution state. We have the following cases:

- weak mobility: the execution state is discarded, and the initial state is sent instead. This is the weakest form of mobility.
- strong mobility: the last execution state is sent. Here, the last execution state may be more or less precise, depending on the interrupt (or suspension) mechanism involved. For example, most video games have checkpoints, so migration will restore the game

according to the last checkpoint, and not the last execution state proper. As another example, most modern text editors use both a manual and an automatic checkpoint systems. Manual checkpoints are realised by the user, and are less regular than automatic checkpoints which are hence more precise.

A second, simpler, classification may be achieved by looking at the mechanisms involved only. Then, there is weak mobility when there is no suspension, and only code plus eventually the initialisation data may be sent: in this case, weak mobility and code mobility are the same. And, there is strong mobility when suspension is involved, and the last execution state is sent (without care for precision — the precision will depend on the interrupt or suspension mechanism). **It is this second classification that we adopt in this thesis.**

In [49], EUs are sequential programs only. The particular execution order of the instructions of a program may not affect the mobility of the program, hence such a restriction on EUs is not essential. **In this thesis, EUs will denote any program independently of their architecture.**

Another form of mobility is defined in [49], namely *object mobility*, according to the Object Oriented (OO) paradigm. This thesis is not concerned with that form of mobility, or not directly. Seeing that the OO paradigm is a model of computation, the distinction seems unnecessary. Rather, one should talk, more generally, of mobility in the OO context. The work in [131] discusses the semantics of strong mobility in such a context.

[49] does not consider the mobility of channels. The notion of a location for a channel may not be easily defined therein. Since CEs are the only connected elements, it is possible to discuss channel mobility as the movement of a channel from one CE to another. However this is quite restrictive since programs running on the same machine may be able to communicate, and hence, pass channels between themselves. **In this thesis, we will allow EUs to communicate and hence of changing their communications channels.** This seems to introduce a hierarchy of locations, a potential complication. We return to this in greater detail in Chapter 4.

In summary, the framework defined by Fugetta et al. [49] provides us a good starting point for discussing mobile systems. Whilst it has some limitations, for e.g. (i) it introduces too many concepts, and (ii) the computation model is restrictive, it has proved to be of great help in this thesis, notably when reading about formal semantics for mobility in the literature. Concretely and as will be seen later, it has permitted us to clarify the existing UTP model defined in [125, 126] by pointing out at hidden design decisions. We may also more formally distinguish between types of process mobility in process algebras in general, a distinction that is often neglected. Finally, it has also reinforced our initial intuition that generally speaking, processes might be seen as the locations for mobile entities.

3.1.2 UTP-CSP + weak mobility

Process mobility is concerned with the passing of processes along channels, from one *location* to another. After introducing within the UTP theory of CSP processes higher order process variables, process mobility was then achieved by sending and losing the values of these HO variables from the source CE, and subsequently receiving them in the target CE. In [125, 126], the distinction between weak and strong mobility is not made explicit. However, the semantics provided therein correspond with weak mobility.

Section 3.1.2 contains the design of elements of a mobile system, using as a basis modified Fugetta et al. 's framework, presented in the previous section. (We recall our slight modification: EUs represent programs in general, and not sequential programs only.) Such a discussion is absent from the original work [125, 126], and clarifies the results therein, which are presented in §3.1.2.

Concepts and formalisation

As already mentioned, weak mobility is about moving the code of a EU (possibly with some initialisation data) from one CE (the source) to another (the target). We will first discuss the representation of elements of a mobile system, namely the concepts of *code*, EU and CE.

Code, EU, CE. An EU is a processing unit, or program, hence it may be represented by a UTP-CSP process. A CE is a place where computations occur, i.e. also the place where the execution of EUs occur. There is hence a *containment relation* between EUs and CEs, namely: *a CE may contain many EUs; an EU belongs to only one CE* (this does not omit distributed programs). In order to gain a better understanding of what a CE is, we may make a parallel with Operating Systems.

We may say that *an Application Program belongs to an Operating System* if we can actually observe its execution on the Operating System. However, an Application Program may not be run on an Operating System if its *code* is not 'stored' in the Operating System (more precisely, in the Operating System's memory). Application programs running on an Operating System correspond with EUs executing on a CE; that an Operating System is also a program suggests that a CE may be represented by a UTP-CSP process.

The notion of *location* is absent from the basic UTP-CSP model, but not the notion of *environment*. We see that a CE and the UTP concept of environment are similar, except that the first denotes also a location, whilst the second does not, not traditionally. Hence, we will use the terminology only w.r.t. to the containment relation between two processes: both EUs and CEs will be modelled as processes; and a process will be considered as a location for computation (the CE) only with reference to another process (the EU).

The *code* of a program (EU or CE) generally refers to the text describing the computation that the program should perform. In this context however, it is more appropriately referring to a *stored program*, in the sense of an application program stored on an Operating System.

In UTP, stored programs are modelled as the values of higher-order program variables, and are also called *procedures*. The difference between a procedure and a process (or a process expression) is in their evaluation: a procedure denotes a value, whereas a process denotes the execution of such a value.

More formally, the containment relation between process and procedure may be stated thus: *a process contains a procedure if there is a higher-order process variable in the process's alphabet whose value may correspond with the procedure.* Hence, we will say that a CE (process) contains a EU (process) if there exists a higher-order process variable in the alphabet of the CE whose value is (the code of) the EU. That is, a process P will be a CE for a process Q iff

$$\exists\, h \mid (h \textit{ is a higher-order process variable and } h \in \alpha P) \bullet h = \{\mid Q \mid\}$$

where $\{\mid Q \mid\}$ denotes the code of process Q.

Resources. We now discuss an important aspect of the formalisation but easy to be neglected, namely the notion of *(program) resources*, especially data and channels. Fugetta et al. [49] discuss in length the binding of resources to programs and how mobility affects such bindings.[1] In UTP, *initialisation data* is given in the form of an initialisation predicate that should be part of the definition of a process. Hence **a process (EU) may always move with its initialisation data**. The same goes for channels, which are always part of the definition of a process and constitute its interface. Namely, **a process (EU) may always move with the channels that belong to its interface**. However, **the movement of an EU may not affect the interface of its CE process**.

Indeed, consider again the case of Application Programs and an Operating System. Each Application Program may be considered of having 'abstract' channels that are then mapped onto 'concrete' channels provided by the Operating System environment. Whenever an Application Program moves hence, it moves with its (abstract) channels, without affecting the (concrete) channels of the Operating System. In formal terms, there is no distinction between abstract and concrete channels: they are both modelled as CSP channels. The same formalisation of the CE-EU relation between processes may be seen to hold between concrete and abstract channels. Indeed, a higher-order process variable does not affect the interface of the process to which it belongs. This latter fact is, at least, the default case, and is a sensible one.

We may ask whether or not we should allow weak mobility (resp. strong mobility) to induce channel mobility. As far as we are concerned, we are not aware of any actual application for this feature, although we may perceive a possible utility for autonomous systems. This allows us to state that such systems are not our concern in this thesis, hence, **we shall not reason about processes that may move themselves**.

Notwithstanding, the previous question is sensible because in the vast majority of cases,

[1] This discussion was omitted in Section 3.1.1 above, for conciseness.

programs are constructed to operate in a given environment. This may even cover every possible case (including autonomous processes), since an EU is considered as such only with regard to a given environment. Therefore, even if one were to imagine a process Q leaving the environment $CE1$ for the environment $CE2$ in order to extend the channels of $CE2$, then it would be necessary that Q has in its interface channels not already in the interface of $CE2$. It will therefore be impossible for $CE2$ to execute Q so that Q may in turn, as part of its behaviour, increase the interface of $CE2$. If instead, the reception of Q implies also the increase of the interface of $CE2$, then we can separate weak mobility from channel mobility since we may send new channels independently from sending the code of the process that may use them. Wherefore, **there is no need for weak mobility to induce channel mobility**. This reasoning applies equally in the case of strong mobility.

Semantics

As discussed above, giving semantics to weak mobility requires first to introduce High Order Programming into UTP-CSP, which uses mainly first-order data variables. Hoare and He [66, ch. 9] have defined a theory of High Order Programming on the basis of the theory of Designs. Tang and Woodcock [126] have done the same but for UTP-CSP. The presentation below follows from [126].

First, the alphabet of UTP-CSP processes is extended with higher-order procedure variables. Procedure values have the particularity that they may be executed. Let $< h >$ denote the activation of (the procedure value held by) the higher-order variable h. For now, the semantics of $< h >$ may be understood informally, namely, it behaves like the process whose expression corresponds with the value of h.

First-order data may be compared simply by using equality ($=$), whereas higher-order procedure values are subject to implication ordering ($\Rightarrow$). Hence, the introduction of higher-order variables has an impact on the refinement ordering.

Definition 3.1.1 (Variable refinement [126, Def. 2])**.** *Let p and q be two program variables of the same type.*

$$p \sqsubseteq q \;\widehat{=}\; \begin{cases} p = q & \textit{if } p, q \textit{ are data variables} \\ [< q > \Rightarrow < p >] & \textit{if } p, q \textit{ are process variables} \end{cases}$$

□

Procedure values (or procedures) and process expressions (or processes) may also be compared.

Definition 3.1.2 (Procedure/process refinement [126, Def. 3])**.** *Let $\alpha h = \alpha Q, h, h' \notin \alpha Q$. Then*

$$h \sqsupseteq Q \;\widehat{=}\; [< h > \Rightarrow Q] \qquad \square$$

The extension of the alphabet with higher-order process variables does not affect the (healthy) behaviour of processes, hence it preserves the existing UTP-CSP healthiness conditions.

The definition of assignment must be modified to take into account the refinement ordering between higher-order process variables.

Definition 3.1.3 (Higher-order procedure assignment [126, Def. 3])**.** *Let* $\alpha h = \alpha P, h, h' \notin \alpha P$.

$$h := \{\mid P \mid\} \;\widehat{=}\; \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge h' \sqsupseteq P \wedge v' \sqsupseteq v)$$

where v *denotes all program variables except* h. $\square$

The activation of a higher-order process variable is more easily expressed by means of an algebraic law.

Theorem 3.1.4 (Procedure activation 1)**.** *Let* $\alpha h = \alpha P, h, h' \notin \alpha P$. *Then*

$$(h := \{\mid P \mid\} ;; < h >) = (h := \{\mid P \mid\} ;; P)$$

$< h >$ *denotes the activation of (the procedure contained in)* h.

Proof. cf. [126, Law 3]. $\square$

Conversely, any process stands for the execution of the higher-order process variable whose value corresponds with the process's expression.

Theorem 3.1.5 (procedure activation 2)**.** *Let* $\alpha h = \alpha P, h, h' \notin \alpha P$. *Then*

$$P = (\mathbf{proc}\, h := \{\mid P \mid\} ;; < h > ;; \mathbf{end}\, h)$$

Proof. cf. [126, Law 5]. $\square$

The definition of communication must also take into account the communication of procedure values viz. the refinement ordering of said values. The communication of procedure values is denoted by $c.M$, instead of $c.m$ for first order values.

Definition 3.1.6 (Higher-order communication [126, Def. 12])**.**

$$ch.E \mathrel{\widehat{=}} \Box\{do(ch.M) \mid M \sqsupseteq E\} \qquad \square$$

Hereafter are presented the semantics for weak mobility. One way of describing the effect of weak mobility is so called 'copy-then-delete' semantics, meaning that a duplicate version of the code is sent to the target, whilst the original version is deleted from the source; another way is by separating its effect according to the respective viewpoints of the source and target.

From the target, the effect of weak mobility is simply the update of some higher-order variable with the procedure value that was received. There is no difference with (first-order) input prefix. From the source, the higher-order variable used for the communication loses its value at the end. That is, the procedure value is first output, and then the higher-order variable loses that value.

Clone assignment is the process that copies the value of a higher-order variable q into a distinct higher-order variable p viz. the update of p with the value of q.

Definition 3.1.7 (Variable copy (or clone) assignment [126, Def. 11])**.**

$$\begin{aligned} p := q &\mathrel{\widehat{=}} \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge p' \sqsupseteq q \wedge v' \sqsupseteq v) \\ \alpha(p := q) &\mathrel{\widehat{=}} \{\mathbf{o}, \mathbf{o}', p, p', v, v'\} \end{aligned}$$

where v is the set of program variables except p. □

Mobile assignment is the process that copies the value of a higher-order variable, q, into a distinct higher-order variable p, and then loses the value of q.

Definition 3.1.8 (Variable copy-then-delete (or mobile) assignment [126, Def. 10])**.**

$$\begin{aligned} p :=_m q &\mathrel{\widehat{=}} \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge tr' = tr \wedge p' \sqsupseteq q \wedge q' \sqsupseteq CHAOS \wedge v' \sqsupseteq v) \\ \alpha(p := q) &\mathrel{\widehat{=}} \{\mathbf{o}, \mathbf{o}', p, p', q, q', v, v'\} \end{aligned}$$

where v is the set of program variables except p, q. □

The mention of the clause $q' \sqsupseteq CHAOS$ above could have been omitted since it equates to *true*. The clause $q' \sqsupseteq CHAOS$ means that q' has no useful value (cf. Def. 3.1.2).

The effect of moving out a (procedure) value, namely, sending the value to the target process and then deleting it from the (higher-order) variable that contained it, is denoted by $c!!q$.

Definition 3.1.9 (Mobile output prefix [126, Def. 15])**.**

$$c!!q \to P \mathrel{\widehat{=}} \bigsqcap \{(do(c.M) \vee (\neg\, wait' \wedge q' \sqsupseteq CHAOS)) \mid M \sqsupseteq q\} \fatsemi P$$ □

A consequence of mobile assignment, and of mobile output prefix also, is that any subsequent attempt of activating the procedure should fail, unless the variable is set anew.

Theorem 3.1.10 (Undefined activation)**.**

$$\begin{aligned} (g :=_m h \fatsemi <h>) &= CHAOS \\ (c!!h \to <h>) &= (c.h \fatsemi CHAOS) \end{aligned}$$

Proof. cf. [126, Law 4]. □

The following law states the equivalence between mobile assignment and mobile communication.

Theorem 3.1.11 (Assignment-communication equivalence)**.**

$$(p :=_m q) = \big((ch?p \to SKIP) \parallel (ch!!q \to SKIP)\big) \setminus ch$$

Proof. cf. [126, Law 11]. □

Discussion: Local vs distributed mobile assignment. Considering, like Fugetta et al. [49], that mobility is inherently distributed the previous theorem implies that from the outside, it is not possible to distinguish between 'local' and 'distributed' copy-then-delete assignment semantics. This raises the question of the specification of weak mobility: *can it be verified that a construct implements 'strict' weak mobility*? The answer is trivial when there is no hiding of corresponding communications: the presence of events of the form $c!!e$ in the trace can tell. However, when there is hiding, the trace model (viz. trace-based specification) seems to be unsuitable for the task. This is not a weakness in itself because hiding has the expected effect.

In summary, we have presented the semantics for weak mobility as defined in [126]. We have added a paragraph on the formalisation of concepts, absent from the original work. In particular, should one insist that weak mobility is intrinsically distributed, then this framework, in the presence of the hiding operator, is unsatisfactory for distinguishing distributed weak mobility from local copy-then-delete semantics.

3.2 Mobile Channels

3.2.1 FOCUS + channel mobility

FOCUS [27] is a semantic framework for the formal specification of programs based on *input/output* (or I/O) relations. A sequential program may be characterised by a function between its initial state or input and its final state or output. Inputs are received through some input channels, and outputs are produced through some output channels. Nondeterministic programs may produce different outputs from the same input; hence, a program may be characterised instead as a relation between inputs and outputs.

The sequence of messages of a given channel is called a *stream*. A FOCUS *component* or simply a *component* C characterises a relation between input streams and output streams. Components may interact with each other to constitute larger components, appropriately called data flow networks. Hence it is equivalent to say that FOCUS allows modelling systems as *data flow networks*.

FOCUS models may be either timed or untimed. Time is modelled by introducing a global clock that splits the time axis into equidistant time units. All the messages input during the same time unit are then stamped with it. The communication model is *asynchronous*.

FOCUS components are normally static (i.e. their channels do not change during their activation). A static component C is defined by a function $f \in (I \subseteq H) \to (O \subseteq H)$ where H is the set of channel histories; $\theta \in (I \subseteq H)$ is the inputs histories, and $f(\theta) \in (O \subseteq H)$ the outputs histories. A *history* is a sequence of messages sent over a channel in a given period.

In [55, 56, 57, 58], Grosu and Stolen extend FOCUS with channel mobility, based on a *timed* model. [58] synthesises the first three; three models are provided depending on the communication paradigm considered:

- Asynchronous *many-to-many* (m2m) communication: here many components may simultaneously use the same channel.
- Asynchronous *point-to-point* (p2p) *with sharing*: here different components may have the same channel in their interface; but only two components may communicate over the same channel at a time.
- Asynchronous *point-to-point* (p2p) *without sharing*: here different components have disjoint interfaces.

In all variants, the underlying channel-passing mechanism is the same: the interface of a component is allowed to change over time, by communication of channels as messages. Each variant simply restricts what components may be involved in a communication at what time. Since our main interest is the presentation of the channel mobility mechanism, we shall not refer to any of the models in particular.

Components are modelled as *guarded functions*. A function is called *weakly guarded* if at any time, its outputs are independent of its future inputs. A function is *strongly guarded* if it is weakly guarded and its outputs at time t are also independent of its inputs at time t, for given t. Intuitively, strongly guarded functions introduce a delay of at least one time unit between input and output; and weakly guarded functions allow a zero-delay behaviour [58].

Definition 3.2.1 (Guarded function)**.** *A function* $f \in H \to H$ *is* weakly guarded *if*

$$\forall\, \theta \in H, t \in Nat_+ : f(\theta) \downarrow_t = f(\theta \downarrow_t)$$

and strongly guarded *if*

$$\forall\, \theta \in H, t \in Nat_+ : f(\theta) \downarrow_{t+1} = f(\theta \downarrow_t)$$ □

N.B. The previous formulation is different from [58, Def. 1].

A static component may be defined by a set (or singleton) containing functions $f \in I \subseteq H \to O \subseteq H$ where both I and O are known and fixed in advance, and may never change. By a naive analogy, a mobile component would be defined by functions of the form $g : i \to o$ where both i and o are set-valued variables. However, such a representation

is not functional-like as it introduces an explicit manipulation of the range and domain of a function. To solve this issue, consider functions defined over H i.e. $f \in H \to H$.

The definition of static functions may be seen as the application of restrictions on both $dom\, f$ and $ran\, f$, respectively the domain and the range of any function f. Using simple arithmetic, we have $I \subseteq H \Leftrightarrow I = H \cap I$ and $O \subseteq H \Leftrightarrow O = H \cap O$. Any static function $f_{I,O} \in H \cap I \to H \cap O$ may hence be viewed as the projection of $f \in H \to H$ in which $dom\, f$ is restricted to I, $ran\, f$ to O, and everything else is undefined. Let $\alpha \in H$, then, the restriction of α to a subset B of H is denoted by $\alpha \mid_B$ [56]:

$$\alpha \mid_B \mathrel{\widehat{=}} \begin{cases} \alpha & \text{if } \alpha \in B \\ \langle\rangle & \text{otherwise} \end{cases}$$

In the case of channel mobility, I and O may change over time, unlike in the static case, hence they will respectively be denoted by $i(t)$ and $o(t)$ (or simply i and o) instead. Because even static functions are defined in terms of the above restriction (or projection) operation, it is necessary to ensure that f behaves like $f_{I,O}$ i.e. although f is defined over H, it uses only those channels specified in I and O. When such is the case, f is called *privacy preserving* [56, §4], [58, Defs. 3, 11].

Definition 3.2.2 (Privacy preservation)**.** *Let $f \in H \to H \mid \theta \mapsto \delta$. Let $dom\, f = I$ and $ran\, f = O$, initially. Then f is said to be privacy preserving if*

$$\begin{aligned} f(\theta) &= f(\theta \mid_i) \\ \delta \in f(\theta) &\Rightarrow \delta = \delta \mid_o \\ \delta \in f(\theta) &\Rightarrow \delta = \delta \upharpoonright (i \cup o) \end{aligned}$$

where i and o are defined recursively as follows:

$$\begin{aligned} i(t_0) &= I & o(t_0) &= O \\ i(t+1) &= \theta \mid_{i(t)} & o(t+1) &= \theta \mid_{o(t)} \\ i &= \bigcup\nolimits_t i(t) & o &= \bigcup\nolimits_t o(t) \end{aligned}$$

where t_0 stands for the origin of time, i.e. $t = 0$. □

A component is static if its interface remains unchanged after t_0, and is mobile otherwise [58, Defs. 4, 12].

Definition 3.2.3 (Static vs. mobile component)**.** *A component C is called* static *if all of its defining functions are static. A function f is called* static *if, after initialisation, its domain and range do not change i.e.*

$$\forall\, t \bullet \theta \mid_{i(t)} = \theta \mid_{i(t+1)} = I \ \wedge\ \delta \mid_{o(t)} = \delta \mid_{(t+1)} = O$$

A component M is called mobile *if at least one of its defining function is mobile. A function*

is called mobile *if, after initialisation, its domain and range may change i.e.*

$$\exists\, t \bullet \theta\,|_{i(t)} \neq \theta\,|_{i(t+1)} \;\vee\; \delta\,|_{o(t)} \neq \delta\,|_{o(t+1)} \qquad \square$$

Discussion. The characterisation (of static and mobile components) that we have given above is different (in its formulation) from what may be found in [58] and related. The difference is that static and mobile components are characterised by their respective *privacy* preservation law, whereas above, it is clearer that not the privacy preservation law itself, but the change of interface occurring over time, is what makes the difference. Moreover,

> *Privacy preservation is intimately related to the notion of time. For each port p received (passive port p sent) for the first time in time unit t , the function f may communicate via p (respectively via $\tilde{p}$) from time unit $t+1$ onwards. Note that such a causality relation cannot be expressed in an untimed input/output model.* ([58, Def. 11. (Privacy preserving function)])

It would seem that the previous statement is not quite correct. To see this, let us first look at the privacy law itself. The value of $i(t)$ is given recursively as a function of $i(t-1)$; it changes only when a channel is communicated as a message. In other words, the effect of channel mobility occurring at time t may only be visible at time $t+1$.

For example, if an existing channel is output at time t, it is only at time $t+1$ that the channel will no longer be available. Hence, it is as if every channel mobility operation pushed the clock forward by one tick. We then end up with two periods: the time before the channel is moved, say t_{bef}, and the time after, say t_{aft}. Each period corresponds to a given interface, respectively, the interface before, and the interface after the mobility. Hence, $i(t)$ clearly gives the interface at t_{bef}, and $i(t+1)$ the interface at t_{aft}, when mobility has occurred; otherwise $i(t) = i(t+1)$ and no mobility has occurred.

The point is this: given that each input/output is stamped with the time of its occurrence, and given that such a time stamp is recorded in the history as a tick event; if we record the interface at the time of the tick event instead of the tick event itself, we may equally characterise channel mobility without using time. In fact, the value of $i = \bigcup_t i(t)$ given above may only be computed if each $i(t)$ has been recorded somewhere, eventually in the history itself (together with the associated tick event). Unfortunately, because such a recording is but *implicit* in the definitions, Grosu & Stolen could not see that *time is not necessary for defining privacy preservation.*

More results. Notwithstanding the above remarks, some more results may be worth mentioning. In [58], mobile components have an additional set of said *passive channels*, denoted by pM. By opposition, the set aM of active channels contains both sets I and O already mentioned, i.e. $aM = I \cup O$. At any time, both sets must be disjoint. The channels in pM are private in the sense that they are known of the component only, and not of the environment; hence pM always contains the two ends of a channel. Both active and passive channels may

appear as messages of output communications; but only active ones may be used for communications. When a passive channel is output, one of its ends must appear in aM of the current component, thus making it active, whilst the other will be in the corresponding set for the receiving component. In consequence, in the *p2p* model, a channel may become either active or passive, according to whether a component owns both ends of the channel or not. Such may not be the case in the $m2m$ model because many components may share a channel at the same time once it has been made public. The introduction of passive channels has an effect on the dynamics of valid channels, and is captured by the privacy preservation law as follows: at any time, both input and output channels may belong to aM only, and none to pM. We refer the reader to [58] for further details.

Another interesting result is that hiding is defined in the m2m model but not in the other models, and over the *initial* set of active channels only. As a consequence, it is not possible to increase silent channels dynamically.

As a final remark, because communication is asynchronous, it may not be possible to model channel faults; also, it is not possible of modelling refusals and divergences in the context of data flow networks ([66, chap. 8, §8.3, p. 231]).

Other, related works. In [123], Stolen uses a different approach to the problem of extending FOCUS with mobility than the one in [58], presented above. In [123], acquired channels are recorded, and a healthiness condition imposes that only such channels may be used effectively. Unlike [58], which is not concerned with step-wise refinement and formal verification, the work in [123] explicitly addresses those concerns.

More recently, in [29], Broy discusses semantics of channel mobility in FOCUS based on the work in [58]. Broy notably discusses issues such as causality and concepts of object oriented programming.

3.2.2 A CSP model for occam-pi

In [138], Welch & Barnes have proposed a CSP semantics for the channel mobility mechanism implemented in the occam-pi programming language [137, 11]. We present this work subsequently.

occam-pi mobile channels. Occam-pi is a programming language that supports both the mobility of processes and of channels, by means of their communications over channels. In that, it extends the occam programming language [12]. Two types of communication are possible:

- static point-to-point synchronous communication: channels are non-mobile so (their reference or name) cannot be communicated. All communications are point-to point and synchronous.
- dynamic multiplexed synchronous communication: it is possible of sharing channels thus enabling 1-to-many (or many-to-1, or many-to-many) channel configuration.

The entities present in the language are data variables, channels and processes, all of which can be either mobile or not. So, communication bears two semantics: a *copying semantics* (the value communicated is still available at the source) and a *movement semantics* (the value communicated is only available at the destination). Entities that may be moved (i.e. are mobile) must be declared explicitly as such. Hence, it is possible to define *mobile data types*. Variables of such a type lose their value after their assignment to other variables (*copy-then-delete*, or movement, semantics).

There is a distinction between channel ends and channels. An occam-pi process may use a channel only if the latter is declared in the process's definition (similar to variables). Each end of a channel, indicating the direction of the communication, must be declared explicitly.

In order to provide channel mobility, occam-pi introduces the notion of a *bundle* of channels viz. informally, a group of channels that may *all* participate to the communication between two given processes, exclusively. It is implemented as a *record*, whose fields are exclusively channels. A bundle is hence a type, a mobile data type; it is also any entity of said bundle type. Like channels, bundles also have ends. A *bundle end* indicates the direction of use of the channels recorded in its definition: the *server-end*, also called *negative end*, reverses the direction of use of declared channel fields; the *client end*, or *positive end*, conserves their direction. *Channel mobility is thus achieved by the communication or assignment of values of bundle type.*

occam-pi mobile channels in CSP. An occam-pi program, say $O\pi$, consists of two CSP processes running in parallel:

- the application (system) process AS: contains all the processes of the application, which may possess their own static channels;
- the mobile channel kernel process MCK: responsible for the creation and management of bundles.

Both bundles and (occam-pi) mobile channels are modelled as *indexed* CSP processes. They will be referred to (respectively) as bundle process, denoted by $Bdle(bId, nbFlds)$, and as channel-field (component) process, denoted by $ChFld(bId, chId)$. When there is no ambiguity, we shall simply talk of bundle and of channel-field. bId is the bundle unique index (or identifier), $nbFlds$ is the number of channel fields in the bundle; $chId$ is the channel-field unique index within the bundle, whereas $bId.chId$ does uniquely identify a channel within the whole application.

Every channel-field process owns exactly three channels in the set $\{wr, ack, rd\}$, indexed accordingly with $bId.chId$. These are traditional CSP channels, or more precisely, they are channel ends, since they are always used in a single direction, respectively: for a given CSP channel ch, $ch! = wr \wedge ch? = rd$, and ack is simply for synchronisation and carries otherwise no data. A further signal, denoted by $kill$ allows deactivating an existing channel-field; as a

consequence, corresponding channels may no longer be used [138, $Chan(c, i)$].

$$\begin{aligned} ChFld &\mathrel{\widehat{=}} \begin{pmatrix} bId.chId\_wr?x \rightarrow bId.chId\_rd?x \rightarrow bId.chId\_ack?x \rightarrow \\ (ChFld \,\Box\, kill \rightarrow SKIP) \end{pmatrix} \\ chansOf(ChFld) &\mathrel{\widehat{=}} \{bId.chId\_chan \mid chan \in \{wr, ack, rd\}\} \end{aligned}$$

Above, $ChFld$ stands for $ChFld(bId, chId)$. For ease, mention of the parameters will be omitted for other processes also.

A bundle process has four (4) components running in parallel. The easier one is composed of all the channel-field processes in parallel, and will be referred to as the list of fields (process), denoted by $ChList(bId, nbFlds)$. Indeed, this process has only a grouping purpose, and is defined by [138, $Channels(c, fields)$]:

$$\begin{aligned} ChList &\mathrel{\widehat{=}} \Big\|_{1 \leq chId \leq nbFlds} ChFld(bId, chId) \\ chansOf(ChList) &\mathrel{\widehat{=}} \bigcup_{1 \leq chId \leq nbFlds} chansOf(ChFld(bId, chId)) \end{aligned}$$

The three other processes have a control role, and ensure a healthy use of a bundle. The process $Refs(bId, count)$ counts the number of processes currently holding either end of a bundle [138, $Refs(c, n)$].

$$\begin{aligned} Refs &\mathrel{\widehat{=}} \begin{cases} kill \rightarrow SKIP & \text{if } count = 0 \\ \begin{pmatrix} (bId\_enrol \rightarrow Refs(bId, count + 1)) \,\Box \\ \quad bId\_resign \rightarrow Refs(bId, count - 1) \end{pmatrix} & \text{otherwise} \end{cases} \\ actOf(Refs) &\mathrel{\widehat{=}} \{kill\} \cup \{bId\_act \mid act \in \{enrol, resign\}\} \end{aligned}$$

The processes $\{Mutex(bId, bEnd) \mid bEnd \in \{pos, neg\}\}$ ensure that for a given shared bundle end, only one communication occurs at a time [138, $Mutex(c, x)$].

$$\begin{aligned} Mutex(bId, bEnd) &\mathrel{\widehat{=}} \begin{pmatrix} (bId.bEnd\_claim \rightarrow bId.bEnd\_release \rightarrow Mutex(bId, bEnd)) \,\Box \\ kill \rightarrow SKIP \end{pmatrix} \\ actOf(Mutex) &\mathrel{\widehat{=}} \{kill\} \cup \{bId.bEnd\_act \mid act \in \{claim, release\}\} \end{aligned}$$

A bundle process $Bdle(bId, nbFlds)$ is itself defined by [138, $Bundle(c, fields)$]:

$$\begin{aligned} Bdle &\mathrel{\widehat{=}} Refs(bId, 2) \parallel Mutex(bId, pos) \parallel Mutex(bId, neg) \parallel ChList \\ actOf(Bdle) &\mathrel{\widehat{=}} actOf(Refs) \cup actOf(Mutex) \\ chansOf(Bdle) &\mathrel{\widehat{=}} chansOf(ChList) \end{aligned}$$

There is a special bundle process, denoted by $UndefBdle$, whose index is 0 and that is used

as a reference for *undefined* bundles [138, *UndefinedBundle*].

$$\begin{aligned} UndefBdle &\mathrel{\widehat{=}} 0\_resign \rightarrow UndefBdle \,\Box\, nomorebdles \rightarrow SKIP \\ actOf(Bdle) &\mathrel{\widehat{=}} \{0\_resign, nomorebdles\} \end{aligned}$$

As stated above, the role of *MCK* process is the creation and management of bundles. The process that creates new bundles on request is denoted by $NewBdle(bId)$, defined by [138, $CMC(c)$]:

$$NewBdle(bId) \mathrel{\widehat{=}} \left(\begin{array}{l} set?nbFlds \rightarrow get!bId \rightarrow \\ ((Bdle(bId, nbFlds) \,|||\, NewBdle(bId+1)) \,\Box\, nomorebdles \rightarrow SKIP) \end{array} \right)$$

MCK is the process that behaves like *UndefBdle* when a process makes a reference to an undefined bundle, or else runs one or more bundle processes in parallel [138, *MOBILE_CHANNEL_KERNEL*].

$$MCK \mathrel{\widehat{=}} NewBundle(1) \parallel UndefBdle$$

An occam-pi program $O\pi$ may thus be defined by the following CSP process [138, *APPLICATION_SYSTEM*]:

$$\begin{aligned} O\pi &\mathrel{\widehat{=}} ((AS \fatsemi nomorebdles \rightarrow SKIP) \parallel MCK) \setminus kernel\_chans \\ kernel\_chans &\mathrel{\widehat{=}} \{enrol, resign, claim, release, wr, rd, ack, set, get, nomorebdles\} \end{aligned}$$

Note that the set *kernel_chans* as given above is not defined fully (i.e. not every channel has been defined), and is meant to represent all those communications and interactions with the process *MCK*. It is given as such only for readability, since its actual value is easily computable from the definitions of processes.

Each *AS* process may communicate with another through static channels, as any traditional CSP process may do, or else, may communicate through bundles provided by the process *MCK*. By calling the process *NewBdle*, an *AS* process may request the creation of a new bundle. *Channel mobility is achieved by communicating bundle indexes from one AS process to another.* Unfortunately in [138] alphabets are not explicitly discussed, hence it is *not* clear what entitles a process the use of a channel. Said differently, it is not clear what the valid traces of a process are.

Discussion. We agree with [18, 132] in saying that the model above is implementation-oriented, and further precise in what sense. Indeed, the concept of a *channel* as defined in process algebras in general, and in particular in CSP is quite abstract. It may refer to the simple cable linking two computers in a LAN, or to the more complex collection of cables and devices that link two distant computers on the Internet. Precisely, modelling occam-pi mobile channels as CSP processes is thus a choice of implementation. Furthermore, because

of such a choice, by channel mobility, one could be misled at first to thinking that said mobile channel process and hence process, would be moving. Such is not the case, hence, one may wonder if the above model actually describes channel mobility at all. Another question that comes to mind, seeing that the model is not abstract enough (it is implementation-oriented), is whether or not it can actually be simplified. The answer to this second question does actually provide an answer to the first. We discuss both questions subsequently.

i. As a first move towards simplification, consider occam-pi bundles that have only one occam-pi channel. Then, corresponding CSP bundles will have only one CSP channel-field process. As before, each channel-field process will have three channels. Since we are left with a single channel-field, we may drop the *chId* indexes.

ii. Remark that the processes *Refs* and *Mutex* implement a point-to-point communication protocol with sharing, but at the level of bundles. CSP already assumes the same protocol but for CSP channels, so we may question the possibility of eliminating the redundancy. To answer this question, see that accessing a bundle is but a means for *accessing its channel-fields*. In turn, accessing a channel-field process is but a means for *accessing its channels* viz. $\{wr, ack, rd\}$. In CSP, a process may access a channel if the latter is in its interface, as given by the $chansOf()$ function used previously. Hence, we may eliminate the processes *Refs* and *Mutex*; we may also eliminate the channel-field process *ChFld* but keep only its channels $\{wr, ack, rd\}$, which will be identified with *bId* only. (We may easily generalise again to having many channel-field processes; however, we would need to keep *chId* in order to distinguish them.)

iii. Similarly, since *MCK* is but a collection of bundles whose main purpose is to grant access to the channels which they hold, we may eliminate bundle processes and keep only *bId*, for grouping purposes. (A further simplification is possible, by considering a single bundle only. Then we would also drop *bId*. This means that we would have reduced everything to a single channel. However, we would end up with a very simple model. The discussions below are more general, since we deal with both many bundles, and for each bundle, possibly more than one field.)

This first simplification step (i-ii-iii) leaves us with only two processes, *AS*, and *MCK*, in which all the previous components processes have been eliminated, and only their respective sets of channels, indexed accordingly, have remained. As a result, we have: $chansOf(AS) = chansOf(MCK) = \{bId.chId\_chan \mid chan \in \{wr, ack, rd\}\}$. This leads us to our second simplification step, namely, eliminating *MCK*, or perhaps more precisely, abstracting away from it.

Indeed, *MCK* plays the role of a server of all the mobile channels of the application. Since every application process in *AS* may be linked to *MCK*, the latter can be modelled as a set instead: the set of all the mobile channels of the application.
Now, looking at *AS* processes, we see that in most cases, one such process may have only a subset of all the possible indexes. That subset can change according to the movement of

the indexes. When a process releases or moves out an index, it can no longer communicate through the corresponding (generic) channels. And when a process receives an index, it can thereafter use the corresponding (generic) channels. However, not everything seems right. The way the set of indexes is defined, the interface of processes is still static. Indeed, $indexesOf(P) = \{(bId, nbFlds)\}$ uniquely defines what channels may be used. Hence, even if a process P receives an index, say $newId \notin indexesOf(P)$, P may never actually use the channels corresponding to *newId*. This means that the actual effect of moving indexes is activating/deactivating existing channels. If we view $indexesOf(P)$ as a (strict) subset of *MCK* (understood as the universal set of all the possible indexes), then P will never receive indexes outside $indexesOf(P)$, which seems like a severe restriction to the model.

Instead, suppose that we equate $indexesOf(P)$ with *MCK*, for every P in *AS*. This would mean that by default, every process is connected to one another. Again, moving indexes around will have the same activate/deactivate effect, suggesting that initially, it is not necessary that every process has all of its mobile channels active. Hence, we may introduce the following subtlety in the definition of *MCK*:

- *MCK* may represent *movable* (inactive) channels viz. that a process may acquire or move in, at any time, but may never release or move out; whilst
- $indexesOf(P)$ may represent *usable* (active) channels viz. that a process has already acquired and hence may use for its communications, and may also release or move out.

Considering the latter definitions, we may actually discard indexes altogether and move channels directly, as if they were objects themselves, if we remark that in the end, indexed channel names are but channel names. At the same time, this notion of indexing paves the way for linking, in a mathematical sense, both theories of static networks and of mobile networks:

> A mobile network may be *simulated* by a static network that connects all the nodes of the mobile network to one another, and then by indexing channels so that every channel is mapped to a set of indexes, and each index defines all of the channels that were active during the same period. The ordering of the indexes would hence define the consecutive changes of network topology as they have occurred in the mobile network.

In conclusion, we have presented Welch & Barnes [138] model for capturing the channel mobility mechanism of occam-pi using CSP. The result is not abstract enough, and we have said why. We have proceeded further to make their model more abstract. Our analysis has shown that the CSP processes used for modelling occam-pi channels and bundles had merely a structuring role, and that such a structuring could be lifted into some channel naming procedure. Those modelling processes (for bundles and channel-fields i.e. *Bdle*, *ChFld*, *MCK* and related) were hence discarded, leaving us with a more abstract model than the original one. The model in [138] was particularly useful in that it strengthened one of our main

intuition about mobility, namely, the distinction between the *capability* of a process, and its interface (cf. Chapter 4).

3.2.3 CSP||B + channel mobility

CSP‖B [116] is a formal language that aims at combining CSP processes with B machines [1]. A CSP‖B *controlled component* consists of a sequential CSP controller (process), say P, in parallel with a B machine, say M. A B machine is itself modelled as a CSP process, where a B machine operation op with given input s and output t, declared in a machine M as $t \leftarrow op(s)$, is modelled as a CSP event $op.s.t$.

Each machine instance in a CSP‖B system owns a unique *machine reference*, denoted by z. Operation calls correspond to the communication $z.op.s.t$, and the machine reference z can itself be communicated between controllers. As a requirement, only one sequential controller may own a reference z at any one time, so that when z is passed from say controller P_1 to controller P_2 then P_1 may no longer use z.

The following elements are part of the semantics of a controller:

- the set MR of machine references: contains the links to interact with B machines;
- the set CP of control points: contains special channels, through which machine references may be communicated;
- the set C of regular CSP channels (disjoint from CP).

The interface of a controller process is hence $\mathcal{I} = CP \cup C \cup MR$. *Channel mobility is achieved by passing references around between processes*, which modifies the value of MR, hence making the interface dynamic. The valid traces are given by recursion as shown below.

Definition 3.2.4 (Valid mobile CSP||B traces [132])**.**

$$\langle\rangle \upharpoonright \mathcal{I}P = \langle\rangle$$

$$(\langle cp.z\rangle \frown tr) \upharpoonright \mathcal{I}P = \begin{cases} \langle cp.z\rangle \frown (tr \upharpoonright \mathcal{I}P \cup \{z\}) & \text{if } cp \in in\mathcal{A}(P) \wedge z \notin MR \\ \langle cp.z\rangle \frown (tr \upharpoonright \mathcal{I}P \setminus \{z\}) & \text{if } cp \in out\mathcal{A}(P) \wedge z \in MR \\ tr \upharpoonright \mathcal{I}P & \text{if } cp \notin \mathcal{A}(P) \wedge z \notin MR \\ \textit{undefined} & \end{cases}$$

$$(\langle c\rangle \frown tr) \upharpoonright \mathcal{I}P = \begin{cases} \langle c\rangle \frown (tr \upharpoonright \mathcal{I}P) & \text{if } c \in in\mathcal{A}(P) \\ tr \upharpoonright \mathcal{I}P & \text{if } c \notin in\mathcal{A}(P) \end{cases}$$

$$(\langle z.op\rangle \frown tr) \upharpoonright \mathcal{I}P = \begin{cases} \langle z.op\rangle \frown (tr \upharpoonright \mathcal{I}P \cup \{z\}) & \text{if } z \in MR \\ tr \upharpoonright \mathcal{I}P & \text{if } z \notin MR \\ \textit{undefined} & \end{cases}$$

□

The traces above are always given through restriction because of the inherent parallelism of CSP||B components. The second and fourth clauses, involving a machine reference z, are especially relevant to channel mobility.

- Second clause: the first case states that (assuming that the machine reference z is 'new' i.e. $z \notin MR$), after being input, the next valid trace must have $\{z\}$ in its interface; the second case states that (assuming that the machine reference z is 'old' i.e. $z \in MR$), after its output, the next valid trace *may not* contain $\{z\}$ in its interface. The third case is about hiding, and the last one states that any other case is undefined.

- Fourth clause: the first case states that communication through a machine reference does not change the interface of the next process; the second case states that if z is invisible, it should be invisible in the next trace also. The last case states that any other case is undefined; for e.g. if you append a trace $\langle z.op \rangle$ with an interface not containing $\{z\}$.

A sort of healthiness condition imposes that at all times, no two controllers may hold the same machine reference.

Definition 3.2.5 (MR disjointness condition [132])**.** *The set MR of machine references of each controller of a given CSP‖B component must always be disjoint.*

Discussion. The channel-passing mechanism of mobile CSP‖B is thus: each component has a set of mobile channels which increases and decreases according to the movement of the channels therein. The fact that a mobile channel may be held by only one controller process at a time restricts the expressiveness of the model, but this restriction may, it seems, be lifted out with little to no difficulty. Another restriction is that mobility occurs between a CSP controller and B machines, but not amongst CSP controllers themselves. Also, traces semantics only are provided.

A number of assumptions are left implicit so one has to exercise care when constructing processes. This remark concerns particularly the law defining mobile communications $cp.z$. It states that following such an operation, the machine references of the next running process must differ from that of the current one. Whilst this is reasonable (with regard to the algebraic laws of the prefix operator), such may not be the case in the presence of *recursion*, and of *sequential composition*.

As a general remark, operators are not discussed in [132], which is unfortunate. It is noteworthy that *sequential composition* is not part of the syntax of CSP processes defined in [132]. Although hiding seems to be implied in the laws for valid traces above, the *hiding operator* is also not in the syntax of CSP processes given in [132]; and the authors have themselves ruled out internal parallelism in CSP controllers, as it is stated in [132, Conclusion].

Notwithstanding its limitations, this work may be seen to improve on the one in [138] described previously (§3.2.2) in two aspects: actual channels are moved, which corroborates our earlier hypothesis that channels may be moved directly instead of through indexes (§3.2.2,

Discussion); the traces are explicitly calculated, which is not the case in [138].

A result provided in [132] but not presented above is a theorem for verifying divergence-freedom, and another for verifying deadlock freedom (of mobile CSP||B components). Since the network of CSP controllers is static, traditional CSP techniques do apply. Hence, only B machines need to be verified. Such a verification is realised on the data exchanged between a CSP controller and a given B machine. Mobility itself does not play any role in the definitions provided, so it would be interesting to know if the techniques are simply those of static systems that have been *reused* in mobile CSP||B. If it turns out that the techniques are not identical, then it could be interesting to study how static and mobility techniques relate to one another. The more general question is in fact that of the relation between mobile CSP||B and static CSP||B, not explored in [132].

3.2.4 CSL + CSP + channel mobility

In [67], Hoare and O'Hearn propose a traces model for channel mobility based on ideas from both Concurrent Separation Logic (or CSL) and CSP. Their initial intuition is the similarity between the notion of *ownership* used in CSL which is dynamic, and the notion of *alphabet* used in CSP which is static. Indeed, the alphabet of a CSP process determines a form of location for the channels that the process may use and hence *owns*. The problem is then to make the ownership model of CSP alphabet dynamic.

The basic elements of the model are those of CSP with some particularities:

- point-to-point communication only is considered, where each channel has only one sender and one receiver at a time, though not always the same at different times;
- channels can be passed as the contents of messages (channel passing) and be dynamically allocated, as in the pi-calculus;
- channels are considered *concrete*, meaning that they are objects, much like integers;
- dynamic allocation and deallocation of channels are possible, whose effects are respectively to increase and decrease the alphabet;
- alphabets do model ownership but may change over time, as a result of channel passing, allocation, and deallocation aforementioned;

The model includes two separating conjunctions:

- parallel composition is modelled by separation in space;
- sequential composition is modelled by separation in time;

and spatial composition of alphabets ensures that only one process can own a channel end at any time.

An alphabet α is a finite set of channel ends $c!$, $c?$, where c is drawn from an infinite collection of channels. An event set E is a finite set of primitive events drawn from $c!m$ and $c?m$, where the messages m themselves have a structure consisting of a value v and a permission ρ [67, §4.2]:

$$m ::= v\rho \qquad v ::= c \mid 3 \mid ... \qquad \rho ::= \epsilon \mid ! \mid ? \mid !?$$

ϵ stands for the empty permission, for messages other than channels; !? means that both ! and ? permissions are sent alongside a channel. A projection function over a message $m = v\rho$, $res(m)$, returns the set of channels corresponding to m; a projection function over an event e, $pre(e)$, returns the set of channels necessary for the event to occur, and is called the *pre-alphabet* of the event. They are defined below [67, §4.2].

$$res(m) = \begin{cases} \{\} & \textit{if } m = v\epsilon \\ \{v!\} & \textit{if } m = v! \\ \{v?\} & \textit{if } m = v? \\ \{v!, v?\} & \textit{if } m = v!? \end{cases} \qquad pre(e) = \begin{cases} \{c!\} \cup res(m) & \textit{if } e = c!m \\ \{c?\} & \textit{if } e = c?m \end{cases}$$

The traces model allows recording events just like in CSP. Additionally, it also records the value of the alphabet before and after every event in a trace. Said differently, the model allows recording alphabet changes. Hence, a trace is given by a sequence of the form $\langle \alpha_1, E_1, \alpha_2, ..., E_n, \alpha_n \rangle$. Since not every such sequence is admissible, the following properties characterise the set of legal traces.

Definition 3.2.6 (Concurrency, Ownership, and Synchronisation properties [67, §4.2, 4.3])**.**

$$\begin{aligned} &\textbf{\textit{Ccy}} && \forall\, e, e' \in E \bullet e \neq e' \Rightarrow pre(e) \cap pre(e') = \{\} \\ &\textbf{\textit{Own}} && \forall\, e \in E \bullet pre(e) \subseteq \alpha \\ &\textbf{\textit{Sync}} && \{c!, c?\} \subseteq \alpha \Rightarrow \forall\, m \bullet c!m \in E \Leftrightarrow c?m \in E \end{aligned}$$ □

Own means that an event set $e \in E$ may be recorded only if the corresponding pre-alphabet is already owned i.e. $pre(e) \subseteq \alpha$. This means that if a channel is not in the current alphabet none of its related events may appear in the trace. ***Ccy*** states that two sets of events may interleave if they have disjoint pre-alphabets. ***Sync*** states that synchronisation occurs when both ends of a channel are in the alphabet at the same time.

The syntax and semantics of processes defined in [67, §5] are summarised below:

Definition 3.2.7 (Traces of a process)**.**

$$
\begin{aligned}
traces(P \parallel Q) &= traces(P) * traces(Q) \\
traces(P \fatsemi Q) &= traces(P) \fatsemi traces(Q) \\
traces(P + Q) &= traces(P) \vee traces(Q) \\
traces(SKIP) &= \mathbf{skip} \\
traces(STOP) &= \mathbf{skip} \wedge \neg\, \mathbf{notcompleted} \\
traces(\mathbf{new}\, x.P) &= \exists\, x \bullet \mathbf{expand}(\{\}[]\{x!, x?\}\checkmark) \fatsemi traces(P) \\
traces(\mathbf{dispose}\, x) &= \mathbf{expand}(\{x!, x?\}[]\{\}\checkmark) \\
traces(x!z\rho) &= \begin{pmatrix} traces(STOP) \lhd x? \in res(z\rho) \rhd \\ \mathbf{expand}(\ pre(x!z\rho)[x!z\rho](\{x!\} \setminus res(z\rho))\checkmark\ ) \end{pmatrix} \\
traces(x?(y\rho).P) &= \begin{pmatrix} \exists\, y \bullet x? \notin res(y\rho) \wedge \\ \mathbf{expand}(\ \{x?\}[x?(y\rho)](\{x?\} \uplus res(y\rho))\checkmark\ ) \fatsemi traces(P) \end{pmatrix}
\end{aligned}
$$

□

An informal presentation of the notation is given hereafter.

$* : T \times T \to T$ is defined for traces of equal length only, and such that

$$\langle \alpha_1, E_1, ..., E_n, \alpha_n \rangle * \langle \beta_1, F_1, ..., F_n, \beta_n \rangle = \langle \alpha_1 \uplus \beta_1, E_1 \uplus F_1, ..., E_n \uplus F_n, \alpha_n \uplus \beta_n \rangle$$

$\fatsemi : T \times T \to T$ is the traditional sequential composition of traces.

Termination is modelled by the event $\checkmark$; **complete** denotes any terminated trace; and **skip** denotes any trace of either of the forms

$$\langle \alpha_1, \{\}, ..., \{\}, \alpha_n \rangle \qquad \langle \alpha_1, \{\}, ..., \{\}, \alpha_n, \checkmark \rangle$$

expand is the function that inserts empty events (or sets of events) into a given trace. Its introduction is made necessary because of the definition of the function $*$. Intuitively, it serves for defining all possible interleaving (and synchronisation) under parallel composition. Technically, the definition of **expand** is not as trivial as it may seem to be, and we refer the reader to [67] for more detail.

Allocation. **new** $x.P$ is the process that increases the after-alphabet with the pre-alphabet of x and then behaves like P. If ch is the channel contained in x, then the alphabet (at the end of the operation viz. the after-alphabet) is incremented with the two ends of ch, i.e. the set $\{ch?, ch!\}$. Existential quantification ensures that the value of x, i.e. ch, is unique up to

the point of **new** x. No event occurrence is associated with this process.

Deallocation. **dispose** x is the process that decreases the current alphabet with the value of x. Like allocation, there is no corresponding event occurrence.

Channel-passing: move out. Output of an existing channel y through a different channel x increases the trace with the event $x!y\rho$ ($\rho \neq \epsilon$), and *decreases* from the after-alphabet the pre-alphabet of y. In this model, it is possible for a channel to send itself as a message. However, communications of the form $x!(x?)$ are bad since there can be no receiver: this is deadlock, or the process $STOP$. Communications of the form $x!(x!)$ are valid.

Channel-passing: move in. Input of a new channel z through a different channel x increases the trace with the event $x?z\rho$ ($\rho \neq \epsilon$), and *increases* the after-alphabet with the pre-alphabet of z.

Discussion. The model just described, in the words of its authors, should be considered to be preliminary in nature. However, this should not prevent us from making a few remarks. There are a few design decisions worth pointing out, as they contribute to making the model *intuitively* correct. These design decisions appear most clearly in light of the work of Roscoe [108], presented in §3.2.6.

Two decisions appear as more fundamental: (i) processes must have disjoint alphabets at all times; and (ii) the alphabet of a process is finite. Let us consider the allocation operation. Let $P = \mathbf{new}\, x.P_1$. In a sequential composition $Q \fatsemi P$, existential quantification is enough to guarantee the uniqueness of x, since αQ is finite. Then, any $x \in N \setminus \alpha Q$ is valid. In a parallel composition $P \parallel Q$, things are slightly more complicated.

First, assume that there is no allocation in Q. Because N is infinite, there is a possibility that $x \in N$ and also $x \in \alpha Q$ (before the allocation in P). This issue is resolved by the disjointness condition over processes; the consequence is that all conflicting traces will be eliminated, leaving only expected traces.

Now, suppose that $Q = \mathbf{new}\, y.Q_1$. In Q, any $y \in N \setminus \alpha Q$ is valid. In $P \parallel Q$, things are slightly more complicated, but, the infinity of N actually guarantees that all clashes can be eliminated. To see this, externalise (to the leftmost) the existential quantification as follows. Suppose that P and Q respectively have a number of **new** x.**skip** in their definitions — this generalises the case discussed above. Then, let $\overrightarrow{x} = (x_1, ..., x_n)$ denote the vector of new variables in P, and let $\overrightarrow{y} = (y_1, ..., y_n)$ denote the same but for Q. The indices denote the sequential order of evaluation of each allocation operation. Let α_0 denote the value of the alphabet before the first allocation. Then, it is possible to determine statically to which set x_i will belong upon existential quantification:

$$x_1 \in (N - \alpha_0 P), ..., x_n \in (N - \alpha_0 P - \{x_i \mid i \leq n\})$$

A similar development may be obtained for Q by replacing x and $\alpha_0 P$ above by y and $\alpha_0 Q$, respectively. Because alphabets are finite, the value of N will never be exhausted in each process individually. When composed in parallel, we can have a leftmost existential composition by considering, this time, all the possible interleaving of x_i and y_j such that the above development is correct. Again, because traces are finite whilst channel names are infinite, it is impossible of running out of new names.

A question concerning the set of names N may be raised, however. Where do these names come from? How do they relate to a given process? What do names in N characterise? Since the language is CSP-like, what would the equivalent of N be in the context of CSP? (cf. Chapter 4, Discussion)

Let $(\alpha_{bef}, P, \alpha_{aft})$ denote the process P with before-alphabet α_{bef}, and after-alphabet α_{bef}. Consider the process $(P + Q) \fatsemi R$, with $\alpha_{aft} P \neq \alpha_{aft} Q$. Then, R must have two branches: $(\alpha_{bef} R = \alpha_{aft} P, R, \alpha_{aft})$ and $(\alpha_{bef} R = \alpha_{aft} Q, R, \alpha_{aft})$. The question is thus if the latter two processes are equivalent. Clearly, they differ only in their initial interface; but after that has been fixed, we expect them both to behave like R.

Let $R = \mathbf{new}\, x.R_1$. If $P \fatsemi R$, then according to our earlier analysis, $x \in N - \alpha_{aft} P$. If $Q \fatsemi R$, then $x \in N - \alpha_{aft} Q$. This means that the behaviour of a process depends entirely on its alphabet. However, there are some processes for which such is not the case. Take a Mobile Communications network, or the Internet; they grow and shrink, and yet they still behave in a certain way. For such systems, it would certainly help being able to define a process that takes whatever initial alphabet and yet behaves according to a certain *pattern*. In this sense hence, the above model is quite deterministic. In fact, the definition of $\mathbf{new}\, x.P$ actually restricts the alphabet of P, so that the alphabet calculus may not be as obvious as it seems to be.

As a final remark, whilst the language above is illustrative, its extension with the hiding operator may raise a few issues. For example, it would be difficult of hiding allocated channels, their value being existentially quantified means that they cannot be known in advance, by no means. A solution would be to hide any channel not in the existing alphabet, but this is too radical a solution. Another solution could be of renaming any new name by some name that is guaranteed to be new. For that, it is necessary to define a set X of channels that is obviously disjoint from N. But then, we would be facing further complications.
On the other hand, let us input a new channel y through some distinct channel ch, then hide ch but not y, i.e.

$$(ch?(y).\mathbf{skip}) \setminus \{ch\} = (\mathbf{new}\, y.\mathbf{skip})$$

That is, allocation may be seen as increasing the alphabet silently or rather internally since its effect on the alphabet is visible externally. As a consequence, the choice of increasing an alphabet with both ends of an allocated channel appears more clearly as a direct consequence of the disjointness condition over process alphabets. Indeed, if one end of the channel

only is generated internally, there is no reason why that end would not already be outside. This allocation model is similar to the one used in [58] for extending FOCUS with mobility. [108] shows how different semantics may be constructed using slightly different assumptions. The traces model is similar to the one in [132], which does not have allocation/deallocation operations.

3.2.5 CSP-like localised traces model for pi-calculus processes

Peschanski simple locations [96]. In this paper, Peschanski proposes two approaches to giving denotational semantics to the pi-calculus, with the aim of providing a traces model like that of CSP.

The first approach consists of building a traces model based on the LTS (Linear Transition System) of pi-calculus processes [86]. In order to preserve branching information into the trace, the pi-calculus model of actions is extended so that each action now also possesses a location that uniquely determines their position on the tree described by the LTS. Syntactically, an action act becomes a localised action (loc, act) ($act :: loc$ in [96]). The traces of a pi-calculus process are then calculated by induction on the transitions of the process. In particular, the traces model is built for early semantics; traces equivalence is shown to coincide with early bisimulation equivalence. The traces model fails, however, to provide late bisimulation. Also, traces equivalence holds only between processes that share the same locations.

In the second approach, the traces model is built upon process terms directly, as in CSP. The semantics of restriction $\nu(z)P$ are based on renaming: z is replaced by a new name ν_z assumed to be globally unique. Hence, names are guaranteed to be fresh. Again, traces equivalence is restricted to processes that share the same set of locations.

Bialkiewicz and Peschanski [18]. The approach here is similar to that of the second model of Peschanski [96] in which the traces model is built directly upon process terms.
A major change is the model for locations. In the previous model, the branching structure of processes could not be inferred from the traces. In this model, branching is explicit since the location of an action is now relative to the process to which it belongs. Hence, instead of inherently localised actions (loc, act) as before, there is a locator function that assigns a location to an action based on the structure of processes. When the + operator is encountered, it creates a node with location nl; each branch is then adjoined a location (nl, bl_i) where i denotes the *ith* branch of the choice.

The semantics of a process is given by its *localised traces*. Such a trace has two components: a traditional trace function, which builds sequences of actions out of process terms; and a locator function, which assigns locations to each action in a trace.
Each location is unique, and may be constructed from the set

$$\{\epsilon, (s, node, branch), (w, node, branch)\}$$

ϵ is called the *empty locator*; this is the location of every action, by default. Also, ϵ is always the location of the first action in a trace, or of the empty trace. $node \in NLocs$ is the location of the current node. There are only two kinds of nodes: non-branching nodes, whose location is $node = \epsilon$; and branching nodes, each identified by a unique index in $NLocs \subseteq \mathbb{N} \setminus \{0\}$. $branch \in I \subset \mathbb{N} \setminus \{0\}$ indicates the current branch of the action, relative to the last *node*: $branch = 1$ if there is no branch, otherwise $branch \subseteq \{1, 2, ..\}$ ($branch \neq \{1\}$).

The locator function is based on a relocation mechanism: the first action in the trace, whose default location is ϵ, is not relocalised; the next action, if there is no branch, is relocalised from ϵ to $(nl, bl) = (\epsilon, 1)$, or equivalently, to $\epsilon.1$. Hence, the *kth* successive action in a single line will be assigned location $\epsilon.1..1$ (with k 1s). If the next action is in a branch, say the *nth* one, it will be relocalised from ϵ to say $(nl, bl) = (-1, n)$, or either $(nl, bl) = (-4, n)$, as long as $nl \neq \epsilon$ is unique. The minus $(-)$ sign is simply to avoid confusing nodes with branches. Then, successive nodes in that branch will have locations $(\epsilon..(-1, n), bl)$.

Finally, the silent τ action is distinguished from other actions $\alpha \neq \tau$ by appending to locations a sign from the set $\{w, s\}$. w stands for a *weak locator* and designates τ actions, whereas s stands for a *strong locator* and designates non-τ actions.

Definition 3.2.8 (Locator and traces functions)**.** *Let $sw \in \{w, s\}$ stand for either a strong or a weak locator, and let $\lambda \in \{\tau, \alpha\}$ stand for either a τ action or any action $\alpha \neq \tau$.*

$$
\begin{array}{ll}
loc(0) = \epsilon & tr(0) = \langle\rangle \\
loc(\lambda.0) = (sw, \epsilon) & tr(\lambda.0) = \langle\lambda\rangle \\
loc\big((\lambda.0)[\epsilon \leftarrow (-nl, bl)]\big) = (sw, -nl, bl) & tr\big((\lambda.0)[\epsilon \leftarrow (-nl, bl)]\big) = tr(\lambda.0) \\
loc(\lambda.P) = loc(\lambda.0).loc(P[\epsilon \leftarrow \epsilon.1]) & tr(\lambda.P) = \langle\lambda\rangle \frown tr(P) \\
loc(\lambda.P +_{nl} \beta.Q) = \begin{pmatrix} loc\big((\lambda.0)[\epsilon \leftarrow (-nl, 1)].P\big) + \\ loc\big((\beta.0)[\epsilon \leftarrow (-nl, 2)].Q\big) \end{pmatrix} & tr(\lambda.P + \beta.Q) = tr(\lambda.P) \cup tr(\beta.Q) \\
loc(\lambda.P \parallel \beta.Q) = loc(\lambda.P) \parallel loc(\beta.Q) & tr(\lambda.P \parallel \beta.Q) = tr(\lambda.P) \parallel tr(\beta.Q)
\end{array}
$$

□

The locator function $loc()$ may be understood as building sequences of locations, just like the traces function builds sequences of actions. Both functions always yield sequences of the same length when applied to the same process. The localised trace of a process is simply the trace containing pairs $\big(loc(\alpha.0), \alpha\big)_i$ formed of the *ith* elements of each $loc()$ and $tr()$.

Definition 3.2.9 (Localised traces [18, Def. 22])**.** $Ltr(P) \in loc(P) \times tr(P)$ □

Let $absloc(\alpha_n)$ denote the *absolute location* of an observation α_n, within a sequence $\langle\big(loc(\alpha_1.0), \alpha\big), ..., \big(loc(\alpha_n.0), \alpha\big), ...\rangle$. Then [18, Def. 5]:

$$absloc(\alpha_n) = loc(\alpha_1.0). \ \ldots \ .loc(\alpha_n.0)$$

The input, output and restriction operations are the most important wrt. channel mobility. Their semantics are given below and complete Def. 3.2.8.

Output prefix. The output $c!x$ of a name x has the traditional effect, when restriction is not involved. Only names are sent, and not their location at the time of the sending.

$$loc(c!x.P) = loc(c!x).loc(P) \quad \wedge \quad tr(c!x.P) = \langle c.x \rangle \frown tr(P)$$

or equivalently [18, Def. 22]

$$Ltr(c!x.P) = \langle (loc(c!x), c.x) \rangle \frown Ltr(P)$$

Input prefix. The input $c?y$ of a name y has a renaming effect: the received name y is mapped onto a generated name $\rho_{absloc(c.y)}$. The generated name is unique by construction since locations are unique. Again, only names are received, without their previous location.

$$loc(c?y.P) = loc(c?y).loc(P) \quad \wedge \quad tr(c?y.P) = \langle c.y \rangle \frown tr(P[\rho_{absloc(c.y)}/y])$$

or equivalently [18, Def. 22]

$$Ltr(c?y.P) = \langle (loc(c?y), c.y) \rangle \frown Ltr(P[\rho_{absloc(c.y)}/y])$$

Restriction. The effect of restriction $\nu(z)P$ is, basically, the replacement of the name z in the trace by a unique, generated name $\nu_{absloc(z)}$, within the scope defined by $\nu(z)$. Notice the difference with the earlier model (Peschanski [96]): here, the uniqueness of names is not *assumed*, but is guaranteed by construction from the uniqueness of locations $absloc(z)$. If there is no scope extrusion, the scope of $\nu(z)P$ is the whole of P; otherwise, its scope reaches as far as the extrusion of z viz. $c!z.0$. Restriction is hence defined recursively on process terms as follows:

$$\begin{aligned} \nu(z)(\alpha.P) &= \alpha.\nu(z)P \\ \nu(z)(c?z.P) &= c?\rho_{absloc(z)}.\nu(z)P \\ \nu(z)(c!z.P) &= c!\nu_{absloc(z)}.P \end{aligned}$$

Interestingly, name clashes are resolved easily thanks to the fact that any received name z is replaced by a unique name ρ_l. The semantics of restriction are given by the left hand side of the previous equations based on the earlier semantics of $loc()$ and $tr()$ i.e.

$$loc(\nu(z)(\lambda\,.P)) = loc(\lambda\,.0).loc(\nu(z)P) \quad \wedge \quad tr(\nu(z)(\lambda\,.P)) = tr(\lambda\,.0) \frown tr(\nu(z)P)$$

or equivalently [18, Def. 22]

$$Ltr(\nu(z)(\lambda\,.P)) = Ltr(\lambda\,.0).Ltr(\nu(z)P)$$

where $\lambda \in \{\alpha, c?z, c!z\}$.

Traces are equivalent up to the renaming of locations i.e. two processes are shown to be equivalent if there is a valid substitution of the locations of the one that yields the other: this is called *split-equivalence*. The trivial form of split-equivalence is when two processes have the same locations, and is called *localised equivalence* instead. It is not entirely satisfactory however, e.g. it does not preserve the property $P + P = P$ due to locations.

Definition 3.2.10 (Localised- and split-equivalence)**.** *Two processes P and Q are said to be* equivalent*, written $P =_L Q$ if*

$$tr(P) = tr(Q) \wedge loc(P) = loc(Q)$$

or equivalently [18, §4]

$$Ltr(P) = Ltr(Q)$$

They are said to be split-equivalent*, written $P =_{u,v} Q$ if*

$$tr(P) = tr(Q) \wedge \exists\, u, v \mid \textit{two functions} \bullet u \circ loc(P) = v \circ loc(Q)$$

or equivalently [18, Def. 24]

$$u \circ Ltr(P) = v \circ Ltr(Q)$$ □

Proving split-equivalence from the above definition is not trivial and fails in most cases. Hence, a normalisation proof technique has been developed to make such proofs easier. The normalisation technique is based on ideas of rewrite systems, and is quite complex ([18]).

Discussion. In the presentation above we have insisted on the separation between the locator and the traces functions in order to emphasize the former. Indeed, the locator function is rather sophisticated, but implements one single specification, namely, that names be *fresh*. It is as if the assumption that locations are unique in the simple location model ([96]) had been implemented into the locator function.

The relocation mechanism bears some resemblance with the relabelling mechanism of Roscoe [108], especially the standardised fresh names operator (*SFN*).
The localised traces model is significantly complex compared with the traditional CSP traces model, even when there is no channel mobility.

The traces model is obviously targeted to capture pi-calculus processes, hence, the language provided is restrictive. It is not evident how one could extend it with more CSP-like operators, nor how to relate the language to CSP itself.

It is claimed that a difference with CSP, and likeness with the pi-calculus, of the above model, is that channels are *concrete* (or first class citizens, from a language point of view).

This is similar to the model of Hoare & O'Hearn presented in the previous section. Unfortunately, there is no discussion (in either models) of the implications of the fact on the semantics provided.

3.2.6 CSP-like operational semantics

CSP-like operators

In [107] Roscoe proposes a definition of what it means for an operator, with given operational semantics, to be CSP-like viz. the operator may be defined by means of CSP operational semantics. The definition of CSP-like operators relies on a mechanism for transforming SOS (Structured Operational Semantics [99]) rules into linear rules, which have a particular format. The rationale for the transformation is that operators may be seen as defining, given a list of *on* and another list of *off* arguments, what arguments will be *on* and *off* next, respectively. Following such a transformation, two properties characterise CSP-like operators [107, §3]:

- There is one rule for each *on* argument representing the promotion of a τ without otherwise influencing the state. No other rule has a τ as a contributing action. This restriction corresponds to the idea that operators are not aware of the τ actions of their operands, and cannot take any positive action on account of these.

- Each *on* argument can only appear at most once amongst the arguments of $op(..)$. This represents the idea that a process which is up and running may not be cloned and then possibly compared against itself. This restriction is necessary in order to model processes by descriptions of individual linear runs — into which category fall all the types of behaviour used in the CSP modelling approach.

CCS processes are shown to be CSP-like. Some preliminary results about the pi-calculus are presented, but a fuller treatment is postponed to another paper [108]. Before presenting the latter, we first make a few comments on the above work.

Discussion. This work suggests a way of building denotational semantics for channel mobility in CSP:

i) given some operational semantics for channel mobility, make a translation into CSP operational semantics; and

ii) if possible, derive the denotational semantics from the CSP operational semantics.

A major problem with this work is the absence of a definition of operational semantics (for processes) that is independent from CSP. Another problem is that the linearisation adds much complexity without making reasoning any easier. We may contrast this work with the work in [120].

In [120], de Simone builds an algebraic framework that allows him to compare and combine two process calculi, MEIJE and CCS. It can be seen from the definition of a (conditional behaviour) rule below that the SOS' formulation gives at a glance the same information as Roscoe's linearisation. Also, the constraints on *on* and *off* arguments are exactly the same as stated above such that they may not be considered to be more characteristic of CSP than of other process calculi. Instead, based on de Simone's framework, CSP would have been formally defined as a process calculus according to the following definition:

Definition 3.2.11 (MEIJE-CCS Process calculus [120, Def. 1.8])**.** *A* Process Calculus *is a triple*
$(F, M, Spec)$*, where* F *is an operator family,* M *is an action set (here we* [de Simone] *shall content ourselves with the simple action/signal monoid over an alphabet; we shall not consider typing operators with different action sets), and Spec is a function assigning a specification to an operator. We shall deal next with what our universe of specifications is.* □

Definition 3.2.12 (MEIJE-CCS Specification [120, Def. 1.9])**.** *The general form of specification we will allow for each operator* F *will consist in a number of conditional behaviour rules. A* conditional behaviour rule *rg is an object of the following shape:*

$$\frac{\forall\, j \in S \bullet p_{i_j} \xrightarrow{u_j} p'_{i_j} \ \wedge\ Pr(u_1, ..., u_l, v)}{F(p_1, ..., p_n) \xrightarrow{v} \mathbf{T}}$$

where $S = \{i_1, ..., i_l\} \subset \{1, ..., ar(F)\}$*,* Pr *is an* $(l+1)$*-ary relation on the action monoid* M*, that is, a subset of* M^{n+1}*, and* $\mathbf{T}$ *is an architectural expression of the process calculus. Indeed, due to the calculus,* $\mathbf{T}$ *may only contain linearly those of the* p_i*s that were not required acting in the condition, and the* p'_i*s that were due to act in that same condition. Thus, it stays an architectural expression.* □

According to Roscoe's results (viz. CCS is CSP-like although the opposite does not hold) and the previous definitions, CCS may be taken as subcalculus of CSP. More interesting would be the characterisation of the pi-calculus. Although MEIJE-CCS does not handle channels specifically, the set $\mathcal{A}$ of actions of a process may still be specified although it does not play a fundamental role in the semantics. However, it would seem that a characterisation of the pi-calculus would require making $\mathcal{A}$ more explicit and also variable (cf. §3.4 below for more discussion).

The pi-calculus is CSP-like

In [108], Roscoe defines what it means for pi-calculus operators to be CSP-like. This is done first by defining a new operator called *generalised relabelling*, which is then used as the basis for giving semantics to the pi-calculus restriction operator, and hence, to scope extrusion. Generalised relabelling has two features related to how it relabels existing names: the replacement mapping may

- vary as the process progresses;

- forbid certain visible actions, which will then map to an empty choice of (replacement) options.

Definition 3.2.13 (Generalised relabelling [108, §3])**.** *Let G be a* generalised relabelling relation *on $\mathcal{A}$ where $(a, t, x) \in G$ says that an event a that occurs after some trace t (i.e. $t \frown \langle a \rangle$), may be replaced by some (possibly distinct) event x. Let $P\langle\langle G \rangle\rangle$ denote the process that can perform an x whenever P performs a. Then $P\langle\langle G \rangle\rangle$ is defined by the following rules:*

$$\frac{P \xrightarrow{\tau} P'}{P\langle\langle G \rangle\rangle \xrightarrow{\tau} P'\langle\langle G \rangle\rangle} \qquad \frac{P \xrightarrow{a} P' \; (a, \langle\rangle, x) \in G}{P\langle\langle G \rangle\rangle \xrightarrow{x} P'\langle\langle G/\langle a \rangle \rangle\rangle}$$

where $G/t = \{(a, s, x) \mid (s, t \frown s, x) \in G\}$.
An equivalent formulation may be obtained by running P in parallel with a process that chooses the renaming.
Let $E = \{(a, (a, x)) \mid a \in \mathcal{A} \wedge x \in \mathcal{A} \cup \{\tau\}\}$ be the function that renames an event a to some event pair (a, x). The reverse renaming is denoted by $C = \{((a, x), a) \mid a \in \mathcal{A} \wedge x \in \mathcal{A} \cup \{\tau\}\}$. Let $Reg(G)$ be the process that selects pairs (a, x) for a given generalised relabelling relation G, defined by

$$Reg(G) \mathrel{\widehat{=}} \Box\{(a, x) \to Reg(G/\langle a \rangle) \mid (a, \langle\rangle, x) \in G\}$$

Then

$$P\langle\langle G \rangle\rangle \mathrel{\widehat{=}} (P[\![E]\!] \parallel Reg(G)[\![C]\!]) \setminus \{\tau\}$$

where $P[\![E]\!]$ is the renaming operator, which yields a process that behaves like P but with events in P renamed as specified in the relation E. □

Note: The relation G defined above is considered to be finite.

In [108], Roscoe first establishes the relation between CCS and CSP, given that many operators of CCS are shared with the pi-calculus. The pi-calculus is then considered, with an emphasis on restriction and scope extrusion. We give a summary hereafter.

The effect of restriction $\nu(a)P$ is to bind the name a to some name x that may be guaranteed of being fresh. Said differently $\nu(a)$ is a name generator; however, freshness guarantee is the root of all the problems. A name a is *fresh* if the process did not *use* it prior to its first appearance in $\nu(a)$. This is equivalent to stating that the process did not *know* of the name before said first appearance. Different models may be adopted for the formalisation of the concept of *process' knowledge of a name.*

Let *Names* denote the universal set of names, representing the actions that may be used by any process. Then, according to how the set *Names* is distributed between a process and

its environment, two models are possible:

- the *unified approach.* Here, the set *Names* is shared by both the process and its environment. It may be further partitioned into two disjoint subsets: K the set of names currently known both of the process and its environment; and N the set of *eventually* fresh names.
  - There is an *assumption* that names in N are *unique* to the process, but nothing prevents the contrary from occurring.
  - The two sets must always be disjoint, i.e. $K \cap N = \{\}$.
  - It is expected that when N decreases thanks to scope extrusion, K increases; additionally, K increases when the process receives a name not in $K \cup N$. Such a property is enforced by specifying the successive values of both K and N recursively, in the process expression. *Hence, at any time, a process may only engage in the specified values of both K and N.*
- the *bipartite approach.* Here, the set *Names* is partitioned between the process and its environment.

For conciseness we shall present the unified approach only. In the unified approach many relabelling relations may be defined, depending on how one wants to implement freshness. (This is equally true in the bipartite approach).

Notation: unless stated otherwise the set *Names* is assumed to be countably infinite with a fixed enumeration $\{n_0, n_1, ...\}$. If L is a nonempty subset of names then $\mu(L)$ is the name of least index in L. $\overline{L} = Names - L$ denotes the complement of L in *Names* [108].

Remark 1: the semantics of G earlier makes it look like G were defined from the outset. However, G is in fact defined by recursion over process expressions. The renaming relations defined subsequently show this clearly.

Remark 2 (**the naming model**): In CSP two models of names may be used. In the first, names represent the objects (named) themselves, i.e. if x is the name of a channel, it also represents that channel. In the second model, names are variables, which map to other objects themselves represented by distinct names, i.e. the name x used earlier may map to different names, for example n_1 and n_2. The second model may be referred to as the *(naming) environment model*: there is an evaluation function which maps every name x to its value, say n at the time of the observation. *This second naming model is the one used here.* The naming environment must not be confused with the environment of a process (or process environment). The context may clarify which one is used, though we shall use "(naming) environment" when we do *not* mean the process environment, and environment otherwise.

A first, naive approach consists of having a process and the environment deal with fresh names individually. This yields a relabelling relation, denoted by $OF(K, N, P)$ (read Output First), and defined as follows [108, §5.1]:

$$Reg(OF(N,K,\xi)) \mathrel{\widehat{=}} \Box \left(\begin{array}{l} \left(\begin{array}{ll} \{(a.b, \xi(a).\xi(b)) \to Reg(OF(K,N,\xi)) & | \\ \quad a,b \in K \wedge a \in \{a?, a!\}\} & \end{array} \right) \cup \\ \left(\begin{array}{ll} \{(a!b, \xi(a)!\xi(b)) \to Reg(OF(K \cup \{b\}, N \setminus \{b\}, \xi)) & | \\ \quad a \in K, b \in N\} & \end{array} \right) \cup \\ \left(\begin{array}{ll} \{(a?b, \xi(a)?\xi(b)) \to Reg(OF(K \cup \{b\}, N, \xi) & | \\ \quad a \in K, b \notin K \cup N\} & \end{array} \right) \cup \\ \left(\begin{array}{ll} \{(a.b, \xi(a).\xi(b)) \to Reg(OF(K \cup \{n\}, N, \xi \circ xp(b,n)) & | \\ \quad a \in K, b \in N, n = \mu(\overline{K \cup N})\} & \end{array} \right) \end{array} \right)$$

where $\xi(L)$ represents the (naming) environment's view of a set of names L. In other words ξ is the name evaluation function mentioned above (cf. Remark 2).
Pairs $(a.b, \xi(a).\xi(b))$ stand for pairs (a, x) used in the definition of generalised relabelling G earlier. $\xi(a)$ specifies what the correct mapping should be at a given time.

The first clause states that, in a communication $a.b$, when both names a, b are known, no renaming occurs; the next two clauses state what happens when b is fresh ($b \in N$) viz. K and N are updated accordingly; the last clause states what happens in case of name collision, viz. the fresh name of b, say n, is added into K (sort of 'un'-freshed), and b renamed to a new fresh name. $\xi \circ xp(b, n)$ yields the next mapping upon name collision, i.e. from $b \mapsto n$, $n \in N$ (before collision) to $b \mapsto \mu(\overline{K \cup N})$ upon collision.

In a second approach, instead of having the process and the environment deal individually with fresh names, a third process handles fresh names for both. The corresponding relabelling relation, denoted by $NFN(N, Q)$ (read Nondeterministic Fresh Names) is defined as follows [108, §5.2]:

$$NFN(N,Q) \mathrel{\widehat{=}} \bigsqcap \{ Q[\![\xi \cup id_{\overline{N}}]\!] \mid \xi : N \to N \text{ , a bijection} \}$$

The set K has been dropped out because there is a single set of fresh names. Hence no matter what either the process or its environment knows, restriction ($\nu(z)$) will generate a fresh name. $id_{\overline{N}}$ is a collection of mappings for unknown names (viz. not specified by the mapping ξ). $Q[\![\xi \cup id_{\overline{N}}]\!]$ behaves like Q, with potential renaming defined by $id_{\overline{N}}$. This second approach may hence be considered a little more abstract than the previous one.

In yet a third approach, unlike previously, names may not be chosen randomly. Indeed, since names (in the set *Names*) are indexed with a subset of the natural numbers, it is enough to generate a new number, always the smallest, every time a fresh name is requested. The corresponding generalised relabelling relation, denoted by $SFN(K, N, P)$ (read Standardised

Fresh Names) is defined as follows [108, §5.3]:

$$Reg(SFN(N,K,\xi)) \mathrel{\widehat{=}} \Box \left(\begin{array}{l} \left(\begin{array}{l} \{(a.b, \xi(a).\xi(b)) \to Reg(SFN(K,N,\xi)) \mid \\ \qquad a,b \in K \wedge a \in \{a?, a!\}\} \end{array} \right) \cup \\ \left(\begin{array}{l} \{(a!b, \xi(a)!\xi(b)) \to Reg(SFN \left(\begin{array}{l} K \cup \{\mu(N)\}, \\ N \setminus \{\mu(N)\}, \\ \xi + [b \mapsto \mu(N)] \end{array} \right)) \mid \\ \qquad a \in K, b \in \xi(K)\} \end{array} \right) \cup \\ \left(\begin{array}{l} \{(a?b, \xi(a)?\xi(b)) \to Reg(SFN \left(\begin{array}{l} K \cup \{b\}, N, \\ \xi + [b \mapsto \mu(\overline{K \cup N})] \end{array} \right)) \mid \\ \qquad a \in K, b \notin K\} \end{array} \right) \end{array} \right)$$

The clauses above are similar to the ones for $OF()$ relabelling, except for the clause about name collision. Indeed, name collision is resolved by the way fresh names are generated: the next fresh name to be extruded is always the smallest member of N, namely $\mu(N)$. And when a *new* name is input by the process, that name is mapped unto the smallest member of the complement of $K \cup N$, namely $\mu(\overline{K \cup N})$.

In what precedes, the treatment of names has been realised without any distinction between what may be termed *pure* names (of actions excluding communications), and communication events' names. Without such a distinction, the models presented above are simply too general. When channels are considered (so called channel-based CSP models), the previous analysis may be restricted to communication events only. We shall not present the corresponding semantics here however, as it does not add to the understanding of the model.

Remarks. We have taken the liberty of changing certain definitions whenever we found what seemed like typing errors in the original paper. For example, in the definition of $OF(K,N,\xi)$, the second clause has (upon output $a!b$ of a fresh name b) "$(K \cup \{a\}, N \setminus \{a\}, \xi)$" in the original paper, instead of "$(K \cup \{b\}, N \setminus \{b\}, \xi)$" above.

Discussion. The work above presents preliminary results only, and which are quite complex. Most of the complexity seems to come from the fact that the names considered are too general, first when they include every event, and also when they are restricted to communication events. The motivations for such a modelling choice are missing. Notwithstanding, we may argue that *channel mobility may not be modelled through event mobility.* In effect, this is quite counter-intuitive from a modelling point of view: events are meant to characterise what can be recorded into the trace (of the observation of a process's operation over a channel); the movement of such a record, if such is conceivable, may not possibly infer that of the corresponding channel. In other words, there is a difference of nature between channels and events. Were we to use de Simone's algebraic framework [120] as our basis for discussion, clearly, the algebraic characterisation of the set of channels would be quite distinct from the

actions monoid M.

In [107] presented earlier, we pointed out as a limitation the absence of a definition of a process calculus (with operational semantics) that is independent from CSP. There is a striking difference between the approaches in [107] and [108]. In the latter, contrary to expectations, no (functional) relation is established between the (operational semantics of the) pi-calculus, and the operational semantics of CSP. Rather, pi-calculus processes are defined *directly* in terms of CSP, or said differently, a CSP model (operational semantics) for the pi-calculus is proposed but not linked to the pi-calculus itself. Were we in an algebraic framework like that of de Simone mentioned above, and having defined both the pi-calculus and CSP operational semantics, it would be as if some pi-calculus operators had been defined in CSP, but no relation was established between the original pi-calculus operators and their supposedly CSP equivalent.

As a final remark, we may quote the following

> *While this may or may not be apparent to the reader, the author [Roscoe] discovered on numerous occasions that the semantic decisions made in the design of pi-calculus were absolutely crucial to the creation of a reasonably elegant semantics for it in CSP. A prime example of this is the rule that no fresh name can be used as a channel until it has been passed along another channel is necessary for ensuring that the first time a name appears in a $id_{\overline{N}}$ behaviour in a channel-based model is as the "data" field of an actually communicated event. This is key to a number of things working properly in the CSP semantics.* ([108, Conclusion])

Unfortunately, since the characterisation of channels is not discussed in this work a question has been left unanswered: what entitles a name to be considered a channel? Clearly, if it was sufficient for names to be passed around as messages, then such may have already been done in CSP traditional semantics; hence this is not a sufficient condition, though a necessary one. On the other hand, if a process may 'recognise' a name as characterising a channel, this would mean that such a channel was already in the knowledge of the process, for, it is difficult to conceive otherwise how the recognition would be possible.

In relation to the above question, we see that Grosu and Stolen [58] did avoid the problem by a subtlety, namely by assuming that all names are *known* in advance, but only some may be *used* at any given time. Yet, this does not dissolve the difficulty altogether for, if all the names are known in advance, then, we cannot possibly speak of an effective channel mobility. We alluded to a similar problem in the work of Hoare and O'Hearn [67], when we asked questions about the provenance of the names in the set of names N. This problem is in fact common to all the works in the literature, and the six or more models defined in this work [108] do not provide a satisfactory answer.

Closed-world mobile CSP

In ([109], §20.3), Roscoe discusses a way of adding mobility into CSP (the denotational semantics) directly. The semantics are built for a restricted type of mobility where the set of

channels to be moved is known in advance. An example of such a system which we provide is the Buffer with mobile channels (cf. Chap. 4, §4.5.5).

In this model, Roscoe keeps the original representation for channels found in standard CSP, hence, channels are not represented as concrete entities (or first class citizens). The model considers event mobility in general, with the possibility of a restriction to channel names. Releasing an event reduces the alphabet of the sender whilst increasing that of the receiver, and vice versa.

Discussion. The basis for a model of channel mobility in CSP are outlined, but no model is itself defined. We have found some aspects of the work confusing, because some terminology is left undefined. For example, the statement *the set of channels to be moved is known in advance*. By whom?

- Not by every process, since an example is then given of a parallel composition in which a process receives a channel whose name it did not *know* before.
- The correct answer would be the designer, because he can then use that known channel name for specifying the correct interface for the next process (in an action prefix, and in a sequential composition), in the same way as Vajar et al. [132], for example.

Unfortunately, even this definition of closed-world semantics already appears to be problematic, from a compositional point of view. Indeed, the opposite concept of *open-world semantics* suggests that unspecified names may be received. However, the closed-world semantics already used processes P and Q, which may themselves be considered as open, although their parallel composition is closed. As a consequence, it does not seem that a definition of channel mobility should rely on the designer's (pre)knowledge of movable or mobile channels.

3.3 Other Works

Fully Abstract semantics of the pi-calculus. In [48], [122], and [62], the authors explore the issue of giving denotational semantics for the pi-calculus on the basis of category theory. The task is not trivial as it requires defining a category for pi-calculus processes first, and then finding suitable morphisms for giving their semantics. Many approaches are possible, and so are the results obtained. In [48], only strong bisimulation and congruence were defined; in [122], both early and late equivalence were defined; and in [62], testing equivalence was defined. Category theory may itself serve as a foundation for Computer Science. In that sense, these works have a different theoretical basis from the ones presented earlier.

In [101], Popescu proposes a coalgebraic semantics for the pi-calculus, with the aim of formalising weak early bisimilarity of pi-calculus processes. The approach requires giving operational semantics to pi-calculus processes within the coalgebraic framework first. Then a traces model is built on top of such processes and shown to be fully abstract with regard to the operational semantics. The traces model defines a denotational semantics whose domain

is a coalgebra instead of a more standard mathematical domain, e.g. a set of actions (as in CSP). An interesting concept of *channel configuration* plays the role of the *interface* between two processes (or between a process and its environment), and may *change* with the mobility of channels. There is a distinction between *private* links, unknown of the environment, and *public* links, known of the environment. Scope extrusion makes links that were previously private become public.

As a general remark:

> *The above mentioned research attempts to find mathematical models suited to describe the behaviour of already completed systems. A formalism well-suited for describing an already completed system is not necessarily ideal as a specification language to be used in a process of step-wise system development, or as a notation for formal reasoning and verification.* ([123, Introduction])

Petri Nets + channel mobility. There exists a body of work for extending Petri nets with channel mobility. The survey in [117] may be used as a starting point. By their nature, such works are closer to operational than denotational semantics. [139] also provides a survey containing such works. In particular, both [117] and [139] consider that no single framework may be used to model all the aspects of dynamic reconfiguration. As a consequence, in [139], three models are proposed, each one based on a different formalism (all providing operational semantics).

Strong mobility. We could find a single work only on denotational semantics for strong mobility, in [131]. The work aims at providing strong mobility in the context of Object Oriented programming. The functional programming language Haskell is used as a basis for the denotational semantics, an approach that is not standard, although that is meant as a first step before using standard mathematical domains (not defined in [131]).

The relation between weak mobility and strong mobility is explored in [15]. It is argued that any language suitable for expressing weak mobility may be used to express strong mobility also. Such a view is reasonable if one thinks in terms of implementation since, in general, interrupts are implemented (i.e. provided in programming languages, as a feature thereof) without regard for mobility, and also if we conceive that weak mobility may arise even upon interrupt, as is the case in the literature. For such a reason, in this thesis, we use the term *weak mobility* only when no interrupt occurs, and strong mobility otherwise. This does not prevent one from further classifying strong mobility according to the precision of the interrupt state, however.

3.4 Final considerations

A reading of the literature has not left us oblivious. The many discussions introduced above have hinted towards our appreciation of each work individually. Here, we intend to make general remarks concerning our *intuitions* about an adequate model for mobility, hinting as necessary towards possible improvements of existing work, or how some work have benefited us.

Let us assume a universal set of channel names shared by both a process and its environment. We may at first exclude the notion of the *knowledge* (of a channel) in favour of the *effect of such a knowledge* on the behaviour of processes. This means that a process *does not* need to care about what its environment knows, nor the environment care for what a process knows. In particular, we see that the 'construction' of such a knowledge is empirical (cf. [108]), implying that an update at a single node would of necessity trigger a similar update at all nodes: a sort of *knowledge propagation* of the kind dealt with by routing protocols on the Internet. In sum, any incorporation of such a knowledge may not be abstract enough.

The naming problem that gives rise to the pi-calculus restriction operator would also arise in any CSP model. If *hiding* implies the privatisation of the hidden or silent names, then it is possible of receiving channel names that are meant to be public, and yet equal to the existing hidden ones. The problem becomes even more intricate if we allow a dynamic increase of hidden names. Either way, there can be no satisfying definition of the restriction operator that could rely on the set of hidden channels, precisely because *hiding is independent of the process' intended behaviour*: the process that hides (or expresses hiding), $P \setminus X$, is different from the hidden process, P. We will return to this later on.

As an example of *dynamic hiding*, consider a mobile communication network. Certain real-world events, e.g. for the Olympic games the number of users greatly increases thus requiring a corresponding increase of the (core) network capacity (resources). If the *way for increasing the network' capacity* is modelled as a *channel carrier* (i.e. a channel that communicates other channels as messages), say κch, and the *increase of capacity* as the effect of receiving from the environment new communication channels (from κch), then we have a case of a modification of the internal interface (of the core network) that is yet invisible to the observer (viz. any user of the network).
On the other hand, it is not necessary that the (effect of a) change of the internal interface be absolutely 'invisible'. Indeed, as often happens, a lack of network resources has an effect on the user of the network although he may *not* observe said resources at work. This means that channel mobility may readily be used to model such a behaviour: a look at the interface may tell not only that something is wrong, but also what, e.g. a channel is missing. That is, *channel mobility is adequate for modelling channel faults.*

A possible way of solving the naming problem may be to regard channel mobility from the point of view of the processes only. Hence, without regard for the environment, and without regard for closed/open world issues, or notwithstanding, *a process always knows if it has received (resp. released) a new (resp. old) channel, or not.* And whenever it performs

either action, it interacts with its environment, without regard for *structure*. Structuring information may be added later, as we have suggested in the revision of Welch & Barnes [138] semantics, into some channel naming procedure.
Therefore whether a channel may be used *internally* or *externally* is of no concern to the process: it is enough that the process may use the channel for its communications. And this is only right since *hiding* is independent from the process itself, thus reflecting the fact that a process *does not* need to care about the structure or the architecture of its environment: If a channel (mobile or not) may be used internally, then hide it, otherwise do not. All that we need to ensure is that a process may not move in an existing (channel) name, and may not move out a non-existing one.

Indeed, there is an underlying theory of observation in CSP, that fundamentally differs, at least at first sight, from that of the pi-calculus. Let us elaborate on this. As Roscoe states in [108], the pi-calculus is as much a calculus of names as it is a calculus of channel mobility. In effect, the essential question posed by the pi-calculus is the following: *when do two names, identical syntactically, denote the same object viz. have the same semantics?*, or equivalently *when do two names, identical syntactically, denote the same channel?*

To illustrate this question, consider two programs P and Q written respectively by two distinct programmers, and such that each has its own distinct resources (supposedly). The programs are meant to be run in parallel. A third programmer that reads the two programs and symbolically executes them in parallel may not be able to distinguish between two *names* respectively used by each program. So, the third programmer needs to be told when the two names are identical.

The pi-calculus solves the issue by defining the restriction operator. At a first glance, it seems that the decision of using the restriction operator is imposed by the algebra (in de Simone sense) used to construct the pi-calculus. However, we believe that the choice of the operation for evaluating names (viz. name freshness) is guided by a choice of axioms for defining channels, which is independent from the underlying algebraic framework itself.

In CSP, the axioms that define channels are different. Returning to our earlier remark, we see that when defining some process P, we define its interface as well, as a way of saying that P may use those channels for its communications. This means that P always knows their names, at the very least. This further means that P *does not* need to care about scope extrusion: P always extrudes its names. It is then the environment to decide accordingly if the environment already knows a received name or not. In the other direction, if P already knows a name, then P may simply signal to the environment that it already knows such a name.

With regard to hiding, we see that P is not responsible for the hiding, i.e. it is not P itself that decides what of its channels are hidden. Again, P does not need to care about it. The process responsible for the hiding, $P \setminus X$, is the one responsible for what is external (or internal). Hence, $P \setminus X$ is responsible for preventing external communications through the channels in X (viz. the 'use' of a channel); it has no effect on what channel may be input or

output (viz. the 'communication' or 'transport' or 'movement' of a channel).
On the other hand, if P receives a name corresponding to that of a hidden channel, it remains the case that P knows of such a channel, and hence, can always tell if there is intrusion (the name is new) or not. As for extrusion, if P gets rid of a channel, P does *not* need to be concerned about that channel anymore. Either way, $P \setminus X$ has no say in the matter.

Whence we draw a subtle difference with the pi-calculus' underlying theory of observation: the third observer needs not to resolve (channel) name equality or collision itself: it may just ask each program individually. Hence, there is no need for a CSP process to *generate* fresh names.

This latter remark seems to hold some non-negligible theoretical importance. To cut things short, it means that the main difference, if not the only one, between our modelling approach and the pi-calculus, is that the pi-calculus *is also a calculus of names*. We may elaborate on this by using the notion of *axiomatisation* employed earlier. We will use the algebraic framework of de Simone [120] as our basis for discussion. We may also profit of the occasion to answer the following fundamental question: **what entitles a process the use of a channel?** Or equivalently, **what makes a channel (name) a channel?**

CCS and CSP are then two process calculi that share some operators in common, and differ in some others. Instead of discussing their difference in terms of the individual specification of their respective operators (the function $Spec : F \to Specifications$), we find it easier to discuss that difference in terms of their eventual axiomatisation. Then, we see that CCS and CSP differ mainly in this one axiom that says how to interpret silent τ actions. That is, if we assert with de Simone [120] that three basic elements pervade all process calculi, namely: *non-determinism*, *concurrency*, and *synchronisation*; then, instead of reasoning at the level of the operators for each process calculus, we may rather reason about how each calculus axiomatises each said basic concept. Wherefore we consider that CCS and CSP differ in their set of axioms for synchronisation.

Whilst de Simone proposes an axiomatisation of actions and synchronisation events, he left out that of communication events. But their inclusion poses no major difficulty, and we may assert that CCS and CSP share exactly the same axiomatic characterisation of both channels and communication events. Informally, it may look something like this:

Definition 3.4.1 (Static processes (incl. CCS, CSP) - Axioms for channels)**.**

A1. *Channels exist.*

Corollary A1. *Communication events exist.*

A2. *Channels may be used* only *as objects in communications obj.subj.*

A3. *Let $(\mathcal{A}, \mathcal{I}, P)$ denote a process with alphabet $\mathcal{A}$ and interface $\mathcal{I}$. Then $c.m.P$ is well-defined, i.e. we assume that it is always the case that $c \in \mathcal{I}(c.m.P)$.* □

Axiom **A3** is the most interesting. It says that when evaluating an expression $c.m.P$, channel c is in the interface of process $c.m.P$. That is, process $c.m.P$ is entitled of using any channel in its interface, here c, because $c \in I$. Axiom A3 assumes that I does not change. Let us compare with similar axioms for the pi-calculus this time. We have:

Definition 3.4.2 (Mobile processes (incl. pi-calculus) - Axioms for channels)**.**

B1. *Axiom* **A1** *holds.*

B2. *A channel may be used* either *as object or subject in communications obj.subj.*

B3. *The interface of a process may vary.*

Corollary B3. *$c.m.P$ is well-defined if, and only if, $c \in \mathcal{I}(c.m.P)$.* □

Notice the difference between axioms **A2** and **B2**. The question that may be asked here is: *why does* **A2** *forbid channels from being subject in communications (i.e. from being sent as messages)?* If one has the pi-calculus in mind, the trivial answer would be that **A2** guarantees that the interface of a process may not change. However, it was never stated, in the cases of CCS and CSP, that moving a channel in/out as a message increases/decreases the alphabet of a process. In fact, **A2** may seem to be needlessly strong: we may indeed communicate (move) channels just like we do other messages, and yet not increase the interface. In the end, the answer is in the question: *there is an axiom which states that the acquisition (resp. release) of a channel may increase (resp. decrease) the interface*, accordingly. This is what **B3** does.

Axiom **B3** states two things: the interface of a mobile process may vary, and hence, *must* be evaluated in advance. In fact, **A3** is just the same as **B3**, where the evaluation always yields the same result, whereas **B3** itself supposes that such an evaluation may yield different results, depending notably on what has happened before — given that the interface is now, so to say a *variable*, it may be manipulated by mobile processes, unlike for static processes.

The axioms given above are not specific to any single process calculus, and are independent of any semantics style. Clearly hence, we can envisage an extension of de Simone' algebraic framework with denotational semantics. The operators defining channel mobility would obey the same axiomatic specification. In particular, any extension of either CCS or CSP with mobility should obey the same axioms as does the pi-calculus. In this regard, the pi-calculus is simply one way of implementing channel mobility. More generally, we believe that instead of *generating fresh names*, a more abstract view of channel mobility (than the pi-calculus') is possible, namely, one in which the process calculus is content with "indicating" the current interface. Now, this does not strictly means that the pi-calculus restriction operator is doing otherwise. Rather, it might simply be a difference of *interpretation*.

Indeed, in the model that we propose, since the interface is variable, it means that the interface of a process is *computable*. So, the scope extrusion/inclusion may be seen as realising such a computation in lieu of computing random names. What characterises a name of

being a channel in CSP is the presence of that name in a special set: the interface of the process. Presumably, each name is unique, hence two identical names identify the same channel (axiomatically speaking, every name is bound). If a process receives a known name, it does not increase its interface: this way, the *observer* may *also* know that the received name is not fresh (for the receiving process); if the interface increases, then the observer may also know that the name received is fresh; and if the interface is hidden, then it is not meant to be seen.

3.5 Summary and concluding remarks

A great deal of discussions has been consecrated to channel mobility. The literature review itself shows that channel mobility poses a greater theoretical difficulty than process mobility.

We have presented an informal framework for reasoning about process mobility, [49], and have shown how we intended to use it in our discussion on process mobility. In particular, it has permitted us to justify the existing formalisation for weak mobility in UTP, [126]. Such a justification is original to our work, and notably answers the question of the formalisation decision for anyone who might be interested. We have also presented the only work on semantics for strong mobility that we could find, [131].

Indeed, strong mobility does not pose great theoretical difficulties, but rather technical difficulties. For example, Sangiorgi & Walker [112] estimate that the introduction of interrupts to HOpi may significantly change its semantics. Another difficulty is that control must be represented explicitly. Whilst there is much literature on control flow using functional semantics, almost none exists in the relational semantics.

As hinted above, we use a slightly modified distinction between weak and strong mobility unlike what is traditionally used in the literature: the first is equivalent to code mobility; and the second involves of necessity the use of an interrupt mechanism.

Many works on channel mobility have been presented. Of those based on denotational semantics, the works in [58], [132], and [67] are similar in the way that channels are passed around, and in their treatment of dynamic interfaces.

In particular, [67] allows communications of the form $x!(x!)$;[132] uses a closed model in the sense of Roscoe, where the channels that may be used are all known in advance from the outset, although each process may only hold a disjoint subset of it at a time; and [58] uses channels in quite a specific way that consists of outputting, from a set of private names, the output end of a channel to the environment, whilst inputting the input end to the process itself. This operation is a little awkward at first, and is reminiscent of Roscoe's unified model for the pi-calculus, [108].

The work on CSP-like operators [107] has introduced, in our point of view, a superfluous linearisation procedure; yet, it has led us to find earlier work, notably [120], and [21]. These latter works would permit, in our opinion, recasting nicely the work initiated in [107]. They have also provided us with some ground for discussing the notion of axiomatisation introduced above.

In [108], it is argued that the pi-calculus is CSP-like, and a few models are proposed, based on a generalised relabelling operator, whose effect is either (one-to-many) relabelling of a single name, or making some other names silent. Unfortunately, a great deal of effort has been spent on a notion of name that is too general, and notably includes events that do not relate to communications viz. channels. We find it harmful to move events as does Roscoe, because then, processes may be moved as well. In effect, even if one were to object that the semantics in [108] do not directly enforce this, we keep in mind that events are often abstractions. Hence, suppose that they are meant to represent some function, or a process/procedure hence, moving them around is hazardous.

The generalised relabelling mechanism defined by Roscoe, as well as the location/relocation mechanism defined in [18] may serve to confirm our hypothesis made earlier that the main difference between the pi-calculus and CSP-like process calculi lies in the calculus of names itself, not forcefully in channel mobility. Our proposed simplification of the semantics in [138] illustrates this; it further illustrates a channel-naming procedure that has all the appearances of Roscoe's standardised naming model, and of Bialkiewicz's localised naming model.

Finally, the literature review has prompted us to investigate the question of the relation between CSP and the pi-calculus, although this was not our original objective. However, it is a haunting question given the prominent place occupied by the pi-calculus as a channel mobility formalism.

Based on the de Simone algebraic framework [120], we have suggested that CCS and the pi-calculus have the same axiomatic specification, to the exception of axioms **A2**, and **B2** on one hand, and **A3** and **B3**. The difference between **A2** and **B2** shows this: there is no obligation that because a channel name appears as the subject or message of a communication, such a name must be used as the object of communications in the receiving process. Indeed, it is necessary that the received name be *recognised* as a channel. So the difference between **A2** and **B2** reflects mainly that point, and is not in itself necessary, though it poses the question of what makes such a recognition possible.

The answer to the latter question is provided by axiom **B3**, namely that the received name must be added into the interface of the receiving process, since all such names make valid communications. **A3** is just a (stronger) variant of **B3** because it assumes that in a process expression, every channel belongs to the interface. In Chapter 4 of this thesis, we discuss the implementation of axiom **B3** in the context of UTP-CSP.

Whilst it was not our primary objective to use de Simone algebraic framework, we notice the similitude between the representation of a process calculus as a triple $(M, F, Spec)$ and the UTP representation of theory as a triple (A, Σ, HC). de Simone framework defines operational semantics as a starting point, whereas UTP uses denotational semantics instead. We refer to Hoare and He' remark ([66, Chap. 10, last paragraph]) that any framework based on operational semantics is unsuitable for a unification of theories. A contrario, we may suppose a transposition of de Simone framework in UTP. Then the remarks made earlier concerning the axiomatisation of process calculi translates all the same to UTP. Remark also that both operational semantics and denotational semantics are given as functions: the one in a set-

theoretical form, as a set of rules, and the second in a functional form, as a monotonic and idempotent function. In this thesis we do not pursue the relation between operational and denotational semantics for mobility —an interesting discussion of such a relation in context of UTP may be found in [66]—, and leave it for future work.

Chapter 4

Channel Mobility

4.1 Introduction

Consider the following examples.

Example 4.1.1. *Three students each having a laptop decide to meet in a study room for studying. Student A happens to have in his computer some data of interest to both students B and C. However, there is only one Ethernet cable available, hence only two computers may be connected at a time. One end of the cable (output end) is plugged into A's computer, and the other end (input end) is plugged into B and C. The order in which the input end is plugged (i.e. B's before or after C's computer) needs not to concern us at present. The system composed of the pairs student-computer and the Ethernet cable constitutes a mobile system with channel mobility. The Ethernet cable is moved from B to C, or inversely. Note that in the present case, it is a human agent that is responsible for the mobility of the cable, achieved (implemented) by the* unplug/plug *operation.* □

Example 4.1.2. *Three students A, B, and C, meet at the occasion of some conference, yet they do not all speak the same languages. A can speak both languages say L1 and L2, B can speak only L1, and C can speak only L2. B and C desire to communicate with each other, so they ask A to do the translation between them. However, A would like to quit the translator role, so A teaches the language L2 to B instead. The system composed of pairs student-languages constitute a mobile system with channel mobility. The language L2 is moved (or passed) from A to B, although (and unlike (e.g.4.1.1)), A does not lose the ability of using L2. Again, it is a human agent that is responsible for the mobility of the language, achieved (implemented) by the* teach/learn *operation.* □

Example 4.1.3. *Consider a Mobile Telecommunications Network, which is a system composed of two parts: a user part and an operator part (or core network). Two users may not communicate together directly: they have to establish a radio link with the operator part first, as follows. The core network has two main types of components: an access station (AS), which covers a given physical area, hence the users therein, and a control station (CS), which manages a given number of access stations. Consider a user A who wants to communicate*

with another user B. Then, A first needs to establish a radio link with an access station, say $AS_{1.1}$*, that is managed by (and hence connected to) the control station* CS_1 *(via either a cable or a radio link). Similarly for B. We call by the common name* node *user, access station, and control station; and any link between two nodes, say* n_1 *and* n_2*, we denote by* $n_1 \leftrightsquigarrow n_2$*. Then,* $A \leftrightsquigarrow B = A \leftrightsquigarrow AS_{1.1} \leftrightsquigarrow CS_1 \leftrightsquigarrow CS_2 \leftrightsquigarrow AS_{2.1} \leftrightsquigarrow B$*. During a communication, it is possible for a user, A say, to change of coverage area, i.e. A can move from* $AS_{1.1}$ *to a given* AS_x*. When a change of access station (coverage area) occurs, the control station, here* CS_1*, releases the radio link between the user and the current AS, here* $A \leftrightsquigarrow AS_{1.1}$*, and establishes a link between the user and the next AS, here* $A \leftrightsquigarrow AS_x$*: this operation is called the* handover *(or* handoff*) procedure. The system composed of a single user and the core network constitutes a mobile system with channel mobility. The radio link user*$\leftrightsquigarrow$*core network is moved from* $AS_{1.1}$ *to* AS_x*. In this case, it is an electronic device (or electronic agent) that is responsible for the mobility of the radio link, achieved (implemented) by the* handover *procedure.* □

Example 4.1.4. *Consider the Internet, which is a system composed of, grossly, two parts: a subscriber part and a service provider (ISP) part. We assume that subscribers may communicate only through the network of ISPs; two ISPs may not be connected together directly either, but each one must be linked to a router. By analogy to (e.g.4.1.3) above, and only with regard to (some hierarchy of) communication links, an ISP may be viewed as an access station (AS), and a router as a control station (CS). Any computer that subscribes to an ISP is given an IP (Internet Protocol) address that determines the IP link between the subscriber and its ISP. At different times, the same IP link may connect a different subscriber to a given ISP. For illustration, let A and B be potential subscribers, and* ISP_1 *a given service provider. Initially, A is subscribed to* ISP_1*, with IP address (or link) say @ip1. At some point, A unsubscribes from* ISP_1*, and afterwards, B subscribes to* ISP_1 *and is given @ip1 (the same of A before). The system composed of subscribers and a single ISP constitutes a mobile system with channel mobility. The IP address is moved from one subscriber to another, here from A to B. It is a computer agent that is responsible for the mobility of the IP address, achieved (implemented) by the* subscribe/unsubscribe *operation.* □

Each of the situations given above describes what we have termed a "mobile system with channel mobility", i.e. a system in which communication links are moved between the components that the links permit to connect. We also use the term *dynamic network* system.

Milner's informal characterisation of mobile systems [86] —*links* move in a virtual space of linked processes— permits reasoning about every possible kind of system, including physical systems and computer systems. Looking at (e.g.4.1.1) above, we may consider as a single process the pair of agents (student, laptop). The link that moves is the Ethernet cable, and the human agent (viz. the student) is only considered w.r.t. the role or action of moving the cable. The medium for communicating the cable here would be the physical space, or equivalently, the air medium. (e.g.4.1.1) is an example of a physical system. In (e.g.4.1.4), not the physical entities matter, but rather their computation. That is why IP addresses determine the links,

and neither cables nor radio links. (e.g.4.1.4) is an example of a computer system.

Any (eventual) formal characterisation of channel mobility may be discussed directly in terms of existing formal models that describe *static network* systems, in which the links between components do never change.

In the next section, we discuss the formalisation of dynamic network systems in UTP on the basis of the underlying model for UTP-CSP processes (cf. Chap. 2), which is static. Each concept (of dynamic network systems) is first introduced informally, then we discuss the possible ways of representing the concept formally using UTP.

In Section 4.3 we present the semantics of mobile processes. The highlight there is the definition of the channel-passing mechanism. Dynamic hiding and dynamic renaming are also defined there.

Section 4.5 contains a description of the links between static and mobility theory.

Section 4.5.5 contains as an example a buffer with mobile channels, and its transformation into an equivalent buffer with static channels (and other examples).

Finally, in Section 4.6 we review our results and their implications and conclude this chapter hence.

4.2 Dynamic (Network) Systems - Concepts and their Formalisation

4.2.1 Some definitions

A mobile system with channel mobility, or dynamic network system, is a system in which the links between components may change during the system's activation. Such systems contrast with static systems i.e. systems in which the links between components do not change once defined.

A *network* defines a system of communicating components, hence we may appropriately say that static systems define static networks, whilst mobile systems define dynamic networks.

The *configuration* or *topology* of a network defines the links that exist between the network's components, so we also say that static systems have a static topology, whilst dynamic systems have a dynamic topology. To say that the links between components may change (or not) is thus equivalent to saying that the (network) topology formed by these components may change (or not).

We may conceive of a mobile system as having two parts: a static system, defining some initial topology, and some functionality for changing said topology. Yet another conception of a mobile system would be as an ordered set of static topologies, where the order defines what topology may be observed at what time. Hence, any characterisation of mobile systems must also permit a characterisation of static systems.

The basic model for UTP-CSP processes permits us to represent and reason about static systems only: we will refer to it as static UTP-CSP or equivalently as static CSP. We will

discuss the UTP-characterisation of channel mobility on the basis of static UTP-CSP: the resulting theory will be called mobile UTP-CSP or equivalently mobile CSP. Hence, in what follows, we will formalise the statement *the topology of the network/system has changed*, for any UTP-CSP process.

Topology, Network. The notion of *topology* is a *graphical notion* in the sense that it is associated to some graphical representation of processes and their links. We are not interested in a visual formalism although we will often use some graphical representations when necessary. References to related works on visual formalisms are [85], [74], [98], and [97]. It is also *global* in the sense that it supposes a view of all the processes that compose the system/network considered.

In UTP-CSP, at first, it would seem that the topology of a system corresponds to the *interface* of the process representing that system. However, such is not 'completely' the case. Recall that the interface of a (UTP-CSP) process characterises its links to the environment, whereas it would be more accurate to state that the topology of the same process characterises internal links, not visible by the environment. This suggests a distinction between a notion of *internal interface*, which would define channels not used for communications with the environment, and a notion of *external interface*, which would define channels that may be used for communications with the environment, exclusively.

The interface of UTP-CSP processes corresponds to the concept of external interface. The view is 'local' in that it is the point of view of network components, and accords with the concept of *compositionality* i.e. the fact of constructing systems as a composition of system components. The topology would then correspond to what is obtained at the end of such a composition.

From what precedes we make the following initial assumption. [**Assume:mc:iface**]

> There are some UTP-CSP processes that have no internal interface (viz. no internal communications), and an external interface only: we call them *level-0* processes.
> There are other UTP-CSP processes composed exclusively of the first kind: *level-1* processes. Their internal interface is the union of the interfaces of their component, and is disjoint from their external interface.
> There is a third kind of components, which may be composed from at least two *level-1* processes. Their internal interface is calculated as before, and must be disjoint from their external interface.

Granting the previous assumption, when discussing channel mobility, we may consider to be hidden the internal interfaces of components, and manipulate their external interfaces only. Formally, we are simply redoing what was already done in static UTP-CSP with the hiding operator. This choice is a sensible one as it implies that the internal interface of a system entirely determines what channels its components may move. Reciprocally, this means that a system may never receive from its environment channels that are declared

silent. For illustration, it would mean that for mobile systems, it is not always the case that $(P \setminus X) \parallel Q = (P \parallel Q) \setminus X$ when $X \notin \alpha Q$. The choice of a better representation of the parallel operator may make discussions a little easier, say by adopting an alphabetised version instead (see [106]). Then, we only need to write $P \underset{Y}{\parallel} Q$ to specify the external interface Y (external for both P and Q, but internal for $P \parallel Q$).

Location. The preceding discussion shows that some confusion arises when the notion of *location* is not clearly defined. In effect, we expect that a channel moves from one location to the other.

Let us consider a process P. To characterise that P may use a channel ch for its communications, we say that ch is in the alphabet of P i.e. $ch \in \alpha P$. (We also say that ch belongs to the interface of P.) In other words, the observation that a process P 'owns' a channel ch or equivalently, that a channel ch is 'located' in a process P, is characterised by the presence of the channel within the alphabet/interface of the process. This may seem obvious, but it has great implications.

It means that the presence in (resp. absence from) the alphabet characterises mobility: a channel is moved out if it was in the alphabet, and is no more; conversely, it is moved in if it was not in the alphabet, and now is. This gives us the characterisation of channel mobility for *level-0* processes. The latter is enough also for characterising the internal mobility of *level-1* processes. See that we may drop the distinction between internal and external interface for *level-1* (and higher-level) processes altogether: we simply distribute that interface amongst the component *level-0* processes as follows.

Let X denote the partition of the interface of a *level-1* process P, corresponding to its internal interface, and let Y denote the second partition corresponding to its external interface. Let X_i and Y_i the corresponding partitions for each (*level-0* process) component P_i of P: $P = \left\|_i P_i\right.$. Then $X = \bigcup_i X_i$ and $Y = \bigcup_i Y_i$. We simply iterate this flattening procedure for higher-level processes.

Note that we have now stated more simply what was stated earlier regarding the fact that hiding had no effect on mobility (cf. Chap. 3, §3.4). Alphabetised parallel shows clearly that when two processes are run in parallel, what matters is on which channel they may synchronise, characterised by their so called external interface, denoted by Y above. How P and Q have acquired said channels, if they have actually acquired them, and if they effectively use them (either by communicating through them or by communicating/moving them) is of no concern to the parallel operator. Similarly for hiding. If we write $(P \setminus X) \underset{Y}{\parallel} Q$, then we expect that Y and X are disjoint. Again, hiding does not care about how channels are acquired, but only about hiding any that may be used if and when it is used. We say that those operators have no mobility effect: they do not engender channel mobility, and do not affect it; although, channel mobility may affect them, eventually by increasing the sets Y and X, respectively.

In sum, in order to characterise channel mobility, we model the location of channels by the interface of processes. There is no need for a distinction between internal and external interface. As a consequence, there is no need for distinguishing a closed-world semantics for mobility, as Roscoe [109, §20.3] does. The topology of a process would be better captured with regard to some graphical representation of process networks. In this work, however, we will use such graphics rather informally. In terms of UTP-CSP, the topology of a process may be inferred syntactically from the structure of the process.

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 may actually use that channel for its communications. Associating the interface of a process with the location of channels implies that the interface of mobile systems must be a variable (the interface changes with the movement of channels). Let us talk of either static interface or dynamic interface, accordingly.

The dynamic nature of the interface of mobile systems reflects the statement *a (mobile) process can input a new channel*. By 'new' channel is meant a channel that the process did not 'own' already i.e. a channel that was not in the process's interface during the last observation (keeping in mind that the interface is now a variable). That a process may receive a new channel does not say where that channel comes from. This contrasts with static systems for which all the channels that are owned are defined for once: in the interface. We find it more convenient to define the set of channels that may be 'moved', instead of leaving it undefined — we call that set the *capability* of a process. We justify such a choice through the following illustration.

Consider a router, a device used in the Internet. A router may receive new IP addresses, and delete existing ones, in a non-deterministic way. The capability of a router corresponds to the range of every possible IP address, and its interface at a given (observation) time corresponds to the list of IP addresses in its routing table at that time. The capability of a router may not change for the lifetime of the router, unlike its interface. Also, a router may never input, say radio links.

Hence, the notion of capability not only restricts in a definitive way the range of channels that a process may move, but also their type. And as far as we can see, there is no need for defining a 'dynamic' capability. In effect, consider the router once more. Then, we may say that the router's capability is fixed by the manufacturer once and for all, say *fixed at creation time*. That is, the non-extensibility of the router's capability is defined axiomatically. Prosaically, we may say that it is the *nature* of the router. Now, even if one were to imagine some device whose capability may be extended, then, such an extensibility would also be axiomatic. And, if we regard the relation capability-interface as denoting *the capability of changing the interface*, then by making the capability itself dynamic, we would also be stating something like *the capability of changing the capability*. As if we wanted to state that the capability of a router of moving IP addresses may be extended with the capability of moving

radio links also. However, see that this simply means *the capability of moving IP addresses, until capability extension, and then the capability of moving radio links also.* This capability extension translates into interface dynamics as follows: *the capability of changing the interface by moving IP addresses only, and then, the capability of changing the interface by moving radio links also.*

In more mathematical terms, the capability of the interface determines the greatest value possible for the interface. The capability of the capability would hence determine the maximal value of the capability of a process, and similarly for the capability of the capability of the capability and so forth. So that in the end, it is only necessary to consider the capability itself to be its maximal value already.

The notion of capability brings forth two further analogies. The first analogy is with the mathematical concept of the *limit of a function.* It is often necessary to prove that such a limit exists, although such a proof does not give the actual value of the limit. It is a proof of existence, which is not necessary in this instance because the existence of the capability (the channels therein) is given axiomatically.

The second analogy is with the concept of *human knowledge.* A human being, at a time taken for the origin of time may have some initial knowledge (including the empty one). As he acquires more knowledge in time, he comes to *realise* that he has the capability of acquiring new knowledge. Similarly, he often loses some knowledge acquired previously, so he (realises that he) has the capability of losing or forgetting already acquired knowledge. However, he may never increase nor decrease his capability of knowing. Pragmatically, because such a capability is not in his power: the existence of such a capability was revealed to him through experience; he did not create it himself. Logically, because if he could lose that capability, then he should never realise the loss (otherwise there would be a contradiction), and hence there is no loss. On the other hand, if he can increase his capability, this would mean the capability of knowing a new capability. However, the latter capability of knowing is already included in the original one, so that the capability of knowledge is not only invariant or unchangeable or constant, but is also its own maximal value.

The following definition summarises the previous discussion.

Definition 4.2.1 (Capability)**.** *The* capability *of a process models the statement* a process knows the existence of channels*. Such a knowledge does not confer ownership, as does the interface of a process.*
Also, the capability of a process is its own maximal value, and it may not change, unlike the interface of a process, which may change.
As a consequence, it may not be possible to define a static process in the sense of one whose interface may never change, but rather as one whose interface is forbidden to change. Formally, these are equivalent notions. □

Summary. In the discussions above we have presented some concepts of mobile systems, and how they can be modelled in UTP-CSP. Starting with the notion of topology, we have stated that it can be obtained from the syntactic structure of the process definitions and their interface. We have used the interface for modelling the location of channels, such that a change of topology is modelled by a corresponding change of interface. In particular, we have shown that a change of topology at the network level can be captured at the level of individual network components, by the change of each component's interface. We have shown that channel mobility introduces a notion of capability that is semantically different from the notion of interface. We have described that difference mathematically by saying that a capability is the maximal value of the interface, and that that value is constant. Informally, the capability denotes the fact that channels may be moved, whilst the interface denotes that channels may be used. Whilst the discussions above had a more philosophical character, the ones that follow are more technical. Every concept that we will introduce is immediately followed by its formal characterisation, unlike above. The mention of some concepts that were already introduced earlier may seem redundant, but this is out of necessity.

4.2.2 Formalisation

Process networks whose configuration/topology may change throughout their activation are called dynamic networks or mobile systems. *Channel mobility* is the name of the corresponding paradigm. The observation of a dynamic system may be divided according to its different topologies.

Definition 4.2.2 (Snapshot)**.** *A* snapshot *is a (maximal) period of static network topology for a mobile system, and determines the behaviour of the system during that period. Hence, any system must have at least two such snapshots to be considered mobile. The overall behaviour of the system may be obtained by a given concatenation of all the snapshots of the process in their order. In contrast, there is only one such snapshot for any non-mobile/static process.* □

It should be pointed out here that the choice of having a snapshot to be maximal is deliberate, and is not in itself a constraint. Such a choice is motivated here by the fact that it makes it easier for reasoning about mobility. The notion of snapshot is unnecessary for static network systems since their topology is fixed; it must also not be confused with similar notions in the literature, e.g. it is not equivalent to the *state* of a process, and neither to the *dump* of a store.

Processes are connected via links or channels through which they may communicate. Consider a network of three processes P, Q and R connected as shown by *figure4.1(left)*. Another possible topology for such a network may be obtained by removing the link ch_1 between P and Q and using it to connect P and R instead, as shown by *figure4.1(right)*. Note that it is not properly the channel that moves, but its ends. Hence it would be more appropriate to talk of the mobility of channel ends, and this is what should be understood in

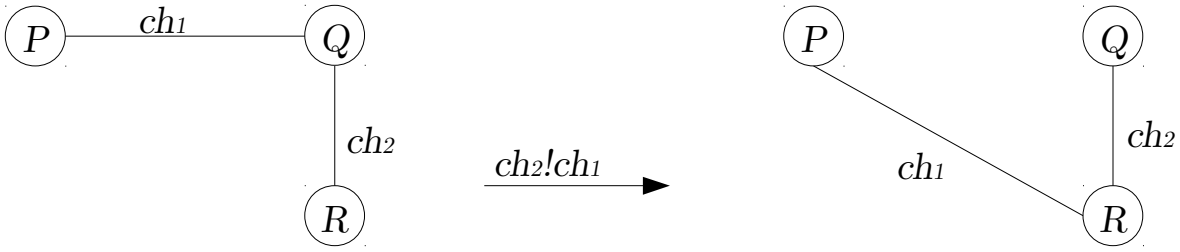

Figure 4.1: Channel Mobility with 3 processes. (left) Before the migration of ch_1. (right) After the migration of ch_1.

the subsequent paragraphs.

In this section we present the necessary changes to the static model of CSP (cf. Chap. 2.5.1) that enable us to give semantics to channel mobility. The mobility model has three main characteristics:

- channels are 'localised' in alphabets or more precisely, in interfaces. Hence the mobility of a channel is from one interface to another;
- channels may be communicated as messages amongst processes (hence we need a new representation for channels);
- the interface of processes may change as a consequence of channel mobility.

Channel names. In the static model of CSP processes, channel names are just *logical identifiers.*[1] For channel mobility channels must rather be modelled explicitly, as data elements: they will also be represented by channel names. How this new set of names relates to the one from the static model, i.e. the interface $\mathcal{I} = \{ch \mid \exists\, e \bullet ch.e \in \mathcal{A}\}$, is shown in subsequent paragraphs.

Let *Chans* denote the set of channels that a process may use for its communications. The names in the set *Chans* represent actual objects or entities, similar to natural numbers. All such names must also belong to the interface of the process as defined in the static model i.e. $Chans = \mathcal{I}$, for every process.

In order to bring as little change as possible to the static model, and to keep reasoning about static and dynamic aspects of a process's behaviour separate, we will maintain elements from the static model whenever possible, and add new elements specifically for mobility.

Mobile channels. For the purpose of a static-dynamic dichotomy, *Chans* will contain static channels only. So we define a set of mobile channels only, denoted by *MCh*. The two sets must be disjoint: $Chans \cap MCh = \{\}$. This will notably ensure that channels in *Chans* may not be moved.

[1] 'logical' in the sense that a channel name $ch \in \mathcal{I}$ represents/models a logical concept viz. 'the occurrence of a communication on the channel named *ch*', and not the channel itself.

Channel mobility works with the assumption that a process may receive new channels, i.e. channels that the process did not previously own. We will denote by $mChans : \mathbb{P}\, MCh$ the variable that may contain such channels when they have been acquired.

Definition 4.2.3 (*MCh*, *mChans*)**.** *We assume a set of mobile channels, denoted by MCh. MCh denotes the* capability *of a process. Then:*
$mChans, mChans' : \mathbb{P}\, MCh$, 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.* □

Definition 4.2.4 (Ownership)**.** *A process* owns *a mobile channel mc if and only if $mc \in mChans$ at the time of the observation.* □

Events of mobile channels. In the static model the set *Chans* is not represented. The interface of a process may be obtained only from its actions set $\mathcal{A}$. 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 4.2.5 (*MCev*)**.** *Let MCev denote the set of events obtained from MCh.*

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

Dynamic alphabetised traces. Let *mtr* denote the trace associated with 'acquired' mobile channels viz. 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: *mtr* must reflect such changes.

Whilst in the static model the type of *tr*, $\mathcal{A}^*$, guarantees that only the actions that are in $\mathcal{A}$ may be recorded, to provide the same guarantee in the context of channel mobility by adopting the typing approach of the static model would require that the type of *mtr* changes whenever *mChans* takes a new value. This is a problem of dynamic typing that may be solved as follows.

First, we recall that the 'type' of a variable determines the values that the variable may take. For the type to change over time simply means that the corresponding set of possible values changes over time. Hence, dynamic typing may be modelled by employing a static type defining all possible values (a sort of default set) and then placing restrictions on that default set where necessary within the process's definition.

In our case, we may define a static type for *mtr*, $(MCev)^*$. This would mean that any event in *MCev* may be recorded, which is too large. We now need to enforce the condition that only the events associated with channels that have already been acquired (i.e. in *mChans*) may be recorded. For that purpose, we need to keep the history of successive interfaces i.e. the history of the value of *mChans*. We could then ensure that at a given time, any

event recorded in *mtr* belongs to the set-value of *mChans* at that time. That is, at a given observation time k, we must record both the value of *mChans*, say $mChans_k$, together with the event, say e, and ensure that $e \in mChans_k$. We thus introduce the notion of *dynamic alphabetised trace*.

In any snapshot, the value of *mChans* is fixed, and differs between any two consecutive snapshots. If we associate the value of *mChans* within a single snapshot with a valid trace for that snapshot, we obtain an *alphabetised trace*. And if we combine the alphabetised traces of every snapshot into a single trace, we obtain a *dynamic alphabetised trace*. For simplicity however, we rather associate the occurrence of every event with the valid dynamic interface at the time of the observation —this permits us to disregard snapshots.

Definition 4.2.6 (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.* □

The null event will be denoted by *nil*. For DATs it is more convenient than an empty space. In particular, there may be many events of the form (s, nil). Every (non-empty) trace must contain at least one such event.

In our construction so far, *tr* and *mtr* have been considered as non-alphabetised. In particular, we have introduced *tr* only w.r.t. the static CSP model. In what follows, we should work with the alphabetised versions only. For a mobile process, the overall trace will be denoted by *dtr*.[2] It should contain elements from both *str* and *mtr*, where *str* denotes the (alphabetised) trace relating to static channels exclusively. Their respective relation to *dtr* is obvious:[3] $str = dtr \upharpoonright \mathcal{A}$ and $mtr = dtr \upharpoonright MCev$, but we shall keep using them informally for the sake of conciseness.

Definition 4.2.7 (Trace of a mobile process)**.** *Let* Σ *denote the actions set for mobile processes, then* $\Sigma \mathrel{\widehat{=}} \{nil\} \cup \mathcal{A} \cup MCev$.
$dtr, dtr' : (\mathbb{P}(Chans \cup MCh) \times \Sigma)^*$*, is the dynamic alphabetised trace of mobile processes.* □

Remark: as already stated, the set *MCh* (viz. *MCev*) models the concept of a *capability* introduced earlier. Looking at the type of *dtr*, we may now see more clearly the benefits from separating *ownership* from *existence* conferred by the concept of capability. If indeed *MCh* also meant ownership as $\mathcal{A}$ does, the typing justification propounded earlier would not have sufficed to establish the correctness of our semantics. Indeed, we would have been faced with a paradox: in the absence of the concept of capability, ownership cannot be dynamic. We discuss this paradox in greater detail in Section 4.6.

We use the following two projections to select each component of an element in a DAT trace: $\pi_1(s, e) = s,\ \pi_2(s, e) = e$. We may then override them to get also the first and second

[2] The variable name *dtr* is used here mainly for readability, to keep separate static and mobile CSP theories. The name dichotomy is *not* essential.

[3] $s \upharpoonright Y$ denotes the sequence s restricted to elements from the set Y.

component of all elements in a trace, respectively. Let $k \in \{1, 2\}$, then

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

We now give a more formal characterisation of the notion of snapshot defined earlier. We say that a process has a static network (or fixed network topology) when its interface is the same whatever the elements of its DAT. Formally:[4]

Definition 4.2.8 (*SN*).

$$\textbf{\textit{SN}} \quad P = P \wedge \left(\begin{array}{l} \forall\, (s_1, e_1), (s_2, e_2) : \mathbb{P}(Chans \cup MCh) \times \Sigma \mid \#\ mtr' \geq 2 \bullet \\ (s_1, e_1) \frown (s_2, e_2) \in mtr' \Rightarrow s_1 = s_2 \end{array} \right) \qquad \square$$

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 separate interfaces. Formally:

Definition 4.2.9 (*DN*).

$$\textbf{\textit{DN}} \quad P = P \wedge \left(\begin{array}{l} \exists\, (s_1, e_1), (s_2, e_2) : \mathbb{P}(Chans \cup MCh) \times \Sigma \bullet \\ (s_1, e_1) \frown (s_2, e_2) \in mtr' \Rightarrow s_1 - s_2 \neq \{\} \end{array} \right) \qquad \square$$

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

Definition 4.2.10 (*MC1*).

$$\textbf{\textit{MC1}} \quad P \;=\; P \wedge \forall\, s : \mathbb{P}(Chans \cup MCh), e : \Sigma \bullet (s, e) \in dtr' \Rightarrow e \in s \qquad \square$$

Literally, ***MC1*** states that every event e that is recorded must belong to the dynamic interface s (the associated events alphabet) valid at the time of the recording.

DATs lead us to reconsider the healthiness condition ***R2***. In effect, ***R2*** is meant to hold for the events history only, not for other types of history. The application of ***R2*** to mobile processes is called ***R2M***, given below.

Definition 4.2.11 (*R2M*).

$$\textbf{\textit{R2M}} \quad P \;=\; \left(\begin{array}{l} \bigsqcap \{ P[t,\ (t \frown (dtr' - dtr))\ /\ dtr, dtr')] \mid t \in (\mathbb{P}(Chans \cup MCh) \times \Sigma)^* \wedge \\ \pi_1(t) = \pi_1(dtr) \} \end{array} \right) \qquad \square$$

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

Literally, ***R2M*** allows replacing the initial history of events without changing their related interface. A case analysis would illustrate how the substitution works.

- The easier case is when the trace has a single element, i.e. $dtr = \langle (a, b) \rangle$. Then, any substitute trace $t = \langle (x, y) \rangle$ must be such that $x = a \ \wedge \ y \in \Sigma$.
- For $dtr = \langle (a, b), (a, c) \rangle$, $t = \langle (x, y), (x, z) \rangle$ where $x = a \ \wedge \ y, z \in \Sigma$.
- The third case is when successive interfaces are distinct, reflecting that channel mobility has occurred: For $dtr = \langle (a1, b), (a2, c) \rangle$, $t = \langle (x1, y), (x2, z) \rangle$ where $x1 = a1 \wedge x2 = a2 \ \wedge \ y, z \in \Sigma$.

The definition of ***R2M*** clearly forbids the substitution of the initial interface history by an arbitrary one and rather conserves the interface history. Hence, the question may be raised of the possibility of changing the initial interface history.

Trivially, whilst the substitution itself is possible, the result of the substitution would not yield a fixed point. Indeed, a change of interface implies that the resulting trace would be different from the original one. Thus P and $P[t, \ (t \frown (dtr' - dtr)) \ / \ dtr, dtr')]$ are *not* equal for just any element $t \in (\mathbb{P}(Chans \cup MCh) \times \Sigma)^*$. The substitution does not yield fixed points for any process that is ***SN*** healthy.

For ***DN*** healthy processes, we may conceive of a process with an arbitrary initial interface. However, substitution and nondeterminism are certainly not the best way for specifying such a process. We may better discuss the consequences of substituting the initial interface by an arbitrary one in a ***DN*** healthy process by means of an illustration.
Let $\mathcal{I}1$ be the initial interface of a process, and suppose that we know the interface $\mathcal{I}2$ of its next snapshot. Let $\mathcal{J}$ be an interface and the substitute for $\mathcal{I}1$.

- For $\mathcal{I}1 = \{ch1, ch2\}$, $\mathcal{I}2 = \{ch2\}$, and $\mathcal{J} = \mathcal{I}2$, the substitution is clearly undesirable since it denies the movement of $ch1$. If instead $\mathcal{J} = \{\}$, then the substitution supposes the mobility of $\{ch2\}$ which would contradict the definition.
- Now let $\mathcal{I}1 = \{ch1\}$, $\mathcal{I}2 = \{ch1, ch2\}$, and $\mathcal{J} = \mathcal{I}2$. Again, the substitution cancels the movement of $ch1$.

In sum, any substitution of the initial interface history for a distinct interface *may have* a mobility effect, which is unhealthy.

In line with the previous discussion, 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 $\langle (a1, b), (a2, c) \rangle$, where $a1 \neq a2$, $\langle (a1, b) \rangle$ would belong to the first snapshot and $\langle (a2, c) \rangle$ to the second. *Any two consecutive snapshots may not possibly have disjoint interfaces, except when either interface is empty.*

In a single step, it is only possible to release the whole of the actual interface at once, but not to acquire a new channel at the same time. This is true if the interface is unique

for every process in a parallel composition. It must remain true in the case where part of (or the whole) interface is shared amongst parallel processes since it is inconceivable that any process should be able to use a channel after it has been moved out, even if by another process. Interleaving of traces in the semantics of the parallel composition operator further implies the interleaving of interfaces. That any two consecutive interfaces may not be disjoint (unless at least one is empty) is expressed by the following healthiness condition:

Definition 4.2.12 (***MC2***)**.**

$$\textbf{\textit{MC2}} \quad P = P \wedge \left(\begin{array}{l} \forall\, (s_1, e_1), (s_2, e_2) : \mathbb{P}(\mathit{Chans} \cup \mathit{MCh}) \times \Sigma \mid \# \ \mathit{mtr}' \geq 2 \bullet \\ (s_1, e_1) \frown (s_2, e_2) \in \mathit{mtr}' \Rightarrow (s_1 \subseteq s_2 \ \vee \ s_2 \subseteq s_1) \end{array} \right) \qquad \square$$

Example 4.2.13. *The interface history*

$$\langle \{ch_1, ch_2\}, \{ch_2, ch_3, ch_4\} \rangle$$

is forbidden by ***MC2****; however,*

$$\langle \{ch_1, ch_2\}, \{ch_2\}, \{ch_2, ch_3, ch_4\} \rangle$$

is a valid interface history. $\square$

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.

Refusals. In the basic model, $\mathit{ref} : \mathbb{P}\,\mathcal{A}$ contains events in which a process may refuse to engage, although they are authorised. The type of ref, $\mathbb{P}\,\mathcal{A}$, guarantees that a process may only refuse authorised events i.e. *a process cannot refuse events that it does not own.* We will denote by *dref* the refusals set for mobile processes, and by *sref* and *mref* its static and mobility components. With channel mobility, the type of *mref* would need to follow the changes of the dynamic interface, so we would once again face a problem of dynamic typing. As earlier, we may solve the problem by considering a static type for *mref*, and then impose a restriction on the events that may be refused, by means of a healthiness condition. The static type for *mref* will be $\mathbb{P}\,\mathit{MCev}$. The healthiness condition expressing that only owned events may be refused is given below.

Definition 4.2.14. ***MC3*** $P \ = \ P \wedge \mathit{dref}' \subseteq (\mathcal{A} \cup \mathit{mChans}')$ $\square$

Note that mChans' is used above for economy of notation, to denote the corresponding set of events.

Summary. We have presented in this section the fundamental concepts of channel mobility and how they may be formalised in UTP. The formalisation has been based on static UTP-CSP and shows quite clearly, if this was not clear enough, that channel mobility is altogether

a new paradigm. Three new healthiness conditions have been introduced, and a new traces model has been defined which aggregates the interfaces history to the original events history. We have chosen to pair together the elements of both histories the consequence of which is the introduction of a null event *nil*. Another approach is possible where the elements from each history are recorded separately. The latter approach is the one adopted by Grosu & Stolen [58], and Hoare & O'Hearn [67]. In particular, this second model does not require introducing a particular *nil* event. The choice of either model is likely a matter of taste. We think that the two approaches yield equivalent traces models but this conjecture needs further investigation.

4.3 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. In Section 4.2 we have introduced alphabet elements as well as some healthiness conditions. In this section we put them together to define what a mobile process is. The operators are defined afterwards. The highlight of this section is the semantics of the operation that may change the interface of a process during its activation.

The following definition summarizes the previous discussions.

Definition 4.3.1 (Mobile processes)**.** *A mobile process is one that satisfies the healthiness conditions* ***R1****,* ***R2M****,* ***R3****,* ***CSP1****,* ***CSP2****,* ***MC1****,* ***MC2****, and* ***MC3****, and has an alphabet consisting of the following:*

- $\mathcal{A}$*, the set of static events in which it can potentially engage; the events in this set may not be moved.*
- *Chans, the set of static channels (viz. they not be moved), whose events are in* $\mathcal{A}$*.*
- *MCh, the set of mobile channels that can potentially be moved in (acquired) or moved out (released) during activation.*
- *MCev, the set of events whose channels are in MCh; the events in this set may be moved, according to the movement of their corresponding channels.*
- $mChans, mChans' : MCh$*, the* dynamic interface*, also the last element of the interface history.*
- $dtr, dtr' : (\mathbb{P}(Chans \cup MCh) \times \Sigma)^*$*, the trace, (where* $\Sigma = \{nil\} \cup A \cup MCev$*).*
- $dref, dref' : \mathbb{P}\,\Sigma$*, the refusals set.*
- $ok, ok' : \mathbb{B}$*.*
- $wait, wait' : \mathbb{B}$*.*

- v, v'.

For simplicity, we may introduce the following variables that can be calculated from those above:

- $mtr, mtr' : (\mathbb{P}\, MCh \times MCev)^*$, *the partition of the trace* dtr *(resp.* dtr'*) restricted to mobile channels.*
- $mref, mref' : \mathbb{P}\, MCev$, *the subset of the refusals set* $dref$ *(resp.* $dref'$*) restricted to mobile channels.*

□

4.3.1 Healthiness conditions

In this section we present some important results concerning the healthiness conditions introduced thus far. Notably, for each healthiness condition we study its *idempotence*, *closure* with basic operators, and *commutativity* with other healthiness conditions, following the lines of [35].

In [35], Cavalcanti & Woodcock carry out a systematic study of the aforementioned properties for every healthiness condition that they introduce. Adopting the methodology in [35] would be fastidious, however. Some general results provided by Harwood and the previous two authors in [59] may make such a study easier.

In [59], the notion of a *conjunctive healthiness condition* allows generalising a number of important properties about healthiness conditions.

Definition 4.3.2 (Conjunctive healthiness condition [59, Def. 1])**.** *A healthiness condition* $\boldsymbol{CH}$ *is called* conjunctive *if*

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

for some predicate ψ. □

Theorem 4.3.3 (Closure of $\boldsymbol{CH}$ 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* $\boldsymbol{CH}$ *healthy, where* $\boldsymbol{CH}(P) = P \wedge \psi \wedge \psi'$ *(*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.*

Proof. cf. [59, Ths. 1, 2, 3], [66, Chap. 4]. □

In [66, Chap. 4], the following endofunction[5] was defined:

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

where $\mathbf{S}$ is the set of predicates of a (UTP) theory.

Theorem 4.3.4. *and_Q is idempotent.*

Proof. $and_Q \circ and_Q = Q \wedge (Q \wedge X) = Q \wedge X = and_Q$ □

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 $a \wedge b = b \wedge a$.

MC1, ***MC2***, and ***MC3*** are all conjunctive, hence they are all idempotent, pairwise commutative, and $\{\wedge, \vee, \lhd b \rhd, ;, \mu\}$-closed. Furthermore, they are all pairwise commutative with ***R1***, ***R2M***, ***R3***, ***CSP1*** and ***CSP2***. In effect, ***R1*** is conjunctive, so commutativity trivially holds. Substitution in ***R2M***, and sequential composition in ***CSP2*** are also conjunctive. Recall (provided $out\alpha\, P = in\alpha\, Q$):

$$P ; Q = \exists\, v_0 \bullet P(v_0) \wedge Q(v_0) \qquad P[e/x] \mathrel{\widehat{=}} x := e ; P(x) = \exists\, x_0 \bullet x_0 = e \wedge P(x_0)$$

The commutativity with ***R3*** and ***CSP1*** comes from the closure and idempotence of conjunctive healthiness conditions.

We may now define the following healthiness conditions in which the composition order is indifferent.

Definition 4.3.5 (***MC***, ***MC123***)**.**

$$\boldsymbol{MC123} \mathrel{\widehat{=}} \boldsymbol{MC1} \circ \boldsymbol{MC2} \circ \boldsymbol{MC3}$$
$$\boldsymbol{MC} \mathrel{\widehat{=}} \boldsymbol{CSP} \circ \boldsymbol{MC123}$$

□

[5] A function whose domain and range are the same.

4.3.2 Some mobile processes

Assignment. The definition of assignment is similar to the static CSP version, except that it must now be made healthy.

$$(x := e) \;\widehat{=}\; \boldsymbol{MC123} \circ \boldsymbol{R3} \circ \boldsymbol{CSP1}(ok' \wedge \neg\, wait' \wedge x' = e \wedge dtr' = dtr \wedge v' = v)$$

Prefix. In the basic model, the occurrence of an action a corresponds with 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. The value of s may be given by the value of variable *mChans* at the time of the record. 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 $mdo_{\Sigma}(a)$.

Definition 4.3.6 ($mdo_{\Sigma}(a)$)**.**

$$mdo_{\Sigma}(nil) \;\widehat{=}\; dtr := dtr \frown \langle (mChans, nil) \rangle$$

For any event $a \neq nil$*:*

$$mdo_{\Sigma}(a) \;\widehat{=}\; m\Phi(a \notin dref' \lhd wait' \rhd dtr' = dtr \frown \langle (mChans, a) \rangle)$$

where Φ *is defined as in Def. 2.5.11, and* $m\Phi \;\widehat{=}\; \mathbf{MC123} \circ \Phi$*.* □

4.3.3 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 i.e. it must be removed from *mChans*. Clearly, any attempt of moving out a channel, say *oldch*, not already owned must fail: formally, this gives the assumption $(oldch \in mChans)_{\perp}$. Because of **MC3**, the new value of *mChans* must be recorded into the trace i.e. $mdo_{\Sigma}(nil)$, so that any future refusal may not contain the event that has just been removed. This further means that all of the events related to the channel must be removed from *dref* as well, if they were already in *dref*, to avoid chaotic behaviour. The operation for releasing a channel will be called (channel) *s-assignment* and denoted by $:=_s$.

Definition 4.3.7 (Channel s-assignment)**.** *Let oldch be the channel to be released, then:*

$$(\kappa ch :=_s oldch) \;\widehat{=}\; \left(\begin{array}{l} (oldch \in mChans)_\perp \fatsemi \\ \left(\begin{array}{c} \kappa ch \\ mChans \\ dref \end{array}\right) := \left(\begin{array}{l} oldch \\ mChans \setminus \{oldch\} \\ \left(\begin{array}{l} dref \setminus \alpha\, oldch \\ \lhd\, (\alpha\, oldch \in dref) \,\rhd\, dref \end{array}\right) \end{array}\right) \fatsemi \\ mdo_\Sigma(nil) \end{array}\right)$$

To model the situation where the sending process just sends the channel but still retains its value, normal or clone assignment (:=) may be used. □

Any attempt of using a channel after it has been moved out by s-assignment leads to undefinedness or *CHAOS*.

Theorem 4.3.8 (Undefined channel)**.** $(\kappa chans :=_s ch \fatsemi ch.e) = CHAOS$

Proof.

$mdo(ch.e)$
$=\{mdo\,\text{def}\}$
$m\Phi(ch.e \notin dref' \lhd wait' \rhd dtr' = dtr \frown \langle(mChans, ch.e)\rangle)$
$=\{\lhd b \rhd \text{ def}, \text{conj. health. cond.} \vee -\text{closed}\}$
$m\Phi(ch.e \notin dref' \wedge wait') \vee m\Phi(\neg\, wait' \wedge dtr' = dtr \frown \langle(mChans, ch.e)\rangle)$
$=\{m\Phi \text{ def once}\}$
$m\Phi(ch.e \notin dref' \wedge wait') \vee \mathbf{MC123} \circ \Phi(\neg\, wait' \wedge dtr' = dtr \frown \langle(mChans, ch.e)\rangle)$
$=\{(\kappa chans :=_s ch.e) \Rightarrow (ch \notin mChan), \text{then}\, \boldsymbol{MC1}(dtr' = dtr \frown \langle(mChans, ch.e)) = false\}$
$m\Phi(ch.e \notin dref' \wedge wait') \vee \mathbf{MC23} \circ \Phi(\neg\, wait' \wedge false)$
$=\{\Phi\,\text{def}, \text{conj. health. cond.}, \text{pred. logic}\}$
$m\Phi(true \wedge wait') \vee false$
$=\{wait' = true, \text{pred. logic}\}$
$m\Phi(true)$
$=\{m\Phi\,\text{def}, \Phi\text{-def}\}$
$\mathbf{MC123} \circ and_B \circ \boldsymbol{R}(true)$
$=\{CHAOS\,\text{def}\}$
$CHAOS$

□

Acquisition. Moving in /receiving a channel *newch* requires that the receiving process must not own *newch* prior to receiving it, which corresponds to the assumption $(newch \notin mChans)_\perp$.

newch must then be added into *mChans*. Due to **MC3**, the value of *mChans* must be recorded into the trace, which will notably permit to increment the value of *dref* with the events of the acquired channel, subsequently. The operation for acquiring a new channel will be called (channel) *r-assignment* and denoted by $:=_r$.

Definition 4.3.9 (Channel r-assignment)**.** *Let newch be the channel to be acquired. Then:*

$$(\kappa ch :=_r newch) \;\widehat{=}\; \left(\begin{array}{l} (newch \notin mChans)_{\perp} \fatsemi \\ \begin{pmatrix} \kappa ch \\ mChans \end{pmatrix} := \begin{pmatrix} newch \\ mChans \cup \{newch\} \end{pmatrix} \fatsemi \\ mdo_{\Sigma}(nil) \end{array} \right)$$

To model the situation where the sending process just sends the channel but still retains its value, normal or clone assignment (:=) may be used. □

The preceding definition actually states that the behaviour of a process that receives a channel already owned should be *CHAOS*. That is a quite strict definition but it is up to the programmer to implement that behaviour however he would like to, probably though, by throwing an exception.

To further ensure that it is r-assignment $:=_r$ and not just assignment $:=$ that is used when receiving a channel, we define a new input prefix denoted by *ch*??, which behaves the same as *ch*? except that normal assignment $:=$ is replaced by r-assignment $:=_r$.

Definition 4.3.10 (Channel-passing input prefix)**.**
$(in??\kappa ch \longrightarrow P) \;\widehat{=}\; \Box_{newch}\, mdo_{\Sigma}(in.newch) \fatsemi \kappa ch :=_r newch \fatsemi P$ □

Channel mobility is characterised by s-assignment for the sending/source process, and by r-assignment for the receiving/target process.

Notation and terminology: In the rest of this thesis, we will often use the expression *channel carrier* to refer to any channel that is used for passing other channels, as the *in* channel above. For convenience, we will also use the notation $in??[newch]$ to denote the input of a new channel *newch*, instead of say $in??\kappa ch$. Thence $[newch]$ will denote the variable that has the channel *newch* for value. Conversely, we will use the notation $[\kappa ch]?msg$ to denote the channel contained in the variable κch.

4.3.4 Example: a mobile telecom. network

We have introduced the components and operations (notably the handover procedure) of a mobile telecom. network at the beginning of this Chapter, in (e.g.4.1.3). The description that follows is partly redundant with (e.g.4.1.3) but introduces new operations.

Description of a Mobile Telecom. Network. A Mobile Telecom. Network has three main elements: a user/client (caller or receiver); a base station or BTS and a control station

or BSC. A BTS provides users with radio links for their communications. Each BTS has a limited number of radio links also called the capacity of the BTS. Physically, a BTS may be conceived of as a big antenna that multiplexes/demultiplexes radio signals coming from smaller (radio) antennas. Each radio antenna or simply antenna provides a single radio link. When an antenna is faulty, the corresponding radio link is lost (viz. it may no longer be available for a radio transmission).

A BSC manages a given number of BTSs. It is also a big antenna, whose number of radio antennas is proportional to the number of BTSs it manages. Besides the passing of text messages (and voice), a BTS may exchange signals. Hence, there is a distinction between data channels (for the transmission of both voice and text messages or sms), and signal channels (for the transmission of protocol information).

When a BTS has a faulty radio antenna, it sends a fault signal or an alarm to its BSC. The alarm is then relayed from the BSC to network technicians, in charge of the maintenance of the network. In general, the fault may be either software, in which case a simple reconfiguration is necessary; or hardware, in which case the radio antenna must be replaced by a new one.

Very often, when a BTS is installed in an area with few people, the number of radio antennas it contains is less than its maximal capacity. However, when the population increases, new radio antennas are added until the full capacity is reached.

When the population in an area increases greatly, the number of BTS and BSC may be increased, or, the network as a whole may be reconfigured. During the time of the reconfiguration when new radio stations are being installed on part of the network, every communication may be passed through another section of the network: this operation is called load balancing. When the new section has been configured, load balancing again permits to transfer a number of communications through the new section.

Subsequently, the different components of the network i.e. the Client, the Base Station, and the Control Station are all represented by processes. The operations of the network that are modelled are: the handover, load balancing, and network maintenance operations such as the addition and the removal of equipments.

- Client: is always waiting 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 may hold a single transmission line only.

$$
\begin{aligned}
Cli^{max} &\mathrel{\widehat{=}} \Big|\Big|_{1\leq j\leq max} j.Cli \\
Cli &\mathrel{\widehat{=}} alloc??[talk] \rightarrow Chat \\
Chat &\mathrel{\widehat{=}} [talk]!msg \rightarrow \left(\begin{array}{l} (Chat \sqcap hangup \rightarrow lose![talk] \rightarrow Cli) \,\Box \\ \\ handoff \rightarrow lose![talk] \rightarrow Cli \end{array} \right)
\end{aligned}
$$

talk is the channel received by *Cli* for its transmissions. The event *hangup* models the action of a Client that puts an end to a transmission. Then, the Client releases its *talk* channel through the channel carrier *lose* and again waits for another *talk* channel. It also releases its *talk* channel when it receives a *handoff* event.

- Base Station (or Base Transceiver Station or BTS): is composed of many radio (or transmitter/receiver or TRE) stations. 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 behaves like the process that has released its transmission channel but still tries using it, which results in a faulty behaviour.

$$
\begin{aligned}
BTS^{max} &\mathrel{\widehat{=}} \Big|\Big|_{1\leq j\leq max} j.TRE \\
TRE &\mathrel{\widehat{=}} (talk?msg \rightarrow TRE) \triangle_{iev} FltyTRE \\
FltyTRE &\mathrel{\widehat{=}} dispose!talk \rightarrow TRE
\end{aligned}
$$

TRE receives messages from *Cli* (we have omitted the passing of the received message to another client for simplicity). The occurrence of a fault is modelled by an interrupt event *iev* which when it is triggered causes *TRE* to behave like *FltyTRE*. *FltyTRE* disposes of the faulty channel through the *dispose* channel and then behaves like *TRE* again, which will result in *CHAOS*. Hence, because of parallel composition in the definition of *BTS*, the previous definition is not good. A better definition would be to replace *TRE* by *STOP* in the semantics of *FltyTRE*.

- Control Station (or Base Controller Station or BSC - or Network Monitor): may communicate with a number of assigned BTS through signalling links. For simplicity we will represent signalling links as synchronisation events instead of communication events since we are not interested in the data exchanged through signalling links. Two particular signals are handover and load balancing. Additionally, the BSC is responsible for the allocation of a

transmission channel to any Client.

$$
\begin{aligned}
Monitor &\;\widehat{=}\; alloc![talk] \rightarrow \left(\begin{array}{l} (RetrieveTalk \sqcap loadb \rightarrow HandOver) \;\Box \\ HandOver \end{array} \right) \sqcap LoadB \\
RetrieveTalk &\;\widehat{=}\; (retrieve??[talk] \rightarrow Monitor)[lose \leftarrow retrieve] \\
HandOver &\;\widehat{=}\; handoff \rightarrow RetrieveTalk \\
LoadB &\;\widehat{=}\; loadb \rightarrow Monitor
\end{aligned}
$$

The *Monitor* process defined above may be conceived of as a distributed process with its instances associated with each *TRE* process. That is why we chose the name *Monitor* over the name *BSC* for the control station.

We chose the name *retrieve* for the channel used by the *Monitor* for retrieving a transmission channel that was previously sent to a Client. *RetrieveTalk* is the corresponding behaviour of *Monitor*. *retrieve* is the counterpart of the *lose* channel used by the Client when releasing an acquired transmission channel.

A handover operation occurs on two instances. In the first instance, the handover is due to a client changing of coverage area: this is modelled by the *handoff* event. In the second instance, the handover is due to load balancing (the client has not itself changed of coverage area, but its current coverage area is full - then, it is the network that triggers a handover procedure). Load balancing is modelled by the *loadb* event. After either a *handoff* or a *loadb* event, control returns to the current *Monitor* which is assumed to send channels associated with a different *BTS*.

- User-Network Interface (or UNI): the part of a telecom. network composed principally of the client and the BTS. This interface is concerned with the communication between the latter two.

$$UNI \;\widehat{=}\; Cli^{max} \parallel BTS^{max} \parallel Monitor$$

The semantics for the handover procedure presented above slightly differs from the one proposed by Milner [86] regarding certain design decisions. (1) In [86], it is the BTS that is responsible for passing a new *talk* channel upon the occurrence of a *handoff* event, received from the *Monitor*. Thus, the new *talk* channel passes from *Monitor* to *BTS*, and then from *BTS* to *Cli*. In our model such is not the case, *Monitor* communicates directly with *Cli*. (2) In [86], in the occurrence of a *handoff* event, *BTS* loses its *talk* channel (in addition to *Cli*), which is not the case in our model.

In the previous definition, the BTS had a static number of TRE which could decrease in case of a faulty channel. In what follows we describe the case where the BTS may acquire new TREs.

- The process that adds a new TRE is defined as follows:

$$\begin{aligned} newTRE &\mathrel{\widehat{=}} new??[talk] \rightarrow (n+1).TRE' \lhd n \leq max \rhd SKIP \\ TRE' &\mathrel{\widehat{=}} (talk?msg \rightarrow TRE) \triangle_{iev} FltyTRE' \\ FltyTRE' &\mathrel{\widehat{=}} dispose!talk \rightarrow STOP \end{aligned}$$

where max denotes the maximum number of TRE of a BTS, and $n > 1$ denotes the initial number of TRE of a BTS.

- A complex BTS may increase its number of TRE. It may also decrease its number of TRE, when a TRE is faulty.

$$\begin{aligned} ComplexBTS &\mathrel{\widehat{=}} \Box_{n \leq k < max} \bullet BTS^k \\ BTS^{n+1} &\mathrel{\widehat{=}} expand(BTS^n) \\ &= BTS^n \parallel newTRE \end{aligned}$$

BTS^n denotes the BTS that has n active TRE.

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

$$BTS^n = (\ \Big|\Big|_{1 \leq j \leq n-1} j.TRE') \parallel STOP = BTS^{n-1}$$

- A complex user-network interface (UNI) may be defined as a UNI which has a complex BTS.

$$ComplexUNI \mathrel{\widehat{=}} Cli^{max} \parallel ComplexBTS \parallel Monitor$$

Along the same line as above, we may define $BSC^m = \Big|\Big|_{1 \leq i \leq m} i.BTS$, the control station with m base stations. We may then expand and reduce the number of BTS just like we did the number of TRE earlier. In turn, we may put a number of BSC in parallel and obtain a mobile telecom. network with its complete hierarchy of components. The monitor process may be decentralised by defining monitor processes hierarchically as follows: *TREMonitor* would monitor a single *TRE* process; *BTSMonitor* would monitor up to *max* total number of *TREMonitor* and would be defined such as to increase and decrease with the number of *TRE*; and *BSCMonitor* would monitor a number of *BTSMonitor*. This construction methodology suggests that network components should be modelled as controlled components, which have two parts: a controller part e.g. *BTSMonitor*, and a machine part, e.g. *ComplexBTS*.

A similar hierarchical construction may be applied to any hierarchical network, including computer networks such as the Internet.

4.3.5 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.

By construction, recording the interface occurs as a by-product of the occurrence of an event. Hence when no event occurs the trace is empty, and we have the exact same laws as in the static case.

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

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

Definition 4.3.11 (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 may define the (DAT) trace merge for parallel composition by recursion as follows:*

$$\begin{aligned}
s \parallel t &\mathrel{\widehat{=}} t \parallel s \\
\langle (i, c)\rangle \frown x \parallel \langle (j, c)\rangle \frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, c)\rangle \frown u \mid u \in x \parallel y\} \\
\langle (i, a)\rangle \frown x \parallel \langle (j, c)\rangle \frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, a)\rangle \frown u \mid u \in x \parallel \langle (j, c)\rangle \frown y\} \\
\langle (i, nil)\rangle \frown x \parallel \langle (j, c)\rangle \frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, nil)\rangle \frown u \mid u \in x \parallel \langle (j, c)\rangle \frown y\} \\
\langle (i, c)\rangle \frown x \parallel \langle (j, d)\rangle \frown y &\mathrel{\widehat{=}} \{\} \\
\langle (i, nil)\rangle \frown x \parallel \langle (j, nil)\rangle \frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, nil)\rangle \frown u \mid u \in x \parallel y\} \\
\langle (i, nil)\rangle \frown x \parallel \langle (j, a)\rangle \frown y &\mathrel{\widehat{=}} \{\langle (i \cup j, nil)\rangle \frown u \mid u \in x \parallel \langle (j, a)\rangle \frown y\} \\
\langle (i_0, a)\rangle \frown \langle (i_1, nil)\rangle \parallel \langle (j, b)\rangle \frown y &\mathrel{\widehat{=}} \begin{pmatrix} \{\langle (i_0 \cup j, a)\rangle \frown u \mid u \in \langle (i_1, nil)\rangle \parallel \langle (j, b)\rangle \frown y\} \cup \\ \{\langle (i_0 \cup j, b)\rangle \frown u \mid u \in \langle (i_0, a)\rangle \frown \langle (i_1, nil)\rangle \parallel y\} \end{pmatrix} \\
\langle (i, a)\rangle \frown x \parallel \langle (j, b)\rangle \frown y &\mathrel{\widehat{=}} \begin{pmatrix} \{\langle (i \cup j, a)\rangle \frown u \mid u \in x \parallel \langle (j, b)\rangle \frown y\} \cup \\ \{\langle (i \cup j, b)\rangle \frown u \mid u \in \langle (i, a)\rangle \frown x \parallel y\} \end{pmatrix}
\end{aligned}$$

□

4.3.6 Dynamic Renaming

Let $P = a \to SKIP$. $P[b \leftarrow a] = b \to SKIP$ is the process that engages in action b whenever P engages in action a. Such a renaming may apply to channels as well, when we would like to

define the process $P[ch_2 \leftarrow ch_1]$ that engages in a given channel say ch_2, whenever P engages in a distinct channel say ch_1.

In the presence of channel mobility, however, such a renaming mechanism *may not* be possible in every case. Indeed, renaming in the static model supposes that $ch_1 \in \mathcal{I}P$. Such an assumption may be easily violated in the mobility model.

Name mismatch. A first issue that must be dealt with is that of name mismatch. If the (channel) name that is expected to be acquired is different from the one actually received, renaming will be vacuous, which is undesirable.

One possibility for solving this issue is by ensuring that the acquired channel is the expected one, when names are known in advance. Hence

$$[RenameSpec1] \qquad P[ch_2 \leftarrow ch_1] = P[ch_2 \leftarrow ch_1] \land P \Rightarrow ch_1 \in \mathcal{A}mc$$

is a possible specification. It states that ch_1 must belong to the list of channels ($\mathcal{A}mc$) that have been acquired by P at some point during its execution.

Unknown names. As stated earlier, renaming in static CSP supposes that the interface of a process is known in advance. Let $f : E \rightarrow F$ denote a renaming function. $f(P)$ denotes the process that behaves like P but with channels in F. We may write instead $f : \mathcal{I}\,P \rightarrow \mathcal{I}\,f(P)$. Then:

$$f(P) = P[\mathcal{I}\,f(P) \leftarrow \mathcal{I}\,P]$$

Static processes are ***SN*** healthy, i.e. (loosely) $mChans = mChans'$. The (static) renaming operation applies to known channels only, i.e.

$$f(P) = P[f(mChans)/mChans, su/mChans']$$
$$\text{where } su = f(mChans) \cup (mChans' - mChans)$$

The above operation notably ensures that it is impossible to specify that a channel may be renamed before the channel's acquisition. Such a restriction applies also to ***DN*** healthy processes.

The problem of unknown names occurs subtly for static renaming in the form of **name collision**. Looking at the value of su above, it is possible that $f(mChans)$ coincides with $(mChans' - mChans)$. Suppose that we could compute, for every process, the set $\mathcal{A}mc$ of all the names that have been acquired during the process activation. Then, the definition of the static renaming operation in the presence of channel mobility should obey the following property: $f(mChans) \cap \mathcal{A}mc = \{\}$.

One possibility of implementing the latter property is by *separating* $f(mChans)$ from $(mChans' - mChans)$, by using renaming. For example, we could define instead

$su = 0.f(mChans) \cup 1.(mChans' - mChans)$. One problem with this solution comes from the fact that we are reasoning from the point of view of the receiving process only. Hence, renaming $(mChans' - mChans)$ in the receiving process should trigger a corresponding renaming of the sending process. This solution is not adequate, unless one is prepared to propagate a local renaming to the whole environment.

The renaming of a process must be as independent of the process's environment as possible. It should represent the statement: *For all environment* $\varepsilon \bullet$ *For all channel* $ch \in \varepsilon \bullet$ *If (a given process)* P *receives* ch*, then* $f(P)$ *receives* $f(ch)$ *(for a given renaming function* f*).*

A simpler solution consists of regarding the given property as a *healthiness condition* on static renaming. Let f be a renaming function whose domain includes both $mChans$, and eventual new channels, in $(MCh - mChans)$. We have $\mathcal{A}mc \subseteq (MCh - mChans)$. Let $undef$ denote any undefined channel. Then:

$$\forall\, unkn \in (MCh - mChans) \bullet f(unkn) = undef$$

In particular, $f \circ f(mChans) = undef$.

In fact, so long that no channel in $f(mChans)$ is effectively acquired i.e. $f(mChans) \cap \mathcal{A}mc = \{\}$, and that f behaves like the identity function over the set $(MCh - (mChans \cup f(mChans))$ (of eventual new channels), then f is a valid renaming. The assumption $f(mChans) \cap \mathcal{A}mc = \{\}$ may thus be modelled by the specification $f \circ f(mChans) = undef$.

The introduction of a special name $undef$ hence makes the specification of static renaming easier. The undefined name should never appear in a trace nor in a refusal. Thus $MCh \cap \{undef\} = \{\}$. By definition, we have $f(undef) = undef$, as if f where actually defined over $\{undef\}$.

It may be tempting to identify the whole of $(MCh - mChans)$ with the undefined value $undef$. This poses no problem in the case of static renaming where we assume that f applies to the *initial* value of $mChans$ only. When dealing with dynamic renaming, we will allow f to apply to successive values of $mChans$ as well.

Definition 4.3.12 ($undef$)**.** *Let* $undef$ *denote the undefined channel.*

$$MCh \cap \{undef\} = \{\}$$

As a consequence, for any renaming function f *defined over processes,*

$$f(undef) = undef \qquad \square$$

Static renaming is characterised by the following law.

Definition 4.3.13 (Static renaming)**.** *Let f denote a renaming function. If*

$$f(P) \Rightarrow (\forall\, kn \in mChans \bullet f(kn) \neq undef) \wedge f(MCh - mChans) = undef$$

then f is valid and is called a static *renaming function.* □

Example 4.3.14. *Let $P = ch1!x \rightarrow acq??[ch2] \rightarrow [ch2]!y \rightarrow SKIP$.*

1. *The renaming $P[ch1 \leftarrow ch2] = ch2!x \rightarrow acq??[undef] \rightarrow [undef]!y \rightarrow SKIP$ is* unhealthy *since $undef \notin MCh$.*

2. *$P[ch1 \leftarrow ch3] = ch3!x \rightarrow acq??[ch2] \rightarrow [ch2]!y \rightarrow SKIP$ is a valid (static) renaming.* □

The problem of unknown names arises only when channel-passing input prefix is concerned, since it is the only operation that may increase the interface. The application of static renaming (using the function f above) to channel-passing input prefix yields the process:

$$f(\kappa\, ch??newch \rightarrow SKIP) = (\kappa\, ch??newch \rightarrow SKIP)[f(mChans)/mChans, su/mChans']$$

Hence (the channel contained in) *newch* will not be renamed. However, the following process renames *newch*:

$$(\kappa\, ch??newch \rightarrow SKIP)[f(mChans)/mChans, sv/mChans']$$
$$\text{where } sv = f(mChans) \cup h(mChans' - mChans)$$

for some renaming function $h \neq f$. Such a renaming operation will be called *dynamic renaming.*

Example 4.3.15. *The following process prefixes every name in the initial interface with the number 0, and then any acquired name with the number 1:*

$$(\kappa\, ch??newch \rightarrow SKIP)[0.mChans/mChans, sv/mChans']$$

where $sv = 0.mChans \cup 1.(mChans' - mChans)$. □

Notice that in the preceding example, we purposefully chose two functions f and h such that $f \neq h$. We also did not write $f(\kappa\, ch??newch \rightarrow SKIP)$, as we expect the function f to apply to $mChans$ only, and h to $(mChans' - mChans)$.

Suppose that above we had defined $sv = 0.mChans \cup 0.(mChans' - mChans)$ instead. Then, it should not be considered that $f = h$. The function f may eventually be defined over *unknown* names, whilst h is defined over *known* names only. That is, there is an implicit *separation* between any (unknown) channel $ch \in E$ (for $f : E \rightarrow F$), and any (known) channel $ch \in (mChans' - mChans)$.

The construction of the dynamic renaming function is discussed hereafter.

Let $f : MCh \rightarrow MCh$ define a function over all possible names, both known and unknown. The function f takes as a parameter a channel name and returns a substitute name. We will denote by $f(x)$ or equivalently f_x the projection of f on a given set x. For example, if ch is a channel, then $f(ch)$ denotes the renaming of ch, whereas $f(\{ch\})$ denotes the projection of f on the singleton $\{ch\}$.

In accordance with our earlier discussions, a number of restrictions must be applied to the construction of f for the semantics of dynamic renaming to be valid.

First, for consistency, assuming that $mChans' = mChans \cup \{newch\}$, we expect that:

$$f(mChans') = f(mChans) \cup h(newch)$$

where h is a given renaming function. The function h is meant to apply to new channels only (as they are acquired). The equality above ensures that previous renamings are preserved.

To ensure the absence of name collision e.g. between $f(mChans)$ and $newch$, we expect that:

$$f(mChans) \cap (mChans' - mChans) = \{\}$$

Although we have written $h(newch)$ above, h needs not be defined exclusively over the singleton $\{newch\}$. Let $mChans''$ denote the final value of $mChans'$ at the end of the future execution. In order to avoid name collision, we must have:

$$h(mChans') \cap (mChans'' - mChans') = \{\}$$

Notice the similarity between this constraint on h and the previous one on f. We return to its implication in greater detail latter.

Ideally, we would have defined h over the set $\mathcal{A}mc$ of all the channels that have been acquired from the beginning to the end of the process activation. However, defining such an h is near impossible, as it would be equivalent to building (renaming) functions dynamically. Rather, whilst keeping the separation between known and unknown names in mind, we first define a function f over both kinds of names. Then, by construction, we ensure that the function f applies to known names only, as they are the only ones to be given in parameter to f (viz. h).

The distinction between the functions f and h is in fact purely syntactical. It allows us to write properties such as:

$$f(mChans) \neq undef \ \wedge \ f(MCh - mChans) = h(MCh - mChans)$$

(Notice that unlike static renaming, we do not write $f(MCh - mChans) = undef$.) Since $mChans$ is known in advance, so is $f(mChans)$. h may possibly be defined over $f(mChans)$ but that is unhealthy viz. it would cause a name collision between substitute names and acquired names. To enforce the disjointness between $f(mChans)$ and $\mathcal{A}mc$ is thus equivalent to enforcing that $h \circ f(mChans) = undef$, for every possible value of $mChans$ i.e.

$$\forall\, newch : MCh \mid newch \notin mChans \bullet f(mChans \cup \{newch\}) = f(mChans) \cup h(newch)$$
$$\wedge\ h \circ f(mChans) = undef$$

or equivalently

$$\forall\, x \bullet f(x) \cap (MCh - (\{x\} \cup f(x))) = \{\}$$

To see this equivalence, let us generalise the equation $f(mChans') = f(mChans) \cup h(newch)$. Since we are concerned with channel-passing input prefix only, we can replace $mChans'$ by $x \cup E$, for a given x. Then:

$$\forall\, x, E \mid x \cap E = \{\} \bullet \exists\, h \bullet f(x \cup E) = f(x) \cup h(E)$$

- $h \circ f(x) = undef$ means that $h(E)$ is defined over $(MCh - f(x))$ only;
- $f(x \cup E) = f(x) \cup h(E)$ implies $h(E)$ is defined over $(MCh - (f(x) \cup \{x\}))$;
- Finally, $h \circ f(x) = undef$ is equivalent to $f \circ f(x) = undef$, thus $f(x)$ is disjoint from the domain of f, $dom\, f$. Hence $f(x) \cap (MCh - (\{x\} \cup f(x))) = \{\}$.

The following definition summarises the previous discussion.

Definition 4.3.16 (Dynamic renaming)**.** *Let* $f : \mathbb{P}\, MCh \to (\mathbb{P}\, MCh \to \mathbb{P}\, MCh) \mid x \mapsto f_x$, *where* f_x *is injective for all* $x \in \mathbb{P}\, MCh$, *and*

$$\forall\, x, E \mid x \cap E = \{\} \bullet \exists\, h \mid h \circ f(x) = undef \bullet f(x \cup E) = f(x) \cup h(E)$$

A function f *that obeys the property above is called a* dynamic *renaming function.* □

Example 4.3.17. *Consider an environment in which every channel name is prefixed with the number* 0. *Let* B *be a set of channels, then* $0.B = \{0.ch \mid ch \in B\}$.

1. *Let* $f : 0.B \to 1.B$ *be a renaming function. Define* $h : 0.B \to 1.B \mid y \mapsto f(y)$, *a renaming function. Then:*

   i. $\forall\, x, E \mid x \cap E = \{\} \bullet f(x \cup E) = f(x) \cup h(E)$

   ii. $\forall\, x \bullet h \circ f(x) = undef$

   Thus, in an environment in which every channel name is prefixed with the number 0, *the function* f *is a valid dynamic renaming function.*

2. Let

$$f'(x) = \begin{cases} f(x) & \text{if } x \in mChans \\ undef & otherwise \end{cases}$$

f' is a static renaming function.

3. Let

$$f''(x) = \begin{cases} f(x) & \text{if } x \in mChans \\ x & otherwise \end{cases}$$

f'' is a dynamic renaming function, since it is defined outside mChans, unlike f'. f'' behaves like the identity function on eventual new channels. □

Example 4.3.18. *Consider an extension of the preceding 0-prefixed environment with 1-prefixed channels. That is, for a given set B, the new environment contains channels from the set $0.B \cup 1.B$ only. The function f defined earlier allows renaming channels from the set $0.B$ only. We would also like to rename channels from the set $1.B$.*

1. Let $g : 1.B \to 2.B$, then g is a valid dynamic renaming, for the same reason as f previously. However, g does rename channels in the set $1.B$ only.

2. $f \cup g$ is a valid dynamic renaming operation iff $ran\, f \cap dom\, g = \{\}$ i.e. $g \circ f = undef$.[6]

3. From (2) we can deduce that $f \cup (g - (g \circ f))$ is a valid dynamic renaming operation. $(g - (g \circ f))$ denotes the projection of g on the set $(dom\, g - ran\, f)$. □

4.4 Dynamic hiding

A process does not need to know at all whether a channel is silent (i.e. hidden from the environment) or not. 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, we may separate communication concerns 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. It is hiding $P \setminus X$ that specifies in X what channel is to be considered silent.

Channel mobility has two consequences on hiding. The first is name *mismatch*.

[6] $dom\, f$ and $ran\, f$ denote respectively the domain and range of the function f.

Name mismatch. Let $P = conceal??\kappa ch \rightarrow [\kappa ch]?x \rightarrow SKIP$, let $X = \{newch\}$, where we suppose that $\kappa ch = newch$ after the communication. Name mismatch occurs if $\kappa ch \neq newch$ instead. A way of detecting name mismatch may be by testing the value of κch, as shown below:

$$P = conceal??\kappa ch \rightarrow (newch?x \rightarrow SKIP \lhd \kappa ch = newch \rhd error!)$$

A more abstract specification is possible, however. Since any acquired name that should be silent will be specified in X, we only need to ensure that X effectively contains acquired names. The set of all mobile channels that have been acquired, denoted by $\mathcal{A}mc$ (read Acquired Mobile Channels), may be calculated from the trace by summing up all the interfaces that have been recorded from start until termination (including both intermediate and final states).

The following law states that silent names must belong to the set of acquired names, independently of how often a name has been acquired and released. The law must be taken as completing the existing definition for hiding (cf. §2.5.1).

$$[NoMismatchSpec] \qquad \begin{aligned} P \setminus X &= (P \setminus X) \wedge X \subseteq \mathcal{A}mc \\ \mathcal{A}mc &\mathrel{\widehat{=}} \bigcup \{s \mid (s, e) \in mtr'\} \end{aligned}$$

The second consequence is the possibility of *unknown names*.

Unknown names. In the presence of channel mobility, silent names may not be known in advance. For example, what IP addresses a router may acquire during its lifetime is generally unpredictable. Nonetheless, what is always known is the channel carrier through which any mobile channel will be acquired. Therefore, specifying that a mobile channel (to be acquired through a given channel carrier *conceal*) should be hidden may take the form *hide any channel that is input through channel conceal*, assuming that *conceal* is already in the interface.

An immediate consequence of having unknown names is that X may not be defined. Another consequence is that mismatch cannot be tested as illustrated above since no name may actually be provided for the test. The [*NoMismatchSpec*] law formulated above may hence serve as a specification for the eventual value of X. However, [*NoMismatchSpec*] does not hint towards a way for building X. We study the construction of X hereafter.

Consider again $P = conceal??\kappa ch \rightarrow [\kappa ch]?x \rightarrow SKIP$. The value of κch may only be known at runtime, hence it may not be provided for the definition of X since (by definition) X must be specified independently of any run of P.

However, since we know that the channel to be hidden will be input through the channel *conceal*, we may substitute it (the new channel) with a given name *subch*, as in the following process:

$$Q = (conceal??\kappa ch)[subch \leftarrow (mChans' - mChans)] \rightarrow [\kappa ch]?x \rightarrow SKIP$$

The process $Q \setminus \{subch\}$ will then hide the acquired channel as expected. However, such a solution introduces yet again the problem of *name mismatch*, in a subtle way. Let $R = conceal![newch] \rightarrow SKIP$ be the process that sends the channel *newch* through the channel *conceal*. Then, in $P \parallel R$, P will receive the channel *newch*, whereas in $Q \parallel R$, Q will receive *newch* but replace it with *subch*. That is, when placed in the same environment, P and Q will exhibit different behaviours: P will communicate through *newch*, whereas Q will communicate through *subch*. The hiding operator $\setminus X$ may be viewed as providing an environment which hides the channels in the set X.

The previous discussion highlights the fact that when renaming is applied to a process, it is the purpose of the programmer to place that process in an environment that already contains the substitute channels. Such is not the case for the process Q above. Unless the programmer may trigger a similar substitution of *newch* by *subch* in the environment when Q receives *newch*, the previous approach is not *safe*. The renaming in Q was introduced specifically to deal with the hiding environment, as if the hiding operator had changed the semantics of P into Q. Ideally, a process should be able to deal with whatever channels are being used in its current environment viz. a process must be defined as independently of its environment as possible. This issue is related to the problem of closed world vs. open world semantics, or equivalently, the problem of the relation between the knowledge of a process and the knowledge of its environment (viz. knowledge of a channel) —cf. §4.6.4.

Instead of employing substitution as shown above, suppose that we had an oracle that could see into the future. The oracle could observe an execution of P and note down what channel P has acquired (in the future). Then, any channel name obtained thanks to the 'foresight' of the oracle could be used for specifying X. This method requires having confidence in the oracle, but thanks to the [*NoMismatchSpec*], we may verify that the oracle has given us the right channel: if such were not the case, [*NoMismatchSpec*] would be violated.

Example 4.4.1 (A motivating example)**.** *A mobile telecom. network and its operations have been described in §4.3.4. Operations such as the replacement of a faulty antenna and load balancing are instances of the addition of network resources, specifically radio links. Whereas the radio links Client-BTS are visible to clients, many radio links such as BTS-BTS, BTS-BSC, and BSC-BSC are invisible to clients. Generally, when the latter kind of links are added to the network, the addition is invisible to clients and constitutes an instance of dynamic hiding. Certain links are added for a private use of either the network operator or some external organisation. Such private links are called PBX (Private Branch eXchange); their addition to the network is invisible to every other network user and constitutes an instance of dynamic hiding.*

In every instance of dynamic hiding cited above, it cannot be said that the channels that were added into the network were known in advance. Yet, they were meant to be hidden. Some of the operations may be more complex than others but since they happen so often, dynamic

hiding can be regarded as a common operation, at least at the implementation level. □

A careful analysis of the previous example shows this: although certain silent names are not known at what may be considered to be the origin of time, they may be hidden only once they are actually known. The question that we are trying to answer is this: *Can we define a predicate (or program) that computes the set of silent channels?*

Such a predicate would take P as input and compute silent channels from the execution of P, in a first step. Then, in a second step, it would remove silent names from the trace of P as traditional hiding does.

Let *sil* denote the set-valued variable that contains channels that should be hidden. *sil* must be added to the alphabet of processes.

Definition 4.4.2 (*sil*, *sil*′)**.** *sil*, *sil*′ : $\mathbb{P}\, MCh$*, sil denotes the set of channels that were hidden in the previous execution; sil*′ − *sil denotes the set of channels to be hidden between the start and the end of the current observation.* □

Remark that every process may be written in the form of the hiding operator:

$$P = P \setminus \{\}$$

i.e.

$$P = P \setminus X \qquad \text{where } X = \{\}$$

In other words, for $X \neq \{\}$, any process of the form P or either $P \setminus X$ is equivalent to the process $\exists\, sil_0 \bullet P \setminus sil_0$. Thus, *any process may be seen as one that assigns a value to the variable sil.*

In what follows, we discuss how to represent mobile CSP processes as processes that contain the variable *sil* in their alphabet.

4.4.1 From mobile processes to silencing processes

In this section we discuss how (mobile) CSP processes compute the set of silent names, under the traditional hiding operator $\setminus X$. In order to make that computation easier, we will try to build processes that may be written under an expression with leftmost hiding suffix e.g. $P \setminus X$ where P does not contain a hiding suffix. Processes in that form will be said to be in *hiding normal form* (or HNF). Whilst this consideration is syntactic at first, it will provide the basis for the semantic characterisation of processes that have *sil*, *sil*′ in their alphabet, called *silencing* processes.

Notation: The following notation will be used to indicate how a given process P computes the value of *sil*: $P \Rightarrow sil' = X$. The value of *sil* is given by a process or a function (not

represented explicitly) that reads a process expression and returns the corresponding value of *sil*. We will also write $\Rightarrow sil := X \fatsemi sil := sil \cup Y$ in place of say $\Rightarrow sil' = X \cup Y$ to indicate how the value of *sil* may be computed step by step.

For ease, we will consider a notion of *atomic* process, written P as usual. Thanks to this, we can build the value of *sil* compositionally. The assumption of atomicity is reasonable. For example, whether one writes $x := 2$ or equivalently $x := 1 \fatsemi x := x + 1$ poses no problem for the computation of the set of silent names, which here is empty. Thus for ease, we will consider single expressions of the form $x := e$ and $do_{\mathcal{A}}(a)$ to be atomic expressions.

First, we discuss how to interpret any static process as one that computes the value of the variable *sil*.

Let P denote a ***SN*** healthy process. From what precedes, we have

$$(P = P \setminus \{\}) \Rightarrow sil' = \{\} \qquad\qquad P \setminus X \Rightarrow sil' = X$$

However, since $P \setminus X = P \setminus (X \cap \mathcal{I}\, P)$ (or equivalently, in the notation of mobile CSP) $P \setminus X = P \setminus (X \cap mChans)$, we must rather have:

$$P \setminus X \Rightarrow sil' = X \cap \mathcal{I}\, P$$

Since $(P \setminus \{\}) \setminus X = P \setminus X$, we have:

$$\begin{aligned}(P \setminus \{\}) \setminus X &\Rightarrow sil' = X \\ &\Rightarrow sil := \{\} \fatsemi sil := sil \cup X\end{aligned}$$

More generally $(P \setminus X) \setminus Y = P \setminus X \cup Y$, we have:

$$\begin{aligned}(P \setminus X) \setminus Y &\Rightarrow sil' = X \cup Y \\ &\Rightarrow sil := X \fatsemi sil := sil \cup Y\end{aligned}$$

Consider the case $(P \fatsemi Q)$, where P and Q are both ***SN*** healthy, and $P \fatsemi Q$ is ***SN*** healthy. Then,

$$\begin{aligned}(P \fatsemi Q = P \setminus \{\} \fatsemi Q \setminus \{\}) &\Rightarrow sil' = \{\} \\ &\Rightarrow sil := \{\} \fatsemi sil := sil \cup \{\}\end{aligned}$$

Similarly,

$$\begin{aligned}((P \fatsemi Q) \setminus X = P \setminus X \fatsemi Q \setminus X) &\Rightarrow sil' = X \\ &\Rightarrow sil := \{\} \fatsemi sil := sil \cup X \\ &\Rightarrow sil := X \fatsemi sil := sil \cup X\end{aligned}$$

The above equivalence means that $P \fatsemi Q$ may be viewed as atomic w.r.t. the computation of *sil*. This also shows that the computation of *sil* is made easier by pushing the hiding suffix $\setminus X$ leftmost.

Consider $P \fatsemi Q \setminus Y$. By definition, $Y \cap \mathcal{I}\, P = \{\}$, otherwise the composition would be invalid viz. it would violate the healthiness condition ***R1***. Indeed, if $ch \in Y \cap \mathcal{I}\, P$ and $ch \notin (\mathcal{I}\, Q - Y)$, then tr may contain events not in tr'. Thus,

$$\begin{aligned}(P \fatsemi Q \setminus Y = P \setminus \{\} \fatsemi Q \setminus Y) &\Rightarrow sil' = Y \\ &\Rightarrow sil := \{\} \fatsemi sil := sil \cup Y\end{aligned}$$

Ideally, we would like to write $P \fatsemi Q \setminus Y = (P \fatsemi Q) \setminus Y$. This is correct since $\mathcal{I}\, P \subseteq (\mathcal{I}\, Q - Y)$ implies that $\mathcal{I}\, P \subseteq \mathcal{I}\, Q$.

Consider $P \setminus X \fatsemi Q$. Then $(\mathcal{I}\, P - X) \subseteq \mathcal{I}\, Q$. Let $Z = \mathcal{I}\, Q - (\mathcal{I}\, P - X)$.

If $X \cap \mathcal{I}\, Q = \{\}$ then $P \setminus X \fatsemi Q = P \setminus X \fatsemi Q \setminus X$. However, $P \setminus X \fatsemi Q \neq (P \fatsemi Q) \setminus X$ since $\mathcal{I}\, P$ may have more elements than $\mathcal{I}\, Q$. For the latter equality to hold, we need to extend the *event* alphabet of Q with at least the set $\mathcal{I}\, P - \mathcal{I}\, Q$. Thus $sil' = X$.

Otherwise, if $X \cap \mathcal{I}\, Q \neq \{\}$, then $P \setminus X \fatsemi Q \neq P \setminus X \fatsemi Q \setminus X$. This means that after assigning X to *sil* in $P \setminus X$, we can no longer write $sil := sil \cup \{\}$ as we have been doing thus far. However, with the help of the renaming operator, it is possible to push $\setminus X$ leftmost.

Our rationale is the following: if we separate the set $X \subseteq \mathcal{I}\, P$ from the set $X \subseteq \mathcal{I}\, Q$, then we can safely hide the first whilst preserving the second. Indeed, channels from the set X are removed from P alone, and not from Q.

We may also regard the problem from the point of view of *visible* names. In effect, in $P \setminus X \fatsemi Q$, all the names in $\mathcal{I}\, Q$ are visible, and so are all the names in $(\mathcal{I}\, P - X)$ since $(\mathcal{I}\, P - X) \subseteq \mathcal{I}\, Q$ (by definition). Thus all that we need to do is to preserve the visibility of the set of visible names i.e. the interface.

Let $0.X = \{0.x \mid x \in X\}$ be the substitute of X in P. Then,

$$P \setminus X = (P[0.X \leftarrow X]) \setminus 0.X$$

Simply replacing $P \setminus X$ by $(P[0.X \leftarrow X]) \setminus 0.X$ is not enough, since we would like to obtain $(P \fatsemi Q) \setminus X$ in the end, i.e. $P[0.X \leftarrow X] \fatsemi Q$. In order for $P[0.X \leftarrow X] \fatsemi Q$ to be valid, we need to extend the alphabet of Q with $0.X$.

The event alphabet extension of a process Q with a set of events B is denoted by $Q_{+B} = Q \parallel STOP_B$. It does not engage in any event of the set B.

We have $\mathcal{I}\, Q = (\mathcal{I}\, P - X) \cup Z \cup X = (\mathcal{I}\,(P[0.X \leftarrow X]) - 0.X) \cup Z \cup X$, thus $\mathcal{I}\, Q_{+0.X} = \mathcal{I}\, Q \cup 0.X$. We obtain the following equivalence:

$$P \setminus X \fatsemi Q = (P[0.X \leftarrow X] \fatsemi Q_{+0.X}) \setminus 0.X$$

Thanks to having a leftmost hiding suffix, the value of *sil* is computed as shown earlier.

Consider $P \parallel Q$. Then,

$$\begin{aligned}(P \parallel Q = P \setminus \{\} \parallel Q \setminus \{\}) &\Rightarrow sil' = \{\} \\ &\Rightarrow sil := \{\} \cup \{\}\end{aligned}$$

If $\mathcal{I}\, P \cap \mathcal{I}\, Q \neq \{\}$, then

$$\begin{aligned}((P \parallel Q) \setminus X = P \setminus X \parallel Q \setminus X) &\Rightarrow sil' = X \\ &\Rightarrow sil := X \cup X\end{aligned}$$

and

$$\begin{aligned}(P \setminus X \parallel Q \setminus Y = (P \parallel Q) \setminus X \cup Y) &\Rightarrow sil' = X \cup Y \\ &\Rightarrow sil := X \cup Y\end{aligned}$$

If $\mathcal{I}\, P \cap \mathcal{I}\, Q \neq \{\}$, then it is possible to hide a channel in $\mathcal{I}\, P \cap \mathcal{I}\, Q$ for P, but not for Q. The situation is similar to the case $P \setminus X \fatsemi Q$ earlier. To obtain a leftmost $\setminus X$, all that we need to do is to separate the hidden channels of P that are visible in Q, by using renaming.

Let $S = \mathcal{I}\, P \cap \mathcal{I}\, Q$. Then, rename $S \cap X$ to $01.(S \cap X)$, and $S \cap Y$ to $02.(S \cap X)$:

$$P \setminus X \parallel Q \setminus Y = \begin{pmatrix} P[01.(S \cap X) \leftarrow (S \cap X)] \parallel \\ Q[02.(S \cap Y) \leftarrow (S \cap Y)] \end{pmatrix} \setminus \begin{pmatrix} 01.(S \cap X) \cup 02.(S \cap Y) \cup \\ (X - S) \cup (Y - S) \end{pmatrix}$$

Notice that we could have more simply renamed X to $01.X$, and Y to $02.Y$. Then,

$$\begin{aligned}P \setminus X \parallel Q \setminus Y &= P[01.X \leftarrow X] \setminus 01.X \parallel Q[02.Y \leftarrow Y] \setminus 02.Y \\ &= (P[01.X \leftarrow X] \parallel Q[02.Y \leftarrow Y]) \setminus 01.X \cup 02.Y\end{aligned}$$

In conclusion, it is possible to reduce every (***SN*** healthy) CSP process P to a form $P' \setminus X$ with a single, leftmost $\setminus X$ (P' some process).

We will refer to the form of a CSP process with a single leftmost $\setminus X$ as its *hiding normal form* or HNF. From our previous formalisation two things have been achieved: the transfor-

mation of (static) CSP processes into their hiding normal form, and the computation of the value of *sil*. In fact, the HNF entirely determines the value of *sil*.

Let $silOf(P)$ denote the process that computes the set of silent channels of a process P that is in hiding normal form. Then,

$$silOf(P) \;=\; P_{+sil} \fatsemi sil := sil \cup X \;=\; P \wedge sil' = sil \cup X$$

The set $\{silOf(P) \mid P \,is\, \mathbf{SN}\, healthy\}$ yields a class of processes that have *sil* in their alphabet. However, the method for their construction requires starting from ***SN*** healthy processes which implies that we are not able yet to characterise the class of predicates that have *sil* in their alphabets, the class of interest to us. In other words, we need to determine healthiness conditions for characterising processes with *sil* in their alphabet. We return to this latter.

Let us now consider ***DN*** healthy processes. A ***DN*** healthy process must have at least two snapshots. Let P and Q denote distinct snapshots, where P and Q are both ***SN*** healthy. Then $P \fatsemi (\kappa\, ch.[x] \rightarrow SKIP) \fatsemi Q$ is ***DN*** healthy.

The computation of the value of *sil* for ***SN*** healthy processes was studied previously, hence we only have to focus on channel-passing input and output prefixes.

We will first discuss the case $(P \fatsemi (\kappa\, ch.[x] \rightarrow SKIP) \fatsemi Q) \setminus X$, and determine if like in static CSP, the equivalence with $P \setminus X \fatsemi (\kappa\, ch.[x] \rightarrow SKIP) \setminus X \fatsemi Q \setminus X$ holds.

Consider the output of a channel. Let $mChans = \{ch1, ch2, ch3\}$ at the beginning of P, and suppose that we lose the channel $ch3$ between P and Q viz. the process $(P \fatsemi (\kappa\, ch![ch3] \rightarrow SKIP) \fatsemi Q) \setminus X$. Then $mChans' = \{ch1, ch2\}$ at the end of $\kappa\, ch![ch3] \rightarrow SKIP$ (granting termination). That is, the initial interface at the beginning of P is $mChans_1 = \{ch1, ch2, ch3\}$; and the interface at the beginning of Q is $mChans_2 = \{ch1, ch2\}$. Let $X = \{ch1, ch3, ch100\}$. Then

$$(P \fatsemi (\kappa\, ch![ch3] \rightarrow SKIP) \fatsemi Q) \setminus X = P \setminus X \fatsemi (\kappa\, ch![ch3] \rightarrow SKIP) \setminus X \fatsemi Q \setminus X$$

Consider the input of a channel. Let $mChans = \{ch1, ch2\}$ at the beginning of P, and suppose that we acquire a new channel $ch3$ between P and Q viz. the process $(P \fatsemi (\kappa\, ch??[ch3] \rightarrow SKIP) \fatsemi Q) \setminus X$. Then $mChans' = \{ch1, ch2, ch3\}$ at the end of $\kappa\, ch??[ch3] \rightarrow SKIP$ (granting termination). That is, the initial interface at the beginning of P is $mChans_1 = \{ch1, ch2\}$; and the interface at the beginning of Q is $mChans_2 = \{ch1, ch2, ch3\}$. Let $X = \{ch1, ch3, ch100\}$.

$$(P \fatsemi (\kappa\, ch??[ch3] \rightarrow SKIP) \fatsemi Q) \setminus X \neq P \setminus X \fatsemi (\kappa\, ch??[ch3] \rightarrow SKIP) \setminus X \fatsemi Q \setminus X$$

The issue lies with $Q \setminus X$. Notice that $ch3$ was provided *manually* in X, however, $ch3$ is actually *unknown* before runtime. As in static CSP, it *should not* be possible to hide a

channel that has not been acquired yet. For a ***DN*** healthy process this means that any silent name must belong to the initial interface whose value is given by the variable *mChans* (viz. of the very first snapshot).

Thus, even in the case $P \setminus X \fatsemi (\kappa\, ch??[ch3] \rightarrow SKIP) \setminus X \fatsemi Q \setminus Y$ $(Y \neq X)$, Y must belong to the interface of the very first snapshot since at the time of the definition of Q, the initial interface of Q is not *entirely* determined. The initial interface of Q is *partially* determined only: it will eventually contain channels from the interface of P, channels that are hence known. As a consequence we can assert that for any ***DN*** healthy process R

$$R \setminus X \Rightarrow sil' = X \cap mChans$$

In order the solve the issue of unknown names, we only need to separate known $ch3 \in \mathcal{I}\, Q$ from unknown $ch3 \in X$. Remark that $ch100$ in the example above poses no problem since $ch100$ is not acquired at any point.

We have used the same separation technique in the static case earlier. When dealing with the process $P \setminus X \fatsemi Q \neq P \setminus X \fatsemi Q \setminus X$ (where $X \cap \mathcal{I}\, Q \neq \{\}$), we separated the set $X \subseteq \mathcal{I}\, P$ from the set $X \subseteq \mathcal{I}\, Q$. This notably allowed us to obtain a leftmost $\setminus X$.

Similarly, by regarding the problem from the point of view of *visible* channels, all the names that have been acquired *at least once* must be visible. We only need to preserve their visibility.

In order to separate $ch3 \in \mathcal{I}\, Q$ from $ch3 \in X$, as before, we will substitute X by $0.X$. However, this is not enough. In the process

$$(\kappa\, ch??[ch3] \rightarrow SKIP) \setminus X \fatsemi Q \setminus X$$

we cannot guarantee that $0.ch3$ *will not* be the channel that is actually input. Thus, we must also substitute $ch3 \in \mathcal{I}\, Q$, hence, *every* acquired channel. See that $ch3 \in \mathcal{I}\, Q$ corresponds with $ch3 \in (mChans' - mChans)$. Hence, after an input prefix, we will replace $(mChans' - mChans)$ by $1.(mChans' - mChans)$.

Let $su = mChans + 1.(mChans' - mChans)$.

$$\begin{aligned} & P \setminus X \fatsemi (\kappa\, ch??[ch3] \rightarrow SKIP) \setminus X \fatsemi Q \setminus X \\ = & \\ & \left(\begin{array}{l} P[0.X \leftarrow X] \fatsemi \\ (\kappa\, ch??[ch3] \rightarrow SKIP)[0.X \leftarrow X, su/mChans'] \fatsemi \\ Q[0.X \leftarrow X] \end{array} \right) \setminus 0.X \end{aligned}$$

Notice how any acquired name x is renamed to $1.x$, thus, the initial interface of Q will contain $1.x$. Also note that the renaming operation $P[0.X \leftarrow X]$ applies to the initial interface *mChans* only (unknown names cannot be renamed 'manually').

Similarly, we have:

$$
\begin{array}{l}
\quad P \setminus X \fatsemi (\kappa\, ch??[ch3] \rightarrow SKIP) \setminus Y \fatsemi Q \setminus Z \\
= \\
\quad \left(\begin{array}{l} P[0.X \leftarrow X] \fatsemi \\ (\kappa\, ch??[ch3] \rightarrow SKIP)[0.Y \leftarrow Y, su/mChans'] \fatsemi \\ Q[0.Z \leftarrow Z] \end{array} \right) \setminus 0.X \cup 0.Y \cup 0.Z
\end{array}
$$

In accordance with sequential composition, $Y \cap (\mathcal{I}\, P - X) = \{\}$ and $Z \cap (\mathcal{I}\, P - (X \cup Y)) = \{\}$. In words, the procedure for obtaining a leftmost $\setminus X$ is the following: rename the initial interface, the very first value of *mChans*, to $0.mChans$; rename any new channel *newch* to $1.newch$; rename any set X in a hiding suffix $\setminus X$ to $0.X$. Then, only names in $0.mChans \cap 0.X$ will effectively be silent.

We have thus defined a hiding normal form (HNF) for ***DN*** healthy processes also. Note that the HNF form defined above applies to static hiding, which as we have argued, does not permit to hide names that will only be known at run time.

As earlier, the set $\{silOf(Q) \mid Q\, is\, \textbf{DN}\, healthy\}$ yields a class of processes that have *sil* in their alphabet.

More generally, the set $\{silOf(Q) \mid Q\, is\, a\, mobile\, process\}$ yields *all* the interesting processes that compute the set of silent channels *under* static hiding. In order to define the dynamic hiding operator, we will need to extend the previous class of processes. Before discussing such an extension, it is necessary for us to characterise more decisively that class of processes.

Recall

$$silOf(P) \;=\; P_{+sil} \fatsemi sil := sil \cup X \;=\; P \wedge sil' = sil \cup X$$

Certainly, by composing two processes P and Q that are in hiding normal form and then reducing the result of the composition again to hiding normal form, we will obtain a process that is in hiding normal form. Thus, by composing $silOf(P)$ and $silOf(Q)$, we *should* obtain a process with *sil* in its alphabet, a process that yields the *expected* value of *sil* (according to HNF).

The process $sil := X_0 \fatsemi P_{+sil}$ is also a process that has *sil* in its alphabet. Yet, the hiding normal form of processes has shown that *sil* may not be assigned just any value. In what follows, it is our objective to describe the rules that govern assignments to *sil*, and formalise these rules as healthiness conditions.

In order to obtain the HNF form of processes of the form $P \setminus X \fatsemi Q \setminus Y$, it was necessary to ensure that the set-value of *sil* only ever grows, i.e. *sil* may take successively the values X, then

$X \cup Y$. Thus, every name from any set X defined after a hiding suffix $\setminus X$ may by default be considered as silent. The renaming introduced in the definition of the HNF was there to ensure that some of these silent names are vacuous, according to the law $P \setminus X = P \setminus X \cap mChans$.

Definition 4.4.3 (***S1***)**.** ***S1*** $P = P \land sil \subseteq sil'$ □

In the process $P \fatsemi Q \setminus Y$, the visible names of P may not be contained in sil'. So in $P \setminus X \fatsemi Q \setminus Y$, $\mathcal{I}\, P - X$ may not be contained in sil'. The hiding normal form $(P[0.X/X] \fatsemi Q_{+0.X}) \setminus 0.X \cup Y$ preserves the separation between the set of visible names $\mathcal{I}\, P - X$, and the set of silent names sil'. That that separation is preserved is implemented in the equality between $P \setminus X \fatsemi Q \setminus Y$ and its HNF. However, whilst the $silOf()$ transformation will preserve that separation through yielding $P \land sil' = 0.X$, it is nonetheless possible to build the silencing process say $P \land sil' = K$ for some set of channels K, instead. Whilst we may build a set say $nosil$ of visible names, in conjunction with sil, yet because sil and $nosil$ are complementary, sil should entirely determine both silent and visible names. To ensure the preservation of visible names, we must simply ensure that sil' may *not* contain any new name from the initial interface besides those already in sil.

Definition 4.4.4 (***S2***)**.** ***S2*** $P = P \land sil' \cap (mChans - sil) = \{\}$ □

In order to avoid the possibility of *manually* adding unknown names for ***DN*** healthy processes, we ensured that no acquired name may ever be passed to sil, when defining the hiding normal form of ***DN*** healthy processes.

Definition 4.4.5 (***S3***)**.** ***S3*** $P = P \land sil' \cap (mChans' - mChans) = \{\}$ □

The law $P \setminus X = P \setminus X \cap mChans$ could be written as the healthiness condition: $P = P \land sil' \subseteq mChans$. However, such a condition is too strong because it prevents from adding into sil even channels that may never be in the interface (presumably). Since static hiding allows passing channels that are not in the interface to the hiding operator, we should not enforce a stricter definition of sil. The healthiness condition ***S3*** is sufficient since it prevents the addition of future channels.

As already stated, the hiding normal form above was given for the static hiding operator, hence it may properly be called *static hiding normal form*. The healthiness conditions ***S1***, ***S2***, and ***S3*** thus characterise mobile CSP processes that may be written in static HNF.

In order to define dynamic hiding, we must allow assignments of the form $\exists\, newch \in mChans' \bullet sil := sil \cup \{newch\}$.

Let $conceal(\kappa\, ch??newch)$ denote the process that is ready to engage in a *given* channel-passing input event $\kappa\, ch??newch$ first, and then adds the value of the variable $newch$ into the variable sil. Then,

$$\begin{aligned} conceal(\kappa\, ch??newch) &\mathrel{\widehat{=}} \kappa\, ch??newch \to sil := sil \cup \{newch\} \\ &= \kappa\, ch??newch \land (sil' = sil \lhd wait' \rhd sil' = sil \cup \{newch\}) \end{aligned}$$

The set

$$\{silOf(Q) \mid Q \textit{ is a mobile process}\} \cup \{conceal(c.[e]) \mid c.[e] \textit{ is a channel} - \textit{passing input event}\}$$

yields all the processes that have *sil* in their alphabet, of interest to us. Processes from the set $\{silOf(Q) \mid Q \textit{ is a mobile process}\}$ have a static hiding normal form, hence they may be characterised by the healthiness condition $\boldsymbol{S1} \circ \boldsymbol{S2} \circ \boldsymbol{S3}$ (the composition order is irrelevant since the healthiness conditions are all conjunctive). However, processes from the set $\{conceal(c.[e]) \mid c.[e] \textit{ is a channel} - \textit{passing input event}\}$ are not ***S3***, since they are defined such that $\exists\, newch \in mChans' \bullet sil := sil \cup \{newch\}$.

Processes that have sil, sil' in their alphabet, and that are $\boldsymbol{S1} \circ \boldsymbol{S2}$ will be called *silencing* processes.

Consider the process $P \setminus X$ to be in static hiding normal form. Then P itself does not contain in its expression the static hiding operator. The effect of the $silOf()$ transformation is actually to replace $P \setminus X$ by a process of the form $P \wedge sil' = X$. The latter process, whilst it contains the variable *sil* in its alphabet, does *not* perform hiding. This means that the postfix (static) hiding operator $\setminus X$ is not defined over silencing processes yet.

We will now define the operator that performs hiding over silencing processes. Let $hide(P, X) = P \setminus X$ denote the application of static hiding to a silencing process P. Then $hide(P, X)$ must be $\boldsymbol{S1} \circ \boldsymbol{S2} \circ \boldsymbol{S3}$, as shown above. Instead of static hiding, we aim to define the (dynamic hiding) operator $hide(Q)$ that takes a silencing process Q as parameter and returns $Q \setminus sil'$. Note that $Q \setminus sil'$ is just an abuse of notation and is meant only to illustrate the 'hiding' effect of the dynamic hiding operator.

Let P be a silencing process, then $P \setminus \{\bullet\}$ will denote the process that hides channels contained in the set-value of the variable sil'.

The hiding normal form of a CSP process allowed us to write any process P in the form $P \setminus X$. Using the $silOf()$ transformation, processes in hiding normal form $P \setminus X$ were translated into silencing processes of the form $P \wedge sil' = X$. The dynamic hiding operator $P \setminus \{\bullet\}$ is meant to translate a silencing process $P \wedge sil' = X$ into the silencing process $P \setminus sil'$. It is thus as if the suffix $\setminus \{\bullet\}$ were always leftmost, as in hiding normal form.

However, notice that $P \setminus \{\bullet\} \fatsemi Q = P \setminus sil' \fatsemi Q$. This means that we obtain in result the same effect of the static hiding operator $\setminus X$ (except that sil' may contain acquired channels in $mChans'$). As a consequence, $\setminus \{\bullet\}$ has a scoping effect on silencing processes: just like in $P \setminus X \fatsemi Q$ the elements of X *are not* hidden in Q, so the value of *sil* at the end of $P \setminus \{\bullet\}$

should not be passed forward to the next process.

$$P \setminus \{\bullet\} = \exists\, tra, refa, sila \bullet \left(\begin{array}{l} P \left[\begin{array}{l} tra/tr', refa/ref', mChansA/mChans, \\ mChansB/mChans', sila/sil' \end{array} \right] \wedge \\ tra = tr' \restriction (mChans - sil') \wedge \\ refa = ref' - sil' \wedge \\ mChansA = mChans - sil' \wedge \\ mChansB = mChans' - sil' \wedge \\ sila = \{\} \end{array} \right) \fatsemi SKIP$$

The previous definition works for silencing processes that may be written in hiding normal form. However, it does not work for the silencing process $conceal(\kappa\, ch??newch)$. Indeed, the substitution of $mChans'$ by $mChansB = mChans' - sil'$ above implies that the acquired channel *newch* will be removed from the alphabet, in the process $conceal(\kappa\, ch??newch) \setminus \{\bullet\}$, which is unhealthy. Indeed, the input of a new channel is meant to determine the interface of the next snapshot. That is, in the process $conceal(\kappa\, ch??newch) \fatsemi Q$, we expect *newch* to belong to the interface of Q. We also expect *newch* to be hidden at the end of the whole process, since **the hiding operator is supposed to hide the *use* of a channel, not its *acquisition*** (viz. its movement). However, the channel carrier $\kappa\, ch$ used for acquiring a new channel may be hidden since it belongs to the current interface. Thus we also define:

$$\begin{array}{l} \quad conceal(\kappa\, ch??newch) \setminus \{\bullet\} \\ = \\ \quad \exists\, tra, refa, sila \bullet \left(\begin{array}{l} P \left[\begin{array}{l} tra/tr', refa/ref', mChansA/mChans, \\ mChansB/mChans', sila/sil' \end{array} \right] \wedge \\ tra = tr' \restriction (mChans - sil) \wedge \\ refa = ref' - sil \wedge \\ mChansA = mChans - sil \wedge \\ mChansB = mChans' - sil \wedge \\ sila = \{\} \end{array} \right) \fatsemi SKIP \end{array}$$

Note that the operator $\setminus \{\bullet\}$ is not ***S1*** healthy, because it ends the scope of sil', by resetting the value of sil' to the empty set $\{\}$.

Recall that a silencing process P represents a mobile process that is in hiding normal form, say $f(P) \setminus X$, f being a given transformation. The suffix $\setminus X$ is the *only* one to cause an increase of the value of *sil* i.e. $sil' = sil \cup X$, when $X \neq \{\}$. Hence the silencing process $P \setminus \{\bullet\}$ may be viewed as the process $f(P) \setminus X \setminus \{\bullet\} = f(P) \setminus X$. Since the suffix $\setminus X$ is

always leftmost, the value of sil' is always increased at the end of a process expression.

However, the silencing process $conceal(\kappa\, ch??newch)$ causes an increase of the value of sil i.e. $sil' = sil \cup \{newch\}$, but before the expression of the process for which the hiding is valid. That is, we must write $conceal(\kappa\, ch??newch) \fatsemi Q \setminus \{\bullet\}$ if we want to hide $newch$ in Q.

The difference between $\setminus X$, which is leftmost, and $conceal(\kappa\, ch??newch)$, which is rightmost, explains the semantics of $\setminus \{\bullet\}$: that hiding must be leftmost, otherwise it is vacuous. Nonetheless, the semantics provided above may be slightly unsatisfactory.

Indeed, given that silencing processes compute the set of names that *must be silent*, it may be argued that using $\setminus \{\bullet\}$ is actually redundant with the variable sil, and that *it should not be necessary* for the programmer to explicitly write $\setminus \{\bullet\}$ in process expressions. That is, the programmer may expect that for any silencing process P, $P = P \setminus \{\bullet\}$. Thus, if $conceal(\kappa\, ch??newch)$ is the last operation in a process expression, then hiding $newch$ will be vacuous; however, if $conceal(\kappa\, ch??newch)$ precedes a given process Q, as in $conceal(\kappa\, ch??newch) \fatsemi Q$, then $newch$ will be hidden in Q.

Definition 4.4.6 (*S4*). *S4* $P = P \setminus \{\bullet\}$ □

S4 is very suggestive of a hiding normal form, but this time for silencing processes themselves. Except $conceal()$ and $P \setminus \{\bullet\}$, every other silencing process has been constructed from the hiding normal form of mobile processes. Since $P \setminus \{\bullet\} = P \setminus sil'$ for all P that do not contain the hiding suffix $\setminus \{\bullet\}$, we can deduce that the equivalence $P = P \setminus \{\bullet\}$ holds. Recall that all processes that are in (static) hiding normal form are characterised by the healthiness condition ***S1*** $\circ$ ***S2*** $\circ$ ***S3***. However, the operator $\setminus \{\bullet\}$ is not ***S1*** healthy since it has a scope ending effect, formalised by the substitution of sil' by the empty set $\{\}$. The healthiness condition ***S1***, which requires that $sil \subseteq sil'$, may be considered to be *necessary* for characterising processes that are in hiding normal form, since it permits to write equivalences of the form $P \setminus X \mathbin{op} Q \setminus Y = (f(P) \mathbin{op} g(Q)) \setminus f(X) \cup g(Y)$, (where f and g are given transformations, op a given operator) as shown earlier. Therefore, the problem of defining ***S4*** healthy processes is the problem of giving a hiding normal form to silencing processes.

To solve that problem, we have to remove the scope ending effect of $\setminus \{\bullet\}$, so that it can be ***S1*** healthy. We also have to separate $sil \in \mathcal{I}\, P$ from $sil \in \mathcal{I}\, Q$, in any sequential composition of the form $P \setminus \{\bullet\} \fatsemi Q \setminus \{\bullet\}$, just like we did for $P \setminus X \fatsemi Q \setminus Y$. Similarly for parallel composition.

Every silencing process for which $sil' \cap mChans = \{\}$ is ***S4***. This comes from the equivalence $P \setminus \{\bullet\} = P \setminus sil'$. For any P not mentioning $\setminus \{\bullet\}$, we have $P = (P \wedge sil' = \{\}) = P \setminus \{\bullet\} = P \setminus \{\}$.

Every silencing process that has a leftmost $\setminus \{\bullet\}$, e.g. $P \setminus \{\bullet\}$, is ***S4***. This comes from the equivalence $P \setminus \{\bullet\} = (P \setminus \{\bullet\}) \setminus \{\bullet\}$. Once the channels in sil' have been hidden a first time, hiding them a second time becomes vacuous.

Equivalences of the form $P \setminus X \mathbin{op} Q \setminus Y = (f(P) \mathbin{op} g(Q)) \setminus f(X) \cup g(Y)$ have been built earlier. The process on the *right hand side* of the equality is trivially ***S4***. Below we define operators *op* that ensure that the composition of two ***S4*** processes yield an ***S4*** process.

The normalised hiding operator, denoted by $\setminus_{HNF} \{\bullet\}$ behaves like $\setminus \{\bullet\}$, but it does not replace sil' by $\{\}$.

$$P \setminus_{HNF} \{\bullet\} = \exists\, tra, refa \bullet \left(\begin{array}{l} P \left[\begin{array}{l} tra/tr', refa/ref', mChansA/mChans, \\ mChansB/mChans' \end{array} \right] \wedge \\ tra = tr' \restriction (mChans - sil') \wedge \\ refa = ref' - sil' \wedge \\ mChansA = mChans - sil' \wedge \\ mChansB = mChans' - sil' \end{array} \right) ;\, SKIP$$

$$\begin{array}{l} \quad conceal(\kappa\, ch??newch) \setminus_{HNF} \{\bullet\} \\ = \\ \exists\, tra, refa \bullet \left(\begin{array}{l} P \left[\begin{array}{l} tra/tr', refa/ref', mChansA/mChans, \\ mChansB/mChans' \end{array} \right] \wedge \\ tra = tr' \restriction (mChans - sil) \wedge \\ refa = ref' - sil \wedge \\ mChansA = mChans - sil \wedge \\ mChansB = mChans' - sil \end{array} \right) ;\, SKIP \end{array}$$

The normalised sequential composition operator, denoted by $P \setminus_{HNF} \{\bullet\} \fatsemi_{HNF} Q \setminus_{HNF} \{\bullet\}$, pushes the hiding operator leftmost as follows:

$$P \setminus_{HNF} \{\bullet\} \fatsemi_{HNF} Q \setminus_{HNF} \{\bullet\} = (P[0.sil' \leftarrow sil'] \fatsemi Q_{+0.sil}) \setminus_{HNF} \{\bullet\}$$

The normalised parallel composition operator, denoted by $P \setminus_{HNF} \{\bullet\} \parallel_{HNF} Q \setminus_{HNF} \{\bullet\}$, pushes the hiding operator leftmost as follows:

$$\begin{array}{l} P \setminus_{HNF} \{\bullet\} \parallel_{HNF} Q \setminus_{HNF} \{\bullet\} = (P[01.1.sil' \leftarrow 1.sil'] \parallel Q[02.2.sil' \leftarrow 2.sil']) \setminus_{HNF} \{\bullet\} \\ \quad where\ sil' = 1.sil \cup 2.sil \end{array}$$

Note that $1.sil$ and $2.sil$ denote *local values* of *sil* in P and Q respectively, whereas $01.1.sil$ (resp. $02.2.sil$) denotes the set $\{01.x \mid x \in 1.sil\}$ (resp. $\{02.y \mid y \in 2.sil\}$).

The process $P[0.sil' \leftarrow sil']$ renames the channels in P that are in the set-value of sil'. It should not be confused with the process $P[0.sil'/sil']$, which substitutes the value of the variable sil' by $0.sil'$. $P[0.sil' \leftarrow sil']$ substitutes the channels from sil' by those in $0.sil'$ *for every* variable that ranges over channels, including $tr, ref, mChans, sil$, and their dashed counterpart.

4.4.2 The semantics

In this section we summarise all the results discussed earlier. The highlight here is the semantics for the dynamic hiding operator. The links with mobile CSP processes are also discussed.

The following definition characterises silencing processes.

Definition 4.4.7 (Silencing processes)**.** *A silencing process is a (**MC** healthy) mobile process whose alphabet is extended with the variables sil, sil', and additionally satisfies the healthiness conditions **S1** and **S2**.* □

The healthiness conditions for silencing processes are defined below.

Definition 4.4.8 (***S1-3***)**.** $\textbf{\textit{S1-3}} \mathrel{\widehat{=}} \textbf{\textit{S1}} \circ \textbf{\textit{S2}} \circ \textbf{\textit{S3}}$ □

S1 states that the fact that a channel was hidden previously does never change. ***S2*** states that a visible channel may not be hidden subsequently. ***S3*** states that unknown channels may not be silent.

All the previous healthiness conditions are conjunctive (cf. §4.3.1), hence they can be composed together in any order. They are also monotonic and idempotent.

As discussed earlier, processes that obey the previous healthiness conditions define mobile processes that have a hiding normal form under the static hiding operator.

Definition 4.4.9 (Static hiding normal form)**.** *A silencing process is in* static hiding normal form *(or static HNF) if and only if it is **S3**.* □

Not all the interesting silencing processes are ***S3*** healthy. The following process, denoted by $conceal(\kappa\, ch??newch)$, is ready to engage in channel-passing input prefix first, and then increases the set of silent channels on termination.

Definition 4.4.10 ($conceal(\kappa\, ch??newch)$)**.**

$$conceal(\kappa\, ch??newch) \;\mathrel{\widehat{=}}\; \kappa\, ch??newch \wedge (sil' = sil \lhd wait' \rhd sil' = sil \cup \{newch\})$$ □

$conceal(\kappa\, ch??newch)$ is not ***S3***. That is because it allows sil' to contain channels that have been acquired during the process's activation.

A particularly interesting silencing process, denoted by $P \setminus_{HNF} \{\bullet\}$, is the process that behaves like the process P except that no channels in the set of silent names of P may be visible.

Definition 4.4.11 (Dynamic hiding)**.** *Let* $P = \textbf{\textit{S1-3}}(P)$*, then:*

$$P \setminus_{HNF} \{\bullet\} = \exists\, tra, refa \bullet \left(\begin{array}{l} P \left[\begin{array}{l} tra/tr', refa/ref', mChansA/mChans, \\ mChansB/mChans' \end{array} \right] \wedge \\ tra = tr' \restriction (mChans - sil') \wedge \\ refa = ref' - sil' \wedge \\ mChansA = mChans - sil' \wedge \\ mChansB = mChans' - sil' \end{array} \right) ;\, SKIP$$

$$\begin{array}{l} \quad conceal(\kappa\, ch??newch) \setminus_{HNF} \{\bullet\} \\ = \\ \exists\, tra, refa \bullet \left(\begin{array}{l} P \left[\begin{array}{l} tra/tr', refa/ref', mChansA/mChans, \\ mChansB/mChans' \end{array} \right] \wedge \\ tra = tr' \restriction (mChans - sil) \wedge \\ refa = ref' - sil \wedge \\ mChansA = mChans - sil \wedge \\ mChansB = mChans' - sil \end{array} \right) ;\, SKIP \end{array}$$

□

We have chosen to present the normalised semantics only, because we are interested in processes that are ***S1*** healthy.

Finally, the hiding normal form for silencing processes is characterised below. It allows defining processes for which the channels in sil, sil' are *effectively* silent.

Definition 4.4.12 (Dynamic hiding normal form)**.** *A silencing process is in* dynamic HNF *if and only if it is* ***S4.2****.*

S4.2 $\quad P = P \setminus_{HNF} \{\bullet\}$ □

Every mobile process may be transformed into a silencing process. All that is needed is to transform the mobile process into a hiding normal form and then to extract the set of silent channels. This is the role of the transformation $silOf()$ introduced in the previous section. However, giving precise semantics to $silOf()$ is complicated by the fact it is very difficult to characterise semantically, in mobile CSP, processes that have a leftmost hiding suffix $\setminus X$. Indeed, such a characterisation may take the following form

$$\textbf{\textit{HNF}} \quad P = \exists f \textit{ a transformation}, X \subseteq \mathcal{I}\, P \mid f(P) \neq \textbf{\textit{HNF}} \circ f(P) \bullet f(P) \setminus f(X)$$

where $f(P) \neq \textbf{\textit{HNF}} \circ f(P)$ denotes that $f(P)$ does not itself contain a hiding suffix.

The other possibility would be to define the transformation *silOf*() over the syntax of mobile processes, but that would require a substantial amount of work. Instead, silencing processes allow us to model ***HNF*** as the healthiness condition ***S1-3*** $\circ$ ***S4.2*** under static hiding, and ***S1*** $\circ$ ***S2*** $\circ$ ***S4.2*** under dynamic hiding.

In the opposite direction, every ***S1-3*** healthy silencing process may be transformed into a mobile process of the form $P \setminus sil'$. The silencing process $conceal(\kappa\, ch??newch)$ may be transformed into the mobile process $(\kappa\, ch??newch)[subch \leftarrow (mChans' - mChans)]$, for a given *subch*. However, the definition of $\setminus \{\bullet\}$ shows that the substitute channel *subch* may not be hidden in the process $((\kappa\, ch??newch)[subch \leftarrow (mChans' - mChans)]) \setminus \{subch\}$, since *subch* will be used only in the subsequent snapshot. That notably ensures that hiding does not cancel the effect of channel-passing input. In sum, we may probably encode the dynamic hiding operator in mobile CSP, but that would involve defining a renaming procedure that could be rather complex (given that it must apply to all processes in any environment or execution context).

The following theorem determines how to encode the static hiding operator $\setminus X$ using the dynamic hiding operator $\setminus_{HNF} \{\bullet\}$ (or $\setminus \{\bullet\}$ - the choice of either is a matter of preference).

Theorem 4.4.13. *Let P be a mobile process viz. such that $sil, sil' \notin \alpha P$. Then,*

$$P \setminus X \;=\; \textbf{var}\, sil := X \fatsemi (P_{+sil}) \setminus_{HNF} \{\bullet\} \fatsemi \textbf{end}\, sil$$

Proof. In the semantics of $\setminus_{HNF} \{\bullet\}$, simply replace sil' by X. The result is $P \setminus X$ (after ending the scope of sil, sil'). □

Example 4.4.14. *The process that acquires five new channels consecutively but hides the first four channels. The process also outputs on each channel that has been acquired, but only the last output should be visible.*

$$SelectHid \;\widehat{=}\; n := 1 \fatsemi \mu Y \bullet \left(\begin{array}{l} (conceal(move??\kappa\, x) \fatsemi \kappa\, x!\text{“}ch\text{”}{+}n \rightarrow n := n + 1) \fatsemi \\ (Y \setminus \{\bullet\} \lhd n < 5 \rhd move??\kappa y \rightarrow \kappa y!\text{“}ch\text{”}{+}n) \rightarrow SKIP \end{array} \right)$$

□

Example 4.4.15 (PBX (Private Branch eXchange))**.** *The part of a mobile telecom. network that is reserved/leased for a private use either by the network operator itself or by an external organisation is called a PBX. We may model a PBX as a BTS (or part thereof) in which every channel is private.*

- A very simple PBX is obtained simply by indicating the set of leased channels, denoted

by pbx, when they are already known.

$$\begin{aligned} PBX(k) &\mathrel{\widehat{=}} (Cli^k \parallel (BTS^k \mid\mid\mid Monitor)) \setminus pbx \\ pbs &\mathrel{\widehat{=}} \{k.talk \mid k \leq max\} \end{aligned}$$

- The public channels of a network are accessible to every possible client. On the other hand leased channels are accessible to private clients only. The network that has both public and private parts is defined below.

$$\begin{aligned} UNI2 &\mathrel{\widehat{=}} Public \parallel PBX \\ Public &\mathrel{\widehat{=}} Cli^m \parallel (BTS^m \mid\mid\mid Monitor) \end{aligned}$$

where $1..k, k+1..m$*. Hence the channels of the process PBX are disjoint from those of the process Public.*

- The creation of a new PBX may be defined as the process that acquires a number k of new links and then hides them.

$$\begin{aligned} newPBX(k) &\mathrel{\widehat{=}} (privCli^k \parallel (privBTS^k \mid\mid\mid Monitor)) \setminus \{\bullet\} \\ privCli^k &\mathrel{\widehat{=}} \Big|\Big|_{1\leq j\leq k} j.privCli \\ privCli &\mathrel{\widehat{=}} conceal(alloc??[talk]) \rightarrow Chat \\ privBTS^k &\mathrel{\widehat{=}} \Big|\Big|_{1\leq j\leq k} j.privTRE \\ privTRE &\mathrel{\widehat{=}} conceal(new??[talk]) \rightarrow (n+1).TRE' \lhd n \leq max \rhd SKIP \end{aligned}$$

- A complex PBX may be defined as the process that adds new leased channels to an existing network.

$$\begin{aligned} ComplexPBX &\mathrel{\widehat{=}} \Box_{n\leq i<max} \bullet PBX^i \\ PBX^{n+1} &\mathrel{\widehat{=}} expand(PBX^n, k) \\ &= PBX^n \parallel newPBX(k) \end{aligned}$$

- A complex user-network interface has both a complex PBX and a complex BTS for elements.

$$\begin{aligned} ComplexUNI2 &\mathrel{\widehat{=}} ComplexPublic \parallel ComplexPBX \\ ComplexPublic &\mathrel{\widehat{=}} Cli^m \parallel (ComplexBTS^m \mid\mid\mid Monitor) \end{aligned}$$

□

4.5 Links with static CSP

The definition of mobile CSP has required the introduction of new symbols such as *dtr*, *dref*, *MCh*, and Σ. In order to define the link with static CSP, it is necessary to unify both notations. We may use $\mathcal{A}$ to stand for the set of authorised actions in both theories. Hence, we may discard Σ.

A subset of $\mathcal{A}$ is constituted of communication events, and is entirely determined by the corresponding set of channels, say $\mathcal{C}$. Whether some of the channels in $\mathcal{C}$ may be moved or not depends on the theory considered: in static CSP, none of the channels in $\mathcal{C}$ may be moved; in mobile CSP, a subset of $\mathcal{C}$, denoted by *MCh* contains mobile channels.

The variables *tr* and *ref* may be used to denote the trace and refusal of a process in either theories. For simplicity, we consider that in both cases, the traces are alphabetised (as defined earlier), and that we omit mentioning the interface element for static CSP (since the interface is a constant therein).

4.5.1 From static CSP to mobile CSP

Assuming that traces are alphabetised, we see that the healthiness condition ***SN*** entirely characterises static processes. Recall:

$$\textbf{\textit{SN}} \quad P = P \wedge \left(\begin{array}{l} \forall\, (s_1, e_1), (s_2, e_2) : \mathbb{P}(\mathit{Chans} \cup \mathit{MCh}) \times \Sigma \mid \#\ \mathit{mtr}' \geq 2 \bullet \\ (s_1, e_1) \frown (s_2, e_2) \in \mathit{mtr}' \Rightarrow s_1 = s_2 \end{array} \right)$$

Literally, ***SN*** implies that by default, a static process may not be understood as one that *cannot* acquire (resp. release) channels, in the sense that it does not have the capability for doing so. Rather, a static process is one that *may not*, under any circumstances, realise any channel-passing operation.

For illustration, (i) consider early desktops. They were *manufactured* such as to have a single Internet port, the Ethernet port. No matter what environment they were put in, they were never 'capable' of being added new ports. This illustrates a case when the limitation is in the device, such that channel mobility is impossible even in an environment where channel mobility is possible. In contrast, modern desktops are manufactured such that new ports may be added to them.

(ii) Consider a modern desktop, and suppose that the desktop has currently an Ethernet port only. Consider the NoWifi country, a country where WIFI ports may not reside (say, they are automatically disintegrated by some mystical phenomenon). Then, if we place a modern desktop in the NoWifi country, the desktop may never receive an additional WIFI port. This illustrates a case when the limitation is in the environment, such that channel mobility is impossible although the device may otherwise receive new channels.

Syntactically, a static process would not even contain a channel-passing operation in its expression/definition. Semantically, this means that the presence of one such operation would

lead to undefinedness, which is signified by ***SN*** being a healthiness condition. In conclusion, ***SN*** defines the identity function from static CSP to mobile CSP. Hence, it may not permit the transformation of a static network ***SN*** process into a dynamic network ***DN*** process.

4.5.2 From mobile CSP to static CSP

SN infers that static processes form a subset of mobile processes. The question here is the transformation of a ***DN*** healthy process into a ***SN*** healthy process. Recall:

$$\textbf{\textit{DN}} \quad P = P \wedge \begin{pmatrix} \exists\,(s_1, e_1), (s_2, e_2) : \mathbb{P}(\mathit{Chans} \cup \mathit{MCh}) \times \Sigma \bullet \\ (s_1, e_1) \frown (s_2, e_2) \in \mathit{mtr}' \Rightarrow s_1 - s_2 \neq \{\} \end{pmatrix}$$

We have seen that we may divide a dynamic network into snapshots, each snapshot defining a distinct topology (cf. §4.2). Syntactically, a snapshot is determined by the presence of a channel-passing operation in the process expression. In other words, a dynamic topology is an interleaving of static topologies.

Suppose then that we project each topology onto a unique static network such that: if P did not already own a mobile channel mch in the mobile network, then P will continuously refuse to engage in sch, the static equivalent of mch in the static network. Then, it may be possible to simulate a mobile network by a static network.
We emit the following hypothesis:

[HypothesisDNtoSN] A ***DN*** process may be simulated by a ***SN*** process.

In order to discuss the possible transformation of a dynamic network into a static one, we will use the case study described below. Our aim is to define first a ***DN*** process, then a ***SN*** process that is assumed to be equivalent to the precedent, and then discuss the transformation of the first into the second. We do this for a particular case first, and then discuss an eventual generalisation of the transformation.

Case Study: a trivial scenario - An informal specification of the link

A very simple ***DN*** process is one that has only two snapshots and in which only one channel has been moved between the two. Consider the network formed of three processes P, Q and R, such that P may send an arbitrary number of messages to Q through a channel mch; until Q passes its channel-end of mch to R; and then P sends messages to R. Such a network was illustrated in *Figure4.1* (§4.2.2); its semantics are given hereafter.

$$\begin{aligned} MN \;&\widehat{=}\; (P \parallel Q \parallel R) \setminus \{\kappa a\} \\ P &= mch!x \rightarrow P \\ Q &= mch?y \rightarrow (Q \,\Box\, \kappa a![mch] \rightarrow \mathit{SKIP}) \\ R &= \kappa a??[mch] \rightarrow \mu\, Y \bullet (mch?z \rightarrow Y) \end{aligned}$$

Notation: $[mch]$ stands for a channel holder variable containing the channel mch.

Consider the following static network formed of three processes P', Q', and R', such that P' may communicate with Q', until Q' is switched off (**move** signal) and then taken over by R'. The semantics are given by:

$$\begin{aligned} SN_1 &\mathrel{\widehat{=}} (P' \parallel Q' \parallel R') \setminus \{\mathbf{move}\} \\ P' &= sch!x \rightarrow P' \\ Q' &= sch?y \rightarrow (Q' \,\Box\, \mathbf{move} \rightarrow SKIP) \\ R' &= \mathbf{move} \rightarrow \mu Y (sch?z \rightarrow Y) \end{aligned}$$

The semantics of SN_1 follow closely the definition of MN. The synchronisation action **move**, has been introduced in order to achieve the same sequencing of the communications of P' first with Q', and then with R'. We have the following equivalence (on traces):

$$MN =_{\mathcal{T}} SN_1[mch \leftarrow sch]$$

There is a clear loss of expressiveness. Q' behaves like $SKIP$ once R' takes over viz. Q' *will no longer use* sch. Replace $SKIP$ by a distinct process (excluding both $STOP$ and $CHAOS$) say Q'_{aft}, i.e. $Q' = sch?y \rightarrow (Q' \,\Box\, \mathbf{move} \rightarrow Q'_{aft})$. How can we ensure that Q'_{aft} will no longer use sch?

A first idea that comes into mind for enforcing such a blocking of sch is to explicitly place sch in the refusals after the occurrence of **move**. This may be achieved in two different ways: (i) ensure that Q'_{aft} does not engage in sch; or (ii) ensure that the environment may no longer engage in sch.

Consider the following static process:

$$\begin{aligned} SN_2 &\mathrel{\widehat{=}} (P'' \parallel Q'' \parallel R'') \\ P'' &= \mu X \bullet (sch_1!x \rightarrow X \,\Box\, \mu Z \bullet (sch_2!x \rightarrow Z)) \\ Q'' &= sch_1?y \rightarrow Q'' \\ R'' &= sch_2?z \rightarrow R'' \end{aligned}$$

SN_2 is obtained by first relaxing the constraint that mch (cf. MN) needs to map to a single channel sch (cf. SN_1). The rationale is to provide, for each snapshot of Q in which mch appears in the interface, a specific static channel, sch_i. Here, the index i is meant to identify each snapshot, remembering that we have only two (2) in this case. sch_2 is meant to replace **move**.

SN_2 is an instance of a definition in which the environment blocks a further use of sch_1 after it has initiated the use of sch_2. The use of sch_2 is meant to represent that the system has entered into a new snapshot. As a consequence, Q'' blocks on any attempt of using sch_1,

which materialises the previous snapshot, once the system has entered into the new one. This also supposes that no process in the environment will ever use sch_1 again: P'' when it behaves like Z, and R'' may not use sch_1.

The renaming of both sch_1 and sch_2 into sch is meant to characterise in which snapshot each process is allowed to use sch (viz. mch). P'' may use sch in both snapshots; Q'' may use sch in the first snapshot, but not in the second one; and R'' may use sch in the second one only.

One thing that is left implicit in the semantics of SN_2 is how the dichotomy of the snapshots (viz. the indexing of sch) has been achieved. Indeed, this has been done manually, rather informally, by associating the occurrence of sch_2 with the entry into a new snapshot. Instead, we could make explicit that a new snapshot has been entered, by using the action **move** as follows:

$$\begin{aligned} SN_{2.1} &\mathrel{\widehat{=}} (P''^1 \parallel Q''^1 \parallel R''^1) \\ P''^1 &= \mu X \bullet (sch_1!x \to X \,\Box\, \mathbf{move} \to X[sch_2 \leftarrow sch_1]) \\ Q''^1 &= (sch_1?y \to (Q''^1 \,\Box\, \mathbf{move} \to SKIP)) \\ R''^1 &= (\mathbf{move} \to \mu Y (sch_2?z \to Y)) \end{aligned}$$

Let us compare SN_1 and $SN_{2.1}$. In the semantics of SN_1, entry into a new snapshot, materialised by the occurrence of **move**, is local to the processes involved, i.e. Q' and R'; P' is not concerned. In fact, a snapshot does not affect the integrity of a channel itself, only which process can use the channel. So P' may use sch in every snapshot, though SN_1 *does not encode the change of snapshot accurately.* Although we could prevent Q' from further using sch after the occurrence of **move**, this does not reflect channel mobility sufficiently well. We go even further by saying that this does not reflect channel mobility at all.

What we seem to be focused on is moving the channel whilst forgetting that such a movement results in a *change of topology.* If a topology must be understood simply as a *set of channels*, and assuming that each channel also determines what processes it links together (notwithstanding the effective use of the channel), we see that we may:

i. model each topology in isolation, as if defining distinct static processes, each one with its own distinct channels;

ii. then interleave them through prefixing or sequential composition: this would result in a process whose interface is the sum of all the interfaces of each snapshot;

iii. project every mobile channel onto a static channel, according to its presence into a snapshot.

We pointed out earlier that we could block sch in Q', in order to effect that sch may *not* be used by Q'. However, the procedure described above shows that *that would not be enough*: the change of snapshot must be 'effected' on every process *at the same time.* Such an effect may be described as follows:

- There is an initial snapshot, common to every process. Since we are in a static system, every channel that the system may use are authorised at once and may be mapped to an initial snapshot identifier, say $id = 1$. We assume that every channel is authorised for every process, by default. Then, in the actual definition, we put in the refusals all those channels that we do not want to appear in the trace just yet; and we remove them from the refusals otherwise.

- When the system enters into a new snapshot, the process that released its channel may no longer use it, e.g. Q'' may no longer use sch_1; meanwhile, the process that received the channel may now use it, e.g. R'' may now use sch_2. This presumes that in snapshot $id = 1$, R'' could not use any channel sch_i $(i \geq 1)$, which we may take to be in the refusals of R''; at the same time, Q'' could use sch_1 (but not sch_i, $i \geq 2$).

- The entry into a new snapshot is characterised by the update of id for every process, e.g. $id := id + 1 = 2$. The consequence is that every process that could *already* use say sch_1 (e.g. P'') will now use its equivalent in snapshot $id = 2$, i.e. sch_2.

Summary: the discussion above, which constitutes an informal specification of the transformation of a mobile ***DN*** process into a static ***SN*** process, suggests that it may *not* be possible to encode (or simulate) the effects of channel mobility in a static network without a mechanism for identifying snapshots. Such a mechanism must notably enforce that all the channels that do not belong to a given snapshot (hence are not identified within the snapshot) may not be used in that snapshot. Using refusals has been proposed as a means for realising such a blocking effect.

Conjecture 4.5.1. *A traces model (incl. refusals) alone is not expressive enough for* simulating *the effects of channel mobility in static CSP. A mechanism for identifying snapshots is also needed.* □

A final remark before formalising the transformation is that there seems to be a need for an explicit synchronisation of all the processes running in parallel. In SN_2 the synchronisation is implicit in the occurrence of sch_2; in $SN_{2.1}$, synchronisation is realised through the action **move**. The synchronisation affects every process in $SN_{2.1}$, whereas, in $SN_{1.1}$ below, it does affect only the processes involved in the mobility of a channel.

$$
\begin{aligned}
SN_{1.1} \;&\widehat{=}\; P'^1 \parallel ((Q'^1 \parallel R'^1)[sch, sch \leftarrow sch_1, sch_2]) \\
P'^1 &= sch!x \rightarrow P'^1 \\
Q'^1 &= (sch_1?y \rightarrow (Q'^1 \,\Box\, \mathbf{move} \rightarrow SKIP)) \\
R'^1 &= (\mathbf{move} \rightarrow \mu Y (sch_2?z \rightarrow Y))
\end{aligned}
$$

The semantics of $SN_{1.1}$ raise the following question: *is it necessary for every process to synchronise upon a change of snapshot at the same time?* Indeed, as far as communication is concerned, *sch* is the same in every snapshot, as SN_1 shows. Hence, *can we not increase of*

snapshot identifier individually in the processes involved, and then determine what channel may synchronise with which one only at the end? To illustrate the interest of this question, consider that at a given time, only one section of a network is concerned by the change of topology, so it is not quite necessary to affect every other process. Meanwhile, it should still be possible to tell, globally, what channel was present in which snapshot. This suggests an algorithm quite different from the one assumed so far. We will answer the latter question in the following section.

4.5.3 MCSN-simulation processes

In the previous section we have seen that a mechanism for identifying snapshots is necessary for defining a static network that may be used to *simulate* a dynamic network. Static ***SN*** healthy processes that simulate ***DN*** healthy processes will be called SN-simulation processes, or simply simulation processes. They are a 'simulation' in the sense that their alphabet is *static*, yet they have enough information to encode channel mobility.

In order to define simulation processes, we introduce into the alphabet of static (***SN*** healthy) processes the new observational variable denoted by id, to identity snapshots.

Definition 4.5.2 (id, id')**.** *Let* $IDs \subseteq \mathbb{N} \setminus \{0\}$ *denote a set of identifiers. Then:*
$id, id' : IDs$*, serve to identify snapshots.* id *identifies the current snapshot,* id' *the following snapshot.* □

It is first necessary to characterise static processes that have the variable id (viz. id') in their alphabet, if we are to transform ***DN*** healthy processes into (static) simulation processes. Let *iDgen* denote the transformation that computes the snapshot-identifier of a given process. Below, we will write $iDgen(P)$ to indicate how a given process P computes the value of id. This first characterisation of *iDgen* is *syntactic*, but it will be useful for achieving the semantic characterisation of simulation processes.

Notation: In what follows we will use the subscript suffix notation, e.g. ${}_sP$, to denote any ***SN*** process, i.e. $\mathbf{SN}({}_sP) = {}_sP$.

In order to build the function *iDgen*, we consider only two forms of predicates: 'snapshot separators' (r- and s-assignment) and snapshots (any other predicate that is ***SN***-healthy).

We expect *iDgen* to break syntactically a ***DN*** process into the successive ***SN*** processes that compose it, and attribute each snapshot a unique identifier: *iDgen* increments the value of the snapshot identifier id when encountering a snapshot separator; otherwise the value of id is unchanged.
- A ***SN*** process does not change of interface, hence, when *iDgen* takes one as input, it may not modify the value of id. Also, every mobile channel not in the interface remains in the future refusals.

$$iDgen({}_sP) \;\widehat{=}\; {}_sP_{+id} \qquad , \text{where } id \notin \alpha\, {}_sP$$

- Sequential composition. Let P and Q denote either snapshots or snapshot separators, then

$$iDgen(P \fatsemi Q) = iDgen(P) \fatsemi iDgen(Q)$$

- Choice operators. The value of *iDgen* entirely depends on the result of the choice. Let **choice** $\in \{\lhd\_\rhd, \sqcap, \Box\}$. Then

$$iDgen(P \textbf{ choice } Q) = iDgen(P) \textbf{ choice } iDgen(Q)$$

- Iteration. An iterative process may be seen as the sequential composition of all its iterative steps, whence

$$iDgen(b * P) = b * iDgen(P)$$

- Recursion. A recursive process $\mu X \bullet F(X)$ may be seen as the sequential composition of the resulting processes $F(X)$, whence

$$iDgen(\mu X \bullet F(X)) = \mu X \bullet iDgen(F(X))$$

- Let $N \subseteq MCev$ a set of channels to be acquired, then r-assignment adds channels from N into the interface. r-assignment is a snapshot identifier, hence, when *iDgen* takes one as input, it does increment the value of *id* whilst *removing* the channels in N from the future refusals.

$$iDgen(ch??N) \ \widehat{=} \ id' = id + 1 \wedge ref' = ref \setminus N$$

- Let $O \subseteq mChans$ a set of channels to be released, then s-assignment erases channels in O from the interface. When *iDgen* takes one as input, it does increment the value of *id* whilst *adding* the channels from O into the future refusals.

$$iDgen(ch!O) \ \widehat{=} \ id' = id + 1 \wedge ref' = ref \cup O$$

- Hiding. If a mobile channel was hidden in mobile CSP, it must be hidden in simulation CSP. *iDgen* simply substitutes every $ch \in X$ by its static equivalent in each snapshot in which *ch* is valid.

$$iDgen(P \setminus X) \ = \ iDgen(P) \setminus \{id \mapsto ch \mid id \in IDs \wedge ch \in X\}$$

- Parallel composition. We first introduce the following definitions. A *network process* defines a network (i.e. a parallel composition) of *node processes*. We consider that the eventual internal topology of each node is hidden. The topology of the network changes whenever the interface of one of its node changes. The interface of a node changes whenever there is a communication of one or more channels between that node and another. Hence, a change of

network topology corresponds to a change of interface of at least two nodes at once. We will say that two *nodes are connected* when they can exchange mobile channels between them. The transformation of the parallel composition operator must be determined by case. Below, we only specify the final value of *id*, in the merge operation.

- The simplest network is the one in which no node is connected to another. Such a network process defines a single snapshot viz. is ***SN***-healthy. Hence

$$\begin{aligned} iDgen({}_sP \parallel {}_sQ) &= ({}_sP \parallel {}_sQ)_{+id} \\ M(id) &\mathrel{\widehat{=}} id' = 1.id = 2.id \\ &\Rrightarrow id' = id \end{aligned}$$

- Next is the network in which only two nodes are connected. e.g. Let $P = P_1 ⨟ \kappa x![mch] ⨟ P_2$, $Q = Q_1 ⨟ \kappa x?[mch] ⨟ Q_2$. Then (expansion law):

$$(P \parallel Q) = (P_1 \parallel Q_1) ⨟ \kappa x.mch ⨟ (P_2 \parallel Q_2)$$

  Applying *iDgen* to the above expansion is trivial:

$$\begin{aligned} iDgen(P \parallel Q) &= iDgen((P_1 \parallel Q_1) ⨟ \kappa x.mch ⨟ (P_2 \parallel Q_2)) \\ &= iDgen(P_1 \parallel Q_1) ⨟ iDgen(\kappa x.mch) ⨟ iDgen(P_2 \parallel Q_2) \\ M(id) &\mathrel{\widehat{=}} id' = 1.id = 2.id \\ &\Rrightarrow id' = id + 1 \end{aligned}$$

  For a 2-process network, the two processes evolve in lock-step synchronisation viz. both enter into the new snapshot at the same time. Hence, we may 'safely' increase *id* locally for each process, as we are guaranteed to have the same value of *id* for both at the end. For the resulting static process, this means that every process involved will enter into the new snapshot at the same time.

- The preceding case gets more complicated if we introduce a third process $R = \boldsymbol{SN}(R)$. R is not involved in channel-passing with either P or Q. The synchronisation between P and Q was obtained for free, thanks to the actual synchronisation of the channel-passing operation itself. However, since R is not involved in that operation, we would first need to transform every channel-passing operation into an explicit synchronisation barrier.

  For simplicity, we will denote by a single event **move** any synchronisation that occurs because of channel mobility. (Or we may add a subscript for readability.) We have to rewrite the value of *iDgen* for snapshot separators such that each channel-passing operation is transformed into an action **move**, and increases the identifier as well.

Loosely, the expected effect would be

$$iDgen(\kappa x.mch) = do(\mathbf{move}) \wedge id' = id + 1$$

Formally:

$$\begin{aligned} iDgen(ch??N) &\mathrel{\widehat{=}} id' = id + 1 \wedge ref' = ref \setminus N \wedge tr' = tr \frown \langle \mathbf{move_{ch.N}} \rangle \\ iDgen(ch!O) &\mathrel{\widehat{=}} id' = id + 1 \wedge ref' = ref \cup O \wedge tr' = tr \frown \langle \mathbf{move_{ch.O}} \rangle \end{aligned}$$

Or equivalently:

$$\begin{aligned} iDgen(ch.E) \mathrel{\widehat{=}}\ & id' = id + 1 \wedge ref' = ref \cup (MCev \setminus mChans') \\ & \wedge\ tr' = tr \frown \langle \mathbf{move_{ch.E}} \rangle \end{aligned}$$

Then, we would need to transform R such that R does forcibly synchronise on **move** whenever P and Q do. Such a specification is almost infeasible (at least) at this level of abstraction. To see this, consider the following expansion:

$$\begin{aligned} (P \parallel Q \parallel R) &= \big((P_1 \parallel Q_1) \fatsemi \mathbf{move} \fatsemi (P_2 \parallel Q_2)\big) \parallel R \\ &= (P_1 \parallel Q_1 \parallel Rbefmove) \fatsemi \mathbf{move} \fatsemi (P_2 \parallel Q_2 \parallel Raftmove) \end{aligned}$$

Above, we have introduced quite purposefully the predicates *Rbefmove* and *Raftmove* such as to have $R = Rbefmove \fatsemi Raftmove$. However, we have no way of effecting such a transformation. If we had to deal with traces only (i.e. only the value of tr), then any interleaving would have sufficed for specifying both *Rbefmove* and *Raftmove*. However, since the whole state space is concerned, we cannot possibly hope to achieve the same sort of specification. Hence, we have to abandon the model that would rely of synchronising every process in a parallel composition on every instance of a channel-passing operation.

In simpler terms, what the last result means is that we cannot yet transform *MN* into $SN_{2.1}$. However, we can transform *MN* into $SN_{1.1}$, with the inconvenience that we can no longer block sch_1 when entering into a new snapshot. In consequence, P'^1 will keep using sch_1 after the occurrence of **move**, instead of sch_2, as we would prefer. What is needed is a way of increasing snapshot identifiers locally e.g. in Q' and R'; have P' synchronise with each in turn, as they happen to hold *sch*; and still map *sch* onto distinct snapshots.
This corresponds to a problem of the *compositionality* of snapshots: *how do we define the snapshots of the whole from the snapshots of the components to obtain the expected effect?*

Let us consider the effect of parallel composition as things stand i.e. snapshot identifiers are incremented locally by each process. This means that the value of *id* is given by the count of **move** actions in the final trace. Since the effect of parallel composition is simply to interleave such actions, it turns out that the final value of *id* is simply the sum of each component's own value of *id*, minus 1 (because the initial interface is common to every

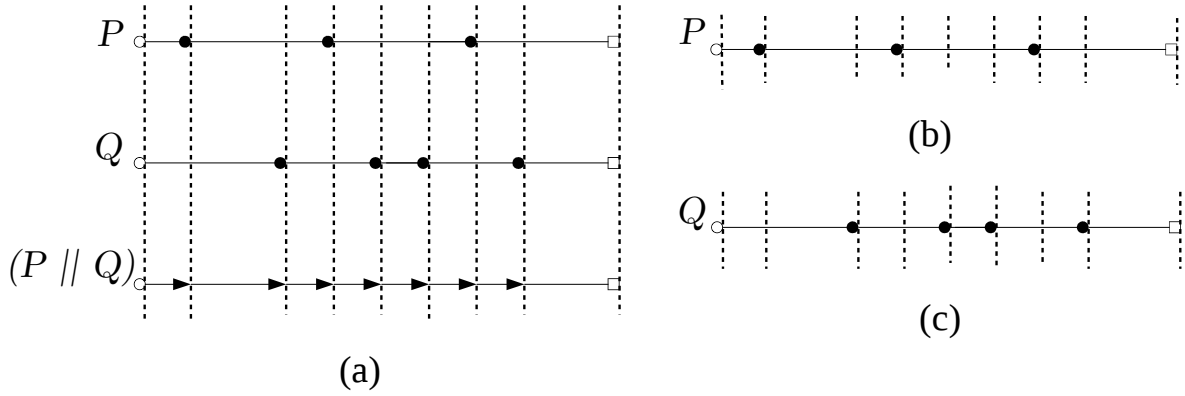

Figure 4.2: Snapshots of two parallel processes

process).

$$\begin{aligned} iDgen(P \parallel Q) &\;\widehat{=}\; iDgen(1.P) \parallel iDgen(2.Q) \\ M(id) &\;\widehat{=}\; id' = (1.id + 2.id) - 1 \end{aligned}$$

The issue of the *compositionality* of snapshots may be restated thus: since id identifies a snapshot uniquely, its final value $id' = 1.id+2.id-1$ seems to increase the number of snapshots of each component. *How can we relate the local snapshot divisions of each component process to the global snapshot division resulting from their parallel composition*? We discuss this question hereafter.

Consider *Figure4.2*. It represents two processes P and Q running in parallel. The execution of a process is represented by a horizontal line; the empty circle and square represent respectively the beginning and the end of the observation. The snapshots of P and Q are both represented by a full circle, whereas those of their parallel execution are represented by a full arrowhead. This is shown in *Figure4.2(a)*; *Figure4.2(b)* (resp. (c)) is the projection of the snapshots of $P \parallel Q$ onto the line of P (resp. Q), where those of the former are marked by dotted vertical lines. P has four (4) snapshots, Q five(5), and $P \parallel Q$ has eight ($8 = (4+5)-(1)$).

We may relabel the actions of P and Q with prefixes 1 and 2 respectively, but also their snapshot identifiers. For $P \parallel Q$, we may use the label 0, for readability. We are interested in the mapping $ch \mapsto \mathbb{P}\, IDs$ which maps a channel to all the snapshot identifiers in which ch has been acquired (and not released).

Let $sch \in \alpha P$ a channel of P, and suppose that sch is fixed, then, we have the mapping $sch \mapsto \{1.1, 1.2, 1.3, 1.4\}$, when P is considered alone. This simply means that sch is present in every snapshot of P. After projecting the snapshots of $P \parallel Q$ onto P, we have the mapping $sch \mapsto \{0.1, .., 0.8\}$, meaning that sch is present in every snapshot of $P \parallel Q$. Thus, there is no loss of information between the local and the global snapshot identification.

Now consider a mobile channel $mch \in \alpha P$ such that $mch \mapsto \{1.2, 1.3\}$. After the projection, we have $mch \mapsto \{0.2, .., 0.6\}$. Again, the original information is preserved: mch is not at the beginning, 1.1 or 0.1, and is not at the end, 1.4 or $\{0.7, 0.8\}$. Parallelism preserves information about channel mobility.

In general, we can map the snapshots of $P \parallel Q$ to those of both P and Q i.e.

$$\{0.1, .., 0.8\} \mapsto \{1.1, .., 1.4\} \cup \{2.1, .., 2.5\}$$

The reverse mappings are more instructive.

$$\begin{array}{llll} \mid 1.1 \mapsto 0.1 & \mid 1.2 \mapsto 0.2, 0.3 & \mid 1.3 \mapsto 0.4, 0.5, 0.6 & \mid 1.4 \mapsto 0.7, 0.8 \\ \mid 2.1 \mapsto 0.1, 0.2 & \mid 2.2 \mapsto 0.3, 0.4 & \mid 2.3 \mapsto 0.5 & \mid 2.4 \mapsto 0.6, 0.7 \mid 2.5 \mapsto 0.8 \end{array}$$

Recall (from $SN_{2.1}$):

$$P''^1 = \mu X \bullet (sch_1!x \rightarrow X \,\Box\, \mathbf{move} \rightarrow X[sch_2 \leftarrow sch_1])$$

Renaming is applied after every **move** action. The reverse mappings above suggest that such a renaming may be done in two steps instead of single one: first locally, e.g. mch will be mapped to $1.2 : mch$ and $1.3 : mch$; and then globally, e.g. $1.2 : mch$ will be mapped to $0.2 : mch$ and $0.3 : mch$, whilst $1.3 : mch$ will be mapped to $0.4 : mch$, $0.5 : mch$ and $0.6 : mch$.

Let us add a third process R to the equation such that $mch \in \alpha R$, and $mch \mapsto \{0.1, .., 0.8\}$. Then, when merging the traces of P and R, we will synchronise both P and R on channel mch in snapshots $0.2\,to\,0.6$. Since there is no local renaming in R, it is pointless to rename mch globally in R. Rather, what we may do is synchronise $1 : mch \in \alpha R$ with any $i : mch \in \alpha P$, where $i \in \{0.2, .., 0.6\}$.

A last remark is that *Figure4.2(a)* shows but a single interleaving of the snapshots of P and Q. Under a different interleaving, we would have a different mapping from local snapshots in P to global snapshots in $P \parallel Q$. A trivial example is the interleaving $P \fatsemi Q$. The mappings would then be

$$\begin{array}{llll} \mid 1.1 \mapsto 0.1 & \mid 1.2 \mapsto 0.2 & \mid 1.3 \mapsto 0.3 & \mid 1.4 \mapsto 0.4 \,..\, 0.8 \\ \mid 2.1 \mapsto 0.1 \,..\, 0.4 & \mid 2.2 \mapsto 0.5 & \mid 2.3 \mapsto 0.6 & \mid 2.4 \mapsto 0.7 \quad \mid 2.5 \mapsto 0.8 \end{array}$$

The application of *iDgen* to processes will be called *iDgen*(*process*). The following theorem summarizes the previous discussion.

Theorem 4.5.3 (*iDgen*(*process*))**.** *Let* ${}_sP$ *denote any* ***SN*** *healthy process; P and Q denote both* ***SN*** *and* ***DN*** *processes.*

$$
\begin{array}{rcl}
iDgen({}_sP) & = & ({}_sP)_{+id} \\
iDgen(ch.[E]) & = & do(\mathbf{move_{ch.E}}) \wedge \left(\begin{array}{l} SKIP_{+id} \lhd wait' \rhd \\ ref' = ref \cup (MCev \setminus mChans') \wedge id' = id + 1 \end{array} \right) \\
iDgen(P \parallel Q) & = & iDgen(1.P) \parallel iDgen(2.Q) \qquad \textbf{where} \quad id' = (1.id + 2.id) - 1 \\
iDgen(P \ \mathbf{op} \ Q) & = & iDgen(P) \ \mathbf{op} \ iDgen(Q) \qquad \textbf{where} \quad \mathbf{op} \in \{⨟, \lhd \_ \rhd, \sqcap, \Box\} \\
iDgen(b^* \, P) & = & b^* \, iDgen(P) \\
iDgen(\mu X \bullet F(X)) & = & \mu X \bullet iDgen(F(X)) \\
iDgen(P \setminus X) & = & iDgen(P) \setminus \{id \mapsto ch \mid id \in IDs \wedge ch \in X\}
\end{array}
$$

where ch.$[E]$ *is a shorthand for both channel-passing input and output prefix.*

Proof. cf. preceding discussion. □

The right-hand-side of the previous equations correspond to processes that have *id* in their alphabet. The latter represent mobile CSP processes as processes that compute the value of the snapshot-identifier.

The purpose of introducing the variable *id* was for it to rename any mobile channel *mch* into equivalent static channels $id \mapsto mch$. The definition above has left the renaming *implicit*, insisting only on the value of *id*.

Although simulation processes are computed from mobile CSP processes above, it poses no difficulty at all to compute them on their own. The discussion concerning the parallel composition operator implies that the parallel composition operator has a different semantics when applied to simulation processes than in static or either mobile CSP. Indeed, for the equality $iDgen(P \parallel Q) = iDgen(P) \parallel iDgen(Q)$ to hold, it is necessary for the parallel composition operator to *ignore* the *id* part in $id \mapsto mch$ and synchronise on *mch* only. This is unlike 'traditional renaming' and emphasizes the fact that *id* is indeed an *observation* variable.

We may build the renaming of mobile channels *mch* into identified static channels $id \mapsto mch$ in the traces directly as follows:

- in a DAT of the form $\langle ..., (s, e), ... \rangle$, replace s by the mapping $id \mapsto s$, where *id* is determined by *iDgen*. Similarly, replace every channel $ch \in s$ by $id : ch$, including both mobile channels and static channels. Hence, we obtain DATs of the form $\langle ..., (id \mapsto s, e), ... \rangle$.

Notice that DATs of the form $\langle ..., (id \mapsto s, e), ... \rangle$ may not serve to specify channel mobility. Since they are traces for static (simulation) processes, their expected form would be $\langle ..., (\mathcal{A}, e), ... \rangle$, but is not very useful for our purpose. Instead, we record the subset $s \subseteq \mathcal{A}$

that is valid for the process in snapshot id, keeping in mind that every other channel in $\mathcal{A} - s$ must be refused.

Granting the above, we may readily apply healthiness conditions ***MCx*** ($\mathbf{x} \in \{1, 2, 3\}$) to the new DATs. We will denote by ***MCSnsx*** (read MC Static network simulation) the simulation of ***MCx*** in a static network.

The translation of ***MC1*** yields the following healthiness condition.

Definition 4.5.4 (***MCSns1***)**.**

$$\textbf{\textit{MCSns1}} \quad P \;=\; P \wedge \forall\, id, s : \mathbb{P}\, MCh, e : MCev \bullet (id \mapsto s, e) \in tr' \Rightarrow e \in s \qquad \square$$

MCSns1 ensures that identified channels only may be recorded.

The translation of ***MC2*** yields the following healthiness condition.

Definition 4.5.5 (***MCSns2***)**.**

$$\textbf{\textit{MCSns2}} \quad P \;=\; P \wedge \left(\begin{array}{l} \forall\, (id_1 \mapsto s_1, e_1), (id_2 \mapsto s_2, e_2) \mid \#\ tr' \geq 2 \bullet \\ (id_1 \mapsto s_1, e_1) \frown (id_2 \mapsto s_2, e_2) \in tr' \Rightarrow \\ (id_1 = id_2 \;\Rightarrow\; s_1 = s_2) \;\vee\; (id_1 < id_2 \;\Rightarrow\; s_1 \subset s_2 \;\vee\; s_2 \subset s_1) \end{array} \right) \qquad \square$$

MCSns2 ensures that successive valid interfaces are not disjoint. For illustration, the interface history $\{sch1, sch2\} \frown \{sch3, sch4\}$ is invalid. ***MCSns2*** also ensures the correctness of a number of assumptions about snapshots, namely the fact that the sequence of snapshot identifiers is ever increasing, and that the same identifier associates to the same interface, and different identifiers to distinct interfaces.

The translation of ***MC3*** yields the following healthiness condition:

$$\textbf{\textit{MCSns3}} \quad P \;=\; P \wedge ref' \subseteq last\, \pi_1(tr')$$

Since we are in static CSP, $last\, \pi_1(tr') = \mathcal{A} = last\, \pi_1(tr)$, hence the clause $ref' \subseteq last\, \pi_1(tr')$ is always true. As a consequence, ***MCSns3*** can be discarded.

Definition 4.5.6 (***MCSns12***)**.** ***MCSns12*** $\widehat{=}$ ***MCSns1*** $\circ$ ***MCSns2*** $\square$

MCSns12 is conjunctive, hence it inherits all the properties of conjunctive healthiness conditions (cf. §4.3.1).

The following definition characterises static (***SN***) processes that are simulations of dynamic (***DN***) processes.

Definition 4.5.7 (Simulation processes)**.** *A* simulation process *is a static (**SN** healthy) process whose alphabet contains the variables id, id', and is additionally **MCSns12** healthy.* □

Def. 4.5.7 above implies that it is not possible to simulate channel mobility within a static UTP-CSP theory that does not mention *id* in its alphabet. This is a sensible result. We discuss its consequences hereafter.

Recall that *id* allows encoding snapshots in static CSP. The encoding takes the form of an association between channel names and *id* so that we can talk of a static model with (snapshot-)identified channels. For ease of reference, we may call it the *(static) CSP simulation model*, or simply *simulation CSP*. More importantly, *id* is used in the renaming of mobile channels into static ones. This renaming is at the heart of the simulation: it maps a mobile channel *mch* to static channels $id : mch$. Hence, unless there were another way of encoding such mappings without the use of *id*, which in turn is at the heart of the renaming, it is impossible of simulating a mobile network into static CSP.

Consequence 4.5.8. *Without a mechanism for identifying snapshots in static CSP, it is impossible of simulating a dynamic network using static CSP.* □

The latter result calls for a discussion of the works in the Literature. In the absence of the concept of *capability*, every traces model presented in Chap. 3 is insufficient for describing channel mobility, respectively. For illustration, the traces of Hoare & O'Hearn, and Vajar et al. may both be given the form $\langle ..., (s, e), ... \rangle$, just like our DATs. However, $s \in \mathbb{P}\,\mathcal{A}$ is used for restricting what channels may be observed, so that in the absence of the concept of *capability*, we may as well consider that they are still in static CSP. Indeed, let us compare $\langle ..., (s, e), ... \rangle$ with the traces in simulation CSP. The latter have the form $\langle ..., (id \mapsto s, e), ... \rangle$, which is identical to $\langle ..., (s, e), ... \rangle$ modulo the presence of *id*, and yet correspond to static CSP traces.

Consequence 4.5.9. *In the* absence of *the concept* capability*, every traces model in the literature defines a static CSP traces model.*
In particular, are concerned the works of Hoare & O'Hearn [67], Vajar et al. [132], Grosu et al. [56, 58, 123, 28, 29], Bialkiewicz & Peschanski [18], and Roscoe [108]. □

In the presence of the concept of capability, models that rely on name generation and freshness *a la* pi-calculus, e.g. [18], [108], are not enough for characterising channel mobility. Indeed, *iDgen* generates unique identifiers *id*, and hence unique names, in a way that is reminiscent of both Bialkiewicz & Peschanski' localised actions and Roscoe' standardized fresh names approaches. Yet, the processes resulting from *iDgen*, which belong to simulation CSP, are static CSP processes.

Consequence 4.5.10. *Guaranteeing name freshness* a la *pi-calculus in a traces model is not sufficient for characterising channel mobility. A further mechanism for identifying snapshots is necessary.* □

Furthermore, the characterisation of channel mobility in mobile CSP did not rely on a mechanism for generating fresh names (cf. §4.2). Concrete channels are unique by construction, thus, it is a fair consequence that two identical names always collide. Whilst *iDgen* provides a mechanism for constructing unique names, such a mechanism is neither sufficient nor necessary for characterising channel mobility.

Consequence 4.5.11. *Generating fresh names (i.e. names guaranteed to be fresh) is* not necessary *for characterising channel mobility; the notion of capability is.*
Generating fresh names is not sufficient *for characterising channel mobility; assuming that names are unique is.* □

In this section, we have characterised simulation processes, i.e. static processes which may be used to model ***DN*** healthy processes, which have a dynamic network topology. Such processes have the variables id, id' in their alphabet, and DATs of the form $\langle ..., (id \mapsto s, e), ... \rangle$. Their definition shows that a static network model may be used to model a dynamic topology, granting that additional information about snapshots is added to the static model. Although both simulation CSP and mobile CSP use DATs, simulation DATs are in fact static: the alphabet does not change, but the identifier associated with given channels.

In the next section we will define the transformation of ***DN*** healthy processes into simulation processes.

4.5.4 From *DN* healthy processes to *SN* healthy processes

In the previous section we have introduced the *iDgen* transformation, which notably extended the alphabet of mobile processes with the variable *id*. Except for channel-passing input and output prefixes, which both increment the value of *id* i.e. $id' = id + 1$, every other process behaves like *SKIP* i.e. $id' = id$.

Additionally, channel-passing input and output prefixes were transformed into synchronisation actions, in order to effect the cancellation of their channel-passing semantics. This latter behaviour is not *stricto sensu* that of *iDgen*, which was originally meant to compute the value of the identifier only. We hence introduce *cp2sync*, which transforms $c.[ch]$ channel-passing events into $\mathbf{move_{c.ch}}$ synchronisation events. Notice that this is a design decision, and we could have transformed $c.[e]$ channel-passing events into $c.e$ value-passing events instead. In fact, we could define a particular class of simulation (CSP) events that increase the value of *id*, just like channel-passing events modify the value of the interface *mChans* in mobile CSP.

Definition 4.5.12 (*cp2sync*)**.** *Let e denote any event excluding channel-passing communications; let* $c.[ch]$ *denote any channel-passing communication event.*

$$\begin{aligned} cp2sync(e) &\mathrel{\widehat{=}} e \\ cp2sync(c.[ch]) &\mathrel{\widehat{=}} \mathbf{move_{c.ch}} \end{aligned}$$

□

The renaming *cp*2*sync* behaves like the identity renaming on every event except channel-passing events.

So far, the renaming of mobile channels into identified static channels has been left implicit. Having computed the value of *id* for each process, we simply have to map *id* to the process's corresponding trace. In fact, all we need to do is to project *iDgen* onto traces and compute *id* as discussed above.

Definition 4.5.13 (*iDgen*(*trace*))**.** *Let e denote any event excluding communication events; let c.m denote any communication event excluding channel-passing communications; let c.*[*ch*] *denote any channel-passing communication event. Let a denote any sort of event.*

$$
\begin{aligned}
iDgen(id, \langle (s, e) \rangle) &\mathrel{\widehat{=}} \langle (id \mapsto s, e) \rangle \\
iDgen(id, \langle (s, c.m) \rangle) &\mathrel{\widehat{=}} \langle (id \mapsto s, c.m \rangle \\
iDgen(id, \langle (s, c.[ch]) \rangle) &\mathrel{\widehat{=}} \langle (id \mapsto s, c.[ch]) \rangle \\
iDgen(id, \langle (s_1, c.[ch]) \rangle \frown \langle (s_2, a) \rangle) &\mathrel{\widehat{=}} \begin{cases} \langle (id \mapsto s_1, c.[ch]) \rangle \frown \langle (id + 1 \mapsto s_2, a) \rangle & \text{if } s_1 \neq s_2 \\ \langle (id \mapsto s_1, c.[ch]) \rangle \frown \langle (id \mapsto s_2, a) \rangle & \text{otherwise} \end{cases}
\end{aligned}
$$

For readability, we have written $(id \mapsto s, c.m)$ *(resp.* $c.[ch]$*) in place of* $(id \mapsto s, id \mapsto c.m)$ *(resp.* $c.[ch]$*). We have also written* $id \mapsto s$ *in place of* $id \mapsto cp2sync(s)$*, and* $c.[ch]$ *in place of* $cp2sync(c.[ch]) = \mathbf{move_{c.ch}}$*.* □

Note that $cp2sync(s)$ renames every event $c.[ch]$ in the set s into a synchronisation event and leaves every other name unchanged.

The function *iDgen*(*trace*) yields a trace composed of identified channels. It is not enough however to define a renaming of mobile channels into identified static channels, because it applies exclusively to the trace of mobile processes.

Let $E(t)$ denote the set of all the elements of a given trace t, e.g. $E(\langle ..., (s_j, e_j), ... \rangle)$ is composed of elements (s_j, e_j). The set of all the channels that have been acquired at some point during an execution is given by

$$\mathcal{A}mc = \bigcup_j \{ s_j \mid s_j \in E(\pi_1(tr')) \}$$

where $\pi_1(t)$ yields the trace composed of the first components of elements of t, e.g. $\pi_1(\langle ..., (s_j, e_j), ... \rangle) = \langle ..., s_j, ... \rangle$.

Similarly, we define the set of corresponding identified channels:

$$id\mathcal{A}mc = \bigcup_j \{ id_j \mapsto ss_j \mid ss_j \in E(\pi_1 \circ iDgen(tr')) \}$$

iDgen(*trace*) entirely determines the relation between $\mathcal{A}mc$ and $id\mathcal{A}mc$, given below:

$$\mathcal{A}mc = \bigcup_j \{ cp2sync^{-1}(ss_j) \mid id_j \mapsto ss_j \in id\mathcal{A}mc \}$$

Recall that every set $s_j \in \mathcal{A}mc$ is transformed into an identified set $cp2sync(s_j) \in id\mathcal{A}mc$. Reciprocally hence, every set $ss_j \in id\mathcal{A}mc$ corresponds to a set $cp2sync^{-1}(ss_j) \in \mathcal{A}mc$, where $cp2sync^{-1}$ denotes the inverse of the function $cp2sync$.

The map $\mathcal{A}mc \mapsto id\mathcal{A}mc$ entirely determines the renaming of mobile channels into identified static channels, for each execution of a mobile process with final trace tr'.

Definition 4.5.14 ($iDgen(process)$)**.** *Let $u \in \{tr, ref, mChans\}$. Then:*

$$iDgen(P) \;\widehat{=}\; P[amc2idamc(u)/u, amc2idamc(u')/u']$$

where

$$amc2idamc : \mathbb{P}\, MCh \to \mathbb{P}(IDs \times MCh)$$
$$\mathcal{A}mc = \bigcup\nolimits_j \{s_j \mid s_j \in E(\pi_1(tr'))\} \mapsto id\mathcal{A}mc = \bigcup\nolimits_j \{id_j \mapsto ss_j \mid ss_j \in E(\pi_1 \circ iDgen(tr'))\} \qquad \square$$

The renaming function $iDgen(process)$ is defined in terms of the function $iDgen(trace)$, which generates identified channels, and the derived function $amc2idamc$ which does actually substitute mobile channels for identified *static* channels. The function $cp2sync$ substitutes mobile channels with synchronisation events, thus enforcing that $iDgen$ actually yields static channels.

Just like the variable $mChans$ in mobile CSP could be computed from the trace, so can id: simulation DATs have triples (id, s, e) for elements. The projection π_1 will be used to return the first element of such triples.

The function denoted by $dn2sn$ transforms ***DN*** healthy processes into ***MCSns*** healthy processes.

Definition 4.5.15 ($dn2sn$)**.** $dn2sn(P) \;\widehat{=}\; iDgen(P) \;\wedge\; id' = \pi_1 \circ last \circ iDgen(tr')$ $\square$

Theorem 4.5.16. *Let P be a mobile process, then*

$$dn2sn(P) = \boldsymbol{SN} \circ \boldsymbol{MCSns12} \circ dn2sn(P)$$

Proof. Correct by construction of the $dn2sn$ (viz. the $iDgen$) transformation. $\square$

Note: In the following examples, for ease, we will use the transformation $iDgen$ instead of $dn2sn$.

Example 4.5.17 (Trivial Scenario)**.** *Below, we apply the function iDgen to the calculation of the static equivalent of MN and compare $iDgen(MN)$ with the expected result, i.e. the static process $SN_{1.1}$.*

$$iDgen(MN) = iDgen(P \parallel Q \parallel R) = iDgen(P) \parallel iDgen(Q) \parallel iDgen(R)$$

$$
\begin{aligned}
iDgen(P) &= iDgen(mch!x \rightarrow P) \\
&= iDgen(mch!x) \rightarrow iDgen(P) \\
&= id : mch!x \rightarrow iDgen(P) \\
iDgen(Q) &= iDgen(mch?y \rightarrow (Q \,\Box\, \kappa a![ch] \rightarrow SKIP))) \\
&= iDgen(mch?y) \rightarrow iDgen(X \,\Box\, \kappa a![ch] \rightarrow SKIP) \\
&= id : mch?y \rightarrow (iDgen(Q) \,\Box\, iDgen(\kappa a![ch] \rightarrow SKIP)) \\
&= id : mch?y \rightarrow (iDgen(Q) \,\Box\, (iDgen(\kappa a![ch]) \rightarrow iDgen(SKIP))) \\
&= id : mch?y \rightarrow (iDgen(Q) \,\Box\, \mathbf{move} \rightarrow SKIP) \\
iDgen(R) &= iDgen(\kappa a??[ch] \rightarrow \mu\, Y \bullet (mch?z \rightarrow Y)) \\
&= iDgen(\kappa a??[ch]) \rightarrow iDgen(\, \mu\ Y \bullet (mch?z \rightarrow Y)) \\
&= \mathbf{move} \rightarrow \mu\, Y \bullet (iDgen(mch?z) \rightarrow iDgen(Y)) \\
&= \mathbf{move} \rightarrow \mu\, Y \bullet (id + 1 : mch?z \rightarrow iDgen(Y))
\end{aligned}
$$

Modulo the identifier, the expected static process, $SN_{1.1}$, and the calculated one, $iDgen(MN)$, are equivalent. □

4.5.5 Example: a circular FIFO buffer with mobile channels

This example is taken from [143]. Syntactically, the formulation of $MBuff^n$ is identical to the one in [143]. Semantically, the latter has Reactive Designs [35], [134], as its semantic domain, whereas the following rather has Reactive Processes as its semantic domain. The transformation is original to this thesis.

Consider a FIFO buffer with at least two cells such that each cell may store at most one data unit. There are two links between any two adjacent cells, so the buffer consists of two chains. The chain $leftrd \leftrightarrow rightrd$ allows passing the rd channel (for input); and the chain $leftwr \leftrightarrow rightwr$ allows passing the wr channel (for output). A cell may input only when it has acquired the input channel rd, and output when it has acquired the output channel wr.

Figure4.3(a) represents such a buffer: cells are represented by empty circles; links are represented by horizontal lines joining the cells. *Figure4.3(b)* represents the links between adjacent cells in greater detail: each link is marked with a channel name. The arrow indicates the flow of messages. In *Figure4.3(a)*, black squares mark rd and wr channels; unlike *Figure4.3(c)*, no line springs from the squares to indicate that rd and wr channels are mobile and hence, have eventually not been acquired yet. *Figure4.3(c)* shows the same buffer but in which rd and wr channels are both static, and are both used to form a third chain.

The behaviour of a n-cell buffer ($n \geq 2$) is given by:

$$MBuff^n \;\widehat{=}\; \langle\!\langle leftrd \leftrightarrow rightrd, leftwr \leftrightarrow rightwr \rangle\!\rangle\, i : 1 \,..\, n \bullet i.MCell$$

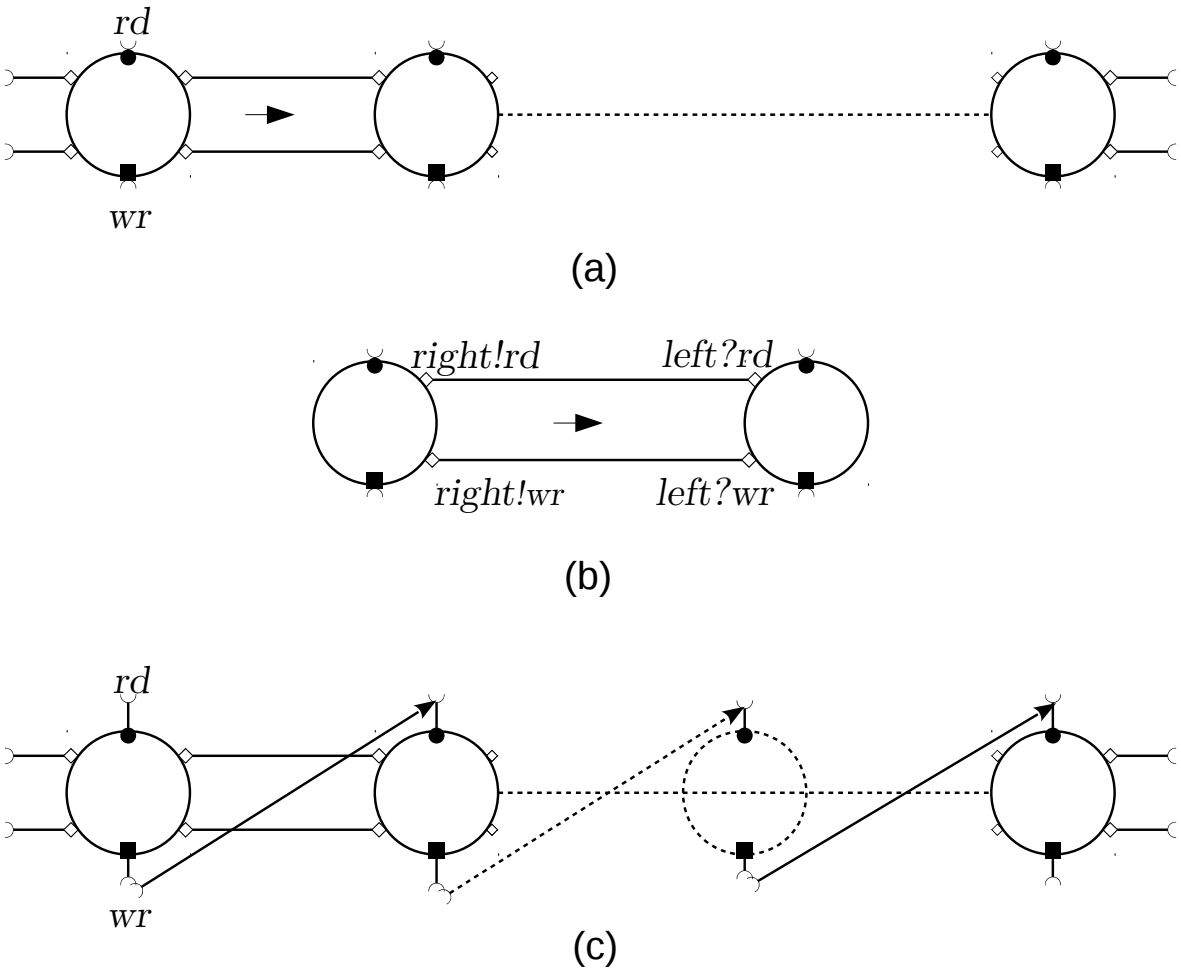

Figure 4.3: A buffer with mobile channels

$$
\begin{aligned}
MCell &\mathrel{\widehat{=}} C(rd, wr) \,\Box\, C \\
C(rd, wr) &\mathrel{\widehat{=}} [rd]?x \to rightrd!rd \to C(wr, x) \\
C(wr, x) &\mathrel{\widehat{=}} [wr]!x \to rightwr!wr \to C \,\Box\, leftrd??rd \to C(rd, wr, x) \\
C &\mathrel{\widehat{=}} leftrd??rd \to C(rd) \\
C(rd, wr, x) &\mathrel{\widehat{=}} [wr]!x \to rightwr!wr \to C(rd) \\
C(rd) &\mathrel{\widehat{=}} leftwr??wr \to C(rd, wr) \,\Box\, [rd]?x \to rightrd!rd \to C(x) \\
C(x) &\mathrel{\widehat{=}} leftwr??wr \to C(wr, x)
\end{aligned}
$$

$MBuff^n$ defines a buffer with mobile channels as a pipe of cells. Each cell is represented by a process *MCell*. Process expressions with BNF $C(\textit{read channel}, \textit{write channel}, \textit{data})$ denote the different states of a cell, such that the presence (resp. the absence) of a parameter indicates that the Cell possesses the corresponding element. For illustration, $C(rd, wr)$ denotes the behaviour of a Cell that owns both *rd* and *wr* channels.
At any time, a cell may be waiting on any of its four channel carriers. The semantics ensure that: an empty cell may never write or output data (i.e. may not use *wr*); a full cell may never read or input data (i.e may not use *rd*).

We want to transform the buffer $MBuff^n$ into a static one which we expect to be equivalent to the following buffer:

$$
\begin{aligned}
Buff^n &\mathrel{\widehat{=}} 1.Cell \,\langle\!\langle\, rd \leftrightarrow wr \,\rangle\!\rangle\, (\, \langle\!\langle\, rd \leftrightarrow wr \,\rangle\!\rangle\, i : 2 \,..\, n-1 \bullet i.Cell) \,\langle\!\langle\, rd \leftrightarrow wr \,\rangle\!\rangle\, n.Cell \\
Cell &\mathrel{\widehat{=}} rd?x \to wr!x \to Cell
\end{aligned}
$$

Notice how in $Buff^n$ both *rd* and *wr* channel ends are linked, which was not the case in $MBuff^n$ (viz. the chain $rd \leftrightarrow wr$ is not circular). This imposes a *further requirement* to the transformation: the creation of the third (non-circular) chain $rd \leftrightarrow wr$. We return to this latter on.

Let $SBuff^n \mathrel{\widehat{=}} iDgen(MBuff^n)$ the transformation of $MBuff^n$ into an equivalent static process. Then:

$$
\begin{aligned}
SBuff^n &= \Big|\Big| i : 1 \,..\, n \bullet iDgen \circ (i.MCell) \\
iDgen \circ MCell &= iDgen \circ C(rd, wr) \,\Box\, iDgen \circ C \\
iDgen \circ C(rd, wr) &= rd?x \to \mathbf{move!rd} \to iDgen \circ C(wr, x) \\
iDgen \circ C(wr, x) &= wr!x \to \mathbf{move!wr} \to iDgen \circ C \,\Box\, \mathbf{move?rd} \to iDgen \circ C(rd, wr, x) \\
iDgen \circ C &= \mathbf{move?rd} \to iDgen \circ C(rd) \\
iDgen \circ C(rd, wr, x) &= wr!x \to \mathbf{move!wr} \to iDgen \circ C(rd) \\
iDgen \circ C(rd) &= \mathbf{move?wr} \to iDgen \circ C(rd, wr) \,\Box\, rd?x \to \mathbf{move!rd} \to iDgen \circ C(x)
\end{aligned}
$$

$$iDgen \circ C(x) = \mathbf{move}?\mathbf{wr} \rightarrow iDgen \circ C(wr, x)$$

The equations above are identical to the ones for the dynamic buffer, except that communications/movement of mobile channels have been replaced by **move** actions (e.g. **move!wr**), so corresponding chains, i.e. $leftrd \leftrightarrow rightrd$ and $leftwr \leftrightarrow rightwr$, have been eliminated. For readability however, we have renamed them as if they were communication events. Also notice that *rd* and *wr* channels have no square-brackets around them, e.g. $[rd]$, since they are now considered to be static. We have chosen to omit mentioning *id*, e.g. $id : rd?x$, also for readability. We only need to keep in mind that each **move!wr** increases the value of *id*.

A consequence of the elimination of the previous chains is that cells now execute in parallel, without passing data from one cell to the next. **move** actions turn out to enforce the expected ordering of data input and output. That is, any input (resp. output) of data by a cell is guarded by synchronisation signals from its predecessor: a cell may output only if its predecessor has already output (**move**?**wr** signal); a cell may input only if its predecessor has already input (**move**?**rd** signal).

In $SBuff^n$, cells are not chained through their rd / wr channels. This is because there is no such chain in the mobile version defined above. In order to achieve a transformation closer to $Buff^n$, it is easier to add the chain at the end of the *iDgen* transformation.

Indeed, suppose that we modify $MBuff^n$ by adding a third chain $rd \leftrightarrow wr$ in its definition. Then, the result is not a chain in the traditional sense since only two nodes may be linked at any one time —instead of all nodes (traditionally). Such an addition would not be efficient since it would introduce unnecessary blocking: Whence the choice of adding the third chain at the end of the transformation.

Since the chain $rd \leftrightarrow wr$ is *not* circular, we need to single out the first and last cells such that they may respectively input from, and output to the environment.
Let $SCell = iDgen \circ MCell \setminus \{\mathbf{move}\}$, then:

$$\begin{aligned}
iDgen \circ MCell \setminus \{\mathbf{move}\} &= \begin{pmatrix} iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \ \Box \\ iDgen \circ C \setminus \{\mathbf{move}\} \end{pmatrix} \\
iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} &= rd?x \rightarrow iDgen \circ C(wr, x) \setminus \{\mathbf{move}\} \\
iDgen \circ C(wr, x) \setminus \{\mathbf{move}\} &= \begin{pmatrix} wr!x \rightarrow iDgen \circ C \setminus \{\mathbf{move}\} \ \Box \\ iDgen \circ C(rd, wr, x) \setminus \{\mathbf{move}\} \end{pmatrix} \\
iDgen \circ C \setminus \{\mathbf{move}\} &= iDgen \circ C(rd) \setminus \{\mathbf{move}\} \\
iDgen \circ C(rd, wr, x) \setminus \{\mathbf{move}\} &= wr!x \rightarrow iDgen \circ C(rd) \setminus \{\mathbf{move}\} \\
iDgen \circ C(rd) \setminus \{\mathbf{move}\} &= \begin{pmatrix} iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \ \Box \\ rd?x \rightarrow iDgen \circ C(x) \setminus \{\mathbf{move}\} \end{pmatrix} \\
iDgen \circ C(x) \setminus \{\mathbf{move}\} &= iDgen \circ C(wr, x)
\end{aligned}$$

By replacing each $C(...)$ expression by its value in each of the previous equations, we get:

$$\begin{aligned} iDgen \circ C \setminus \{\mathbf{move}\} &= iDgen \circ C(rd) \setminus \{\mathbf{move}\} \\ &= iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \,\Box\, rd?x \rightarrow iDgen \circ C(x) \setminus \{\mathbf{move}\} \\ &= iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \,\Box\, rd?x \rightarrow iDgen \circ C(wr, x) \setminus \{\mathbf{move}\} \\ &= iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \,\Box\, iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \\ &= iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \end{aligned}$$

and:

$$\begin{aligned} iDgen \circ C(rd, wr) &= rd?x \rightarrow iDgen \circ C(wr, x) \setminus \{\mathbf{move}\} \\ &= rd?x \rightarrow \left(\begin{array}{l} wr!x \rightarrow iDgen \circ C \setminus \{\mathbf{move}\} \,\Box \\ iDgen \circ C(rd, wr, x) \setminus \{\mathbf{move}\}) \end{array} \right) \\ &= rd?x \rightarrow \left(\begin{array}{l} wr!x \rightarrow iDgen \circ C(rd) \setminus \{\mathbf{move}\} \,\Box \\ wr!x \rightarrow iDgen \circ C(rd) \setminus \{\mathbf{move}\}) \end{array} \right) \\ &= rd?x \rightarrow wr!x \rightarrow iDgen \circ C(rd) \setminus \{\mathbf{move}\} \\ &= rd?x \rightarrow wr!x \rightarrow iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \end{aligned}$$

Then:

$$\begin{aligned} SCell &= iDgen \circ MCell \setminus \{\mathbf{move}\} \\ &= iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \,\Box\, iDgen \circ C \setminus \{\mathbf{move}\} \\ &= iDgen \circ C \setminus \{\mathbf{move}\} \\ &= iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \\ &= rd?x \rightarrow wr!x \rightarrow iDgen \circ C(rd, wr) \setminus \{\mathbf{move}\} \\ &= rd?x \rightarrow wr!x \rightarrow iDgen \circ MCell \setminus \{\mathbf{move}\} \\ &= rd?x \rightarrow wr!x \rightarrow SCell \end{aligned}$$

Let $newSBuff^n$ denote the buffer obtained after introducing the $rd \leftrightarrow wr$ chain in the expression of $SBuff^n$.

$$newSBuff^n \;\widehat{=}\; \left(\begin{array}{l} 1.SCell \,\langle\!\langle\, rd \leftrightarrow wr \rangle\!\rangle \\ (\, \langle\!\langle\, rd \leftrightarrow wr \,\rangle\!\rangle\, i : 2 \,..\, n-1 \bullet i.SCell) \,\langle\!\langle\, rd \leftrightarrow wr \rangle\!\rangle \\ n.SCell \end{array} \right)$$

Modulo the identifier, the expected static process, $Buff^n$, and the calculated one, $newSBuff^n$, are equivalent.

4.6 Discussion

4.6.1 Evaluation of results

In what follows we discuss the results presented in this chapter and their implications.

The formalisation. The concept of a *capability* has been introduced to model the notion of *knowledge of existence of a channel*, which confers no ownership of (viz. no right of access to) the channel considered, unlike the traditional notion of *knowledge of a channel* formalised by the *interface*. We have argued the necessity of the capability, in order to reason about channel mobility.

In particular, channel mobility implies that the interface of a process may change throughout its execution. The concept of a DAT (Dynamic Alphabetised Trace) captures changes of interface, and similar dynamic traces have been used by Hoare & O'Hearn [67], and Vajar et al. [132]. 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 thesis.

Another implication of channel mobility is that channels must be modelled as first class citizens (from a language point of view) or as concrete entities (of the likes of integers). This thesis shows that making channels concrete does not change the underlying model of static CSP processes.

The links. The *iDgen* transformation (viz. $dn2sn$) shows that it is possible to *simulate* channel mobility in a fixed network. For the simulation to be possible, it was necessary to extend static processes with a snapshot-identification mechanism, yielding *simulation* processes.

Simulation processes are static processes whose alphabet contains the observational variable id, and which are ***MCSns1*** and ***MCSns2*** healthy.

iDgen associates every mobile channel with a set of snapshot identifiers (viz. the set of static channels with corresponding identifiers). Each mobile channel is then blocked in snapshots that are outside its associated set of identifiers. In our original hypothesis, refusals were meant to provide the blocking mechanism. However, the blocking is provided by the healthiness condition ***MCSns1***. Since ***MCSns1*** yields the valid interface at any point of the execution, and since any channel outside the valid interface may be added into the refusals, it comes that ***MCSns1*** implicitly validates that hypothesis. The relation between DATs and refusals is discussed in greater detail in §4.6.2.

The existence of *iDgen* is beneficial for *the verification of the properties of mobile systems*, as such a verification may readily profit from existing tools for verifying static systems.

iDgen is the first transformation from mobile CSP to static CSP, and more generally, it is the first link also from a mobility framework to a static framework. Although the semantics provided are denotational, *iDgen* may readily apply to the operational semantics of CSP. The

technique used to relate mobile CSP to static CSP may be used for other frameworks too. In particular, we think that the same sort of relation exists between the pi-calculus and CCS.

Absent from this thesis is a transformation from simulation CSP back to mobile CSP, i.e. a function that can transform a $\boldsymbol{SN} \circ \boldsymbol{MCSns12}$ healthy process into a $\boldsymbol{DN}$ healthy process. $iDgen^{-1}$ denotes the inverse of *iDgen*. Although we may reasonably assume that $iDgen^{-1}$ can be defined, certain difficulties may arise. For example, *iDgen* maintains the relation between the sender and the receiver of a mobile channel through the synchronisation of their respective static equivalent. A design question would be that of the necessity of keeping such a synchronisation action.

Hiding. Our treatment of hiding is quite novel since we allow the set of hidden channels, say X, to grow *ad infinitum*. This is a sensible choice, because we leave out the possibility for X to shrink. However, this is a healthy choice in the following sense. In a static network, the actual set of silent names (i.e. names that will assuredly appear in a trace) is given by $X \cap \mathcal{A}$. This translates naturally to mobile systems viz. in every snapshot $X \cap mChans$ will contain *hidden* channels only, hence X does not need to shrink since *mChans* might.

In order to define the dynamic hiding operator it was first necessary to compute the set of silent channels. This has led us to define a new class of mobile processes, *silent* processes, which have the variable *sil* in their alphabet, and are ***S1***, ***S2***, ***S3*** and ***S4*** healthy. Healthy silent processes formally characterise processes that have a leftmost hiding suffix $\setminus X$, which determines the value of *sil* to be used by the dynamic hiding operator $\setminus \{\bullet\}$. This operator behaves like traditional hiding.

The dynamic operator is original to this thesis. The hiding operator has been defined in the context of channel mobility in [56], and [123]. In [56], it relates to the set of private names, which may not even be used internally to begin with. In [123], they consider the environment as any other component, thus they hide channels from the global interface, as in static hiding. However, they distinguish for each process its internal interface from its external interface, and then impose that each process should have a disjoint internal interface. Whilst they permit the external interface to grow, the internal interface remains static. Hence, the separation between internal and external interface only serve to avoid *name collision*, such that an internal name may not appear in the trace. In comparison, we allow the internal interface of every process to grow, and do not explicitly separate internal and external interfaces. Nonetheless, when computing the *hiding normal form* of processes, we have seen how the operators enforce such a separation.

Our treatment resolves many issues related to hiding in the context of channel mobility. In particular, we resolve the ambiguities related to the concepts of *internal* and *external mobility*. Hiding is meant for hiding, so it is only concerned with the set of silent names, and not with their eventual mobility; whereas mobility is concerned with movement, without care for visibility. If a silent name is moved out by a process, there is *no need* for actually hiding the channel carrier used for the movement: observing a name as a message (viz. the subject of a communication), and observing the same name as a channel (viz. the element of an

interface) are two separate things. Since channels are localised in interfaces, or equivalently that an interface is the location for channels, *channel mobility is the movement from an interface to another*. This means that if a communication $ch.M$ is in the trace with M some channel, if M does not appear in the interface (which is the case when $M \in X$), no observer may effectively infer that M is indeed a channel.

In [111] the authors investigate the possible relationship between CCS and the pi-calculus. They distinguish internal from external mobility, leading them to define a specific characterisation of internal mobility for the pi-calculus, called πI (see also [20]). Their treatment of external mobility yields more complex formulations than πI. It would be interesting to see how our approach could affect their results, but that is a topic for future work.

Renaming. The renaming operator was redefined in order to prevent the renaming of names not in the interface yet. More generally, what the dynamic renaming operator has highlighted is the fact that, for any operator that operates on channel names (including the hiding operator), any assumption about channel names in static CSP may potentially become a healthiness condition in mobile CSP.

It is interesting to compare the dynamic hiding and the dynamic renaming operators. In order to define the first, we had to build a new theory, to apply healthiness conditions to every mobile process. For dynamic renaming, it was necessary to apply a healthiness condition on the operator itself (viz. substitute names must *not* coincide with new names).

Another point of comparison is the following. Both operators may be seen as functions on names, hence from their perspective, there is no notion of known and unknown names, only defined and undefined names. Hiding evaluates names given by the variable *sil*, renaming those given by the variable *mChans*. Yet, hiding applies only at the end of the execution of a process, whereas renaming applies somewhat independently of any execution. This difference may have some theoretical importance. In particular, one may view hiding as a kind of renaming in which given names are replaced by a given silent name τ. τ is used in the pi-calculus and is not 'mobile', seeing that it is not associated with any channel. Thus, it would be interesting to study the correctness of this τ-renaming view of the hiding operator, which may be used in relating CCS with CSP. For example, that view is mentioned in [107] by Roscoe for relating CCS and CSP.

Our formulation of the dynamic renaming operator may be used as a model for Roscoe's generalised relabelling operator [108] (cf. Chap. 3, §3.2.6). It has the advantage of being a function, hence it may more easily be reused.

4.6.2 Of the relation between the alphabetised traces model of simulation CSP and the failures model of CSP

In Section 4.5.2 we have discussed the construction of *iDgen*, meant to transform a dynamic trace into a static trace. Our initial idea was to put channels that must not be used in a given snapshot in the refusals of that snapshot. Meanwhile, we gained the insight that we could

translate DATs (for mobile CSP) into a new form of DATs for simulation CSP. Thus, *iDgen* would now transform a dynamic trace of form $\langle ..., (s, e), ... \rangle$ into a simulation trace of form $\langle ..., (id \mapsto s, e), ... \rangle$, instead of a static trace of form $\langle ..., (\mathcal{A}, e), ... \rangle$. Then, we have defined the healthiness condition ***MCSns1*** that ensures that only specified channels in $id \mapsto s$ may effectively be used. Clearly, ***MCSns1*** provides the guarantee that we were trying to build with refusals.

The failure of a CSP process is given by a pair (t, X) where t is a trace of the process and X is a set of events denoting a refusals set. The set F of all pairs (t, X) denotes the failures of the process. Cavalcanti & Woodcock [35] have defined four healthiness conditions for a correct failures model. The following healthiness conditions are particularly interesting:

$$\textbf{\textit{F1}} \quad traces_{\perp}(P) \;=\; \{t \mid (t, X) \in F\} \quad \text{is non-empty and prefix closed}$$
$$\textbf{\textit{F3}} \quad (t, X) \in F \wedge (\forall\, a : Y \bullet t \frown \langle a \rangle \notin traces_{\perp}(P)) \Rightarrow (t, X \cup Y) \in F$$

where $traces_{\perp}(P)$ denotes the set of all traces in which P can engage, including those that lead to or arise from divergence. ***F1*** states that the set of traces of a process must be captured in its set of failures, and is non-empty and prefix closed. ***F3*** states that if an event is not possible according to the set of traces of the process, then it must be in the set of refusals.

Whilst ***MCSns1*** does not enforce ***F3*** explicitly, it seems to be doing that implicitly, since elements from $\mathcal{A} - s$ may not possibly be recorded. We had discarded ***MCSns3*** as it required that $ref' \subseteq mChans'$, the future valid interface. However, if we substitute the previous relation by $(\mathcal{A} - mChans') \subset ref'$, we obtain the effect described by ***F3*** explicitly, expressed as the following healthiness condition:

$$\textbf{\textit{MCSns3.2}} \quad P \;=\; P \wedge (\mathcal{A} - mChans') \subset ref'$$

By definition, we have $iDgen \;=\; \textbf{\textit{MCSns3.2}} \circ iDgen$.

Cavalcanti & Woodcock "*view the definition of extra healthiness conditions on UTP processes to ensure* ***F1*** *and* ***F3*** *as a challenging task*" ([35, §7.3]). Although that remains to be proven, we emit the following hypothesis:

$$\textbf{\textit{F1}} \circ \textbf{\textit{F3}} \circ \textbf{\textit{MCSns1}} \circ \textbf{\textit{MCSns2.2}} \;=\; \textbf{\textit{MCSns1}} \circ \textbf{\textit{MCSns2.2}}$$

4.6.3 Versus the pi-calculus

The approach to semantics used in this thesis (and in some related works that use CSP as a basis) is strikingly different from the one used in the pi-calculus [86]. We have only been able to distil some preliminary elements of comparison. Our general impression is that the question of the relation between CSP and the pi-calculus has not been adequately expressed so far in the Literature. Our intuition is that the definition of that relation would require an adequate algebraic basis, which should notably formalise also the notion of process capability introduced in this thesis. We have suggested de Simone [120] framework as the sort of

algebraic characterisation we have in mind.

Assuming the existence of such an algebra, we see that operational semantics and denotational semantics are functions on processes. Then, the notions of capability and of interface, which are independent from the style of semantics, should certainly be affected by the same sort of changes whenever the function considered (either operational or denotational) defines the same process. Popescu [101] coalgebraic framework clearly uses a notion of dynamic interface even for characterising pi-calculus processes.

Therefore, if we add the notion of capability to our conceptual algebra, we see that the problem of channel-passing is not simply that of name freshness, but that of dynamic interface. Whilst the pi-calculus may use scope extrusion, it would still be necessary that the effect of scope extrusion is a change of interface (at the algebraic level). We may well suppose another model in which the notion of a *process context* characterises an interface as well, so that the question is no longer to what value a name is bound, but rather whether a name is in the current interface.

This means that we have moved away from the question of *name binding* to that of *dynamic interface*. Certainly, even in the pi-calculus, the programmer expects any channel that appears in a process expression to be in the current interface. The departure from *name binding* is we believe, the *key problem* posed by trying to build traces for pi-calculus processes. Indeed, consider a pi-calculus process, and suppose that we want to build the traces for such a process. Naively, we may simply record whatever name appears in a transition, as may be done when building traces for CCS processes [25]. The question thereon is thus: **how can we from the previous trace deduce channel mobility?**

One possible answer is to try and capture name binding in the trace. Such is the approach employed by Bialkiewicz and Peschanski [18], and by Roscoe [108]. This approach results in quite complex models, which nonetheless leave certain questions unanswered, e.g. what makes a name authorised?, and make others difficult to answer, e.g. how does the traces model (for channel mobility) relate to static CSP?

Another possibility and the one that we advocate is to use both a static capability and a dynamic interface. Then, the effect of scope extrusion would simply be a change of interface, notwithstanding whether one uses operational semantics or denotational semantics. Our naive axiomatisation at the end of the Literature review, together with works such as Popescu's [101], and different works on traces for CCS, and operational semantics for CSP, give us a strong intuition about the feasibility and correctness of such a model. Our proposition does not apply only to relate mobile CSP with the pi-calculus, but also CCS and related frameworks with the pi-calculus.

4.6.4 Closed vs. Open world

In this thesis, we have encountered two operators that deal with names that may be either known in advance or not. The case of known names is a particular case of unknown

names. Clearly, in either case, if we send to a process P a channel kn that is known, P will receive kn. We are just making a *projection* here, from an *unknown-world* (i.e. a world/universe/environment with unknown names) to a *known-world*: inasmuch as P may not know in advance a name that it must acquire, the environment always knows in advance what name it may send to P —channel-passing output prefix states exactly that. Therefore, any renaming of a channel ch_1 in a known-world environment applies also to P, which is in unknown-world instead.

Let $\mathcal{E}$ denote an environment (a process) such that $ch_1 \in \mathcal{I}\mathcal{E}$ is always true. Then, $\mathcal{E}[ch_2 \leftarrow ch_1]$ denotes an environment in which $ch_2 \in \mathcal{I}\mathcal{E}[ch_2 \leftarrow ch_1]$ is always true. Further suppose that $\mathcal{E}$ may successfully send channel ch_1 to P, then $\mathcal{E}[ch_2 \leftarrow ch_1]$ may successfully send channel ch_2 to P. Then:

$$[SpecRename2] \quad (P \parallel \mathcal{E})[ch_2 \leftarrow ch_1] = P \parallel \mathcal{E}[ch_2 \leftarrow ch_1]$$

[*SpecRename*2] may have some theoretical importance: it suggests that *we can always simulate unknown-world semantics by known-world semantics.*

Definition 4.6.1 (Unknown-, known-world semantics)**.** *A mobile process P is said to be in* open/unknown-world *(of names) semantics if P may acquire a new channel, and that channel may not be known in advance, or equivalently, the environment may send P a name undetermined in advance.*
If, however, it is known in advance what channel P may receive, for example when the interface of the sending process is known, P is said to be in closed/known-world *semantics. Typically, static* **SN***-healthy processes define a known-world semantics, in which a process may receive only those names that are already in its interface.* □

Theorem 4.6.2 (unknown-to-known refinement)**.** *If P is in unknown-world semantics, and we send to P a name kn, then P will receive kn. Hence, we may always simulate an unknown-world semantics by a known-world semantics. This is a refinement result: let $ukn(P)$ denote the unknown-world semantics of P, and $kn(P)$ denote a given known-world semantics of P, then*

$$ukn(P) \sqsubseteq kn(P)$$

In consequence, whatever theorem holds in known-world semantics does hold also in unknown-world semantics. □

Unlike what may be suggested by the names closed/open (resp. known/unknown), it appears that the concept of open world does actually apply to a single process: it traduces a *local* view of the environment; whereas the concept of closed world applies to all the interacting processes in the environment/system: it traduces a *global* view of the environment.

Chapter 5

Strong Process Mobility

5.1 Introduction

Process mobility refers to any model or theory that describes the movement of a process from its initial computational environment (or source) to another computational environment (or target). Two forms of process mobility may be distinguished:

- weak mobility: here, only the *code* of the process is moved, possibly including initialisation data.
- strong mobility: here, a process's execution is interrupted, then its code and interrupt state are moved to the target location where the execution is resumed from the interrupt state.

Tang & Woodcock [126] have given the semantics for weak mobility in UTP. We extend their results with semantics for strong mobility. The formalisation of process mobility in UTP is quite challenging as it requires to:

1. model the program counter in order that its value may be included into the interrupt state: This calls for the concept of *continuations*, hence, for an explicit representation of *control flow* as a function of continuations. Whilst Hoare & He [66, Chap. 6] and Woodcock & Hughes [141] have given semantics for programs with continuations in UTP, we show that their models are insufficient for reasoning about *nested* parallel programs and propose a new solution.
2. record the state of a process, explicitly: One possibility is to encapsulate the state into an action representing state transformation, but this may introduce some changes to the underlying UTP-CSP model. We propose a solution that does not rely on recording the state in some extra variable (including the trace, and for example, explicitly saving the state in a third process). Instead, our solution makes the interrupt state observable, say like waiting states, thus making it available for the following process in a sequential composition.

3. interrupt a process: Different semantics may be given to the interrupt operator of which two are particularly interesting. The catastrophic interrupt was given a UTP semantics by McEwan & Woodcock [84] as a form of sequential composition $P \triangle_i Q$ where the interrupting process Q may execute before P's termination, whenever P is in a waiting state if the interrupt event i does occur. The nature of the catastrophic interrupt makes it easy for Q to access the interrupt state of P. The generic interrupt was given a UTP semantics by Kun Wei [135] as a form of parallel composition $P \triangle i \rightarrow Q$ where the execution of Q may overtake that of P whenever the interrupt event i occurs, and not just when P is in a waiting state. However, K. Wei's definition does not permit saving the state of P upon interrupt. We provide a new semantics for the generic interrupt which permits saving the interrupt state of an executing process.

We model each of these elements independently and in turn. The results constitute the content of sections 5.2, 5.3 and 5.4. Process mobility itself is then defined as a particular form of interrupt operator such that the interrupting process is responsible for the movement of the interruptible process. In the circumstance we may rather use the terms *moving process* and *movable process*. This is the content of §5.5. Finally, §5.6 presents a discussion of our results.

5.2 Continuations for Reactive Processes

5.2.1 Formalisation

Note: for ease, we will refer to Hoare & He semantics for continuations in UTP presented in Chap. 2 as HH98 steps or simply HH98. Similarly, we will refer to the work in [141] as WH02 steps or simply WH02.

The Continuation-Passing-Style transformation (or compiler) is inherently sequential [103], [124]. UTP-CSP processes also permit the representation of sequential programs, which form a subset of the class of reactive programs. This suggests that HH98 may be applied at least to sequential UTP-CSP. All that is needed is to extend the alphabet of UTP-CSP sequential processes, and point-wise extend the definition of sequential composition to the control variable l, as suggested in HH98.

For illustration, let us adopt a more functional-like notation. Let $(\alpha P, \varnothing, P)$ (or simply $(\varnothing, P)$) denote a predicate P with no continuations, and $(\alpha P, \alpha lP, P)$ (or simply $(\alpha lP, P)$) denote the same but with continuations $(\alpha lP \neq \{\})$. Then, for UTP-CSP processes,

$$seq(P, Q) = seq((\varnothing, P), (\varnothing, Q)) = P \fatsemi Q$$

and when extended with continuations,

$$\begin{aligned} seq(P, Q) &= seq\big((\alpha lP, P), (\alpha lQ, Q)\big) \\ &= \big((l \in \alpha lP)^{\top} \fatsemi P \fatsemi (l \in \alpha lQ)_{\perp}\big) \fatsemi \big((l \in \alpha lQ)^{\top} \fatsemi Q \fatsemi (l \notin \alpha lQ)_{\perp}\big) \end{aligned}$$

The previous extension is trivially correct for sequential processes whose expression contains neither interaction nor parallel operators. This raises the question of the possibility of such an extension to parallel programs also.

The case of parallel composition is less trivial. Indeed at first, it may be tempting to use the equivalence $P \parallel Q = (P \fatsemi Q) \vee (Q \fatsemi P)$. However, this is not quite right when continuations are involved. To see this, let us again adopt a functional-like notation, and attempt a point-wise extension of parallel composition in the same manner as sequential composition above.

$$\begin{aligned} par(P, Q) &= par\big((\alpha lP, P), (\alpha lQ, Q)\big) \\ &= seq\big((\alpha lP, P), (\alpha lQ, Q)\big) \vee seq\big((\alpha lQ, Q), (\alpha lP, P)\big) \end{aligned}$$

A first theoretical difficulty arises: $seq\big((\alpha lP, P), (\alpha lQ, Q)\big)$ supposes that P knows of the continuations of Q. This contradicts parallel composition, which supposes that P and Q are executing on distinct, possibly remote processors and hence need not be aware of each other (except maybe through interaction). We may however do away with such a consideration by introducing some third process, say T, in charge of facilitating the sequential transit between P and Q. Then

$$\begin{aligned} par(P, Q) &= par\big((\alpha lP, P), (\alpha lQ, Q)\big) \\ &= seq\big((\alpha lP, P), (\alpha lT_1, T_1), (\alpha lQ, Q)\big) \vee seq\big((\alpha lQ, Q), (\alpha lT_2, T_2), (\alpha lP, P)\big) \end{aligned}$$

$$seq\big((\alpha lP, P), (\alpha lT_1, T_1), (\alpha lQ, Q)\big) = \begin{pmatrix} (l \in \alpha lP)^{\top} \fatsemi P \fatsemi (l \in \alpha lT_1)_{\perp}) \fatsemi \\ ((l \in \alpha lT_1)^{\top} \fatsemi T_1 \fatsemi (l \in \alpha lQ)_{\perp}) \fatsemi \\ \big((l \in \alpha lQ)^{\top} \fatsemi Q \fatsemi (l \notin \alpha lQ)_{\perp}\big) \end{pmatrix}$$

$$seq\big((\alpha lQ, Q), (\alpha lT_2, T_2), (\alpha lP, P)\big) = \begin{pmatrix} (l \in \alpha lQ)^{\top} \fatsemi Q \fatsemi (l \in \alpha lT_2)_{\perp}) \fatsemi \\ ((l \in \alpha lT_2)^{\top} \fatsemi T_2 \fatsemi (l \in \alpha lP)_{\perp}) \fatsemi \\ \big((l \in \alpha lP)^{\top} \fatsemi Q \fatsemi (l \notin \alpha lQ)_{\perp}\big) \end{pmatrix}$$

Using single instructions provides a simpler illustration. Let $\langle s, P, f\rangle$ and $\langle t, Q, g\rangle$, then

$$par(\langle s_1, P, f_1\rangle, \langle s_2, Q, f_2\rangle) = \begin{pmatrix} seq(\langle s_1, P, f_1\rangle, \langle f_1, T_1, s_2\rangle, \langle s_2, Q, f_2\rangle) \vee \\ seq(\langle s_2, Q, f_2\rangle, \langle f_2, T_2, s_1\rangle, \langle s_1, P, f_1\rangle) \end{pmatrix}$$

Before discussing the more complex case of basic blocks, remark that the definition above omits alphabet constraints. This is voluntary, for readability purpose. Indeed, it would be

necessary to extend the alphabet of P with that of Q, and inversely, and also the alphabet of T with those of both P and Q.

More importantly, the continuations of T have been chosen quite purposefully. A complete definition of T would actually require some mechanism for relocating the final continuation of P. The subsequent discussion will show how impractical an attempt of building such a mechanism may be.

For basic blocks i.e. sequences of instructions $P = \boxplus P_i$ and $Q = \boxplus Q_j$, the definition is more complex. The rationale is that each step is executed upon a non-deterministic choice between the rest of P and the rest of Q. For readability, we may introduce yet more notation. Let $\bar{P}_i$ denote the rest of the computation after the execution of a given step P_i. We assume that $\bar{P}_i = SKIP$ if P_i is the last instruction to be executed. Then

$$par(P, Q) = seq(P_1, par(\bar{P}_1, Q)) \sqcap seq(Q_1, par(\bar{Q}_1, P))$$

Consider the simple case where $P = seq(P_1, P_2)$ and $Q = seq(Q_1, Q_2)$. Then, e.g. $\bar{P}_1 = P_2$. We want to compute the value of $par(P, Q)$, and in particular, see if we obtain the expected interleaving of the instructions of P and Q. In what follows, we use a single generic process T to denote the transition between P and Q, as discussed previously. Then

$$\begin{aligned}
par(P, Q) &= seq(P_1, T, par(P_2, Q)) \sqcap seq(Q_1, T, par(Q_2, P)) \\
seq(P_1, par(\bar{P}_2, Q)) &= P_1 \fatsemi T \fatsemi par(P_2, Q) \\
&= P_1 \fatsemi T \fatsemi (seq(P_2, T, Q) \sqcap seq(Q_1, T, par(Q_2, P_2))) \\
&= P_1 \fatsemi T \fatsemi \left(\begin{array}{l} (P_2 \fatsemi T \fatsemi Q_1 \fatsemi Q_2) \sqcap \\ (Q_1 \fatsemi T \fatsemi (seq(Q_2, P_2) \sqcap seq(P_2, Q_2)) \end{array} \right) \\
&= \left(\begin{array}{l} (P_1 \fatsemi T \fatsemi P_2 \fatsemi T \fatsemi Q_1 \fatsemi Q_2) \vee \\ (P_1 \fatsemi T \fatsemi Q_1 \fatsemi T \fatsemi seq(Q_2, P_2)) \vee \\ (P_1 \fatsemi T \fatsemi Q_1 \fatsemi T \fatsemi seq(P_2, Q_2)) \end{array} \right)
\end{aligned}$$

Let us eliminate T in the development above, as it behaves like *SKIP* with regard both the alphabets of P and Q. Then, we obtain all the expected interleaving. Also, we remark that in sequences of the form $P_i \fatsemi Q_j$, Q_j behaves like *SKIP* with regard to the alphabet of P, and similarly for sequences of the form $Q_j \fatsemi P_i$, P_i behaves like *SKIP* with regard the alphabet of Q. This leads to the conjecture that any given interleaving is an instance of $P \wedge Q$.

The latter conjecture simply means this: *l is not expressive enough for reasoning about control flow in the presence of parallelism.* The problem actually lies with the design of the control variable as *single-valued*. Possibly, it may be obvious at a first glance that l does not follow the structure of programs. However, one may still attempt a definition based on l: as far as mathematics are concerned, that does not seem to be impossible. The last conjecture shows that such an endeavour would be pointless: since the intended effect of $P \parallel Q$ is *essentially* $P \wedge Q$, it would be better to compute the continuations of P and Q individually

from the start. We need a mathematical model that follows more tightly the computation model. For example, using l in the presence of an interrupt operator, it would be as if a single program was interrupted at a time whereas we should be able to say that many programs may be interrupted at a time.

Nature of the control variable in UTP. Beyond what has just been said, an interesting insight may be derived from the previous discussion, concerning the semantics of l, in comparison with functional continuations as defined in e.g. [128], [124]: although l is a variable, inasmuch as it determines the next instruction to be executed, it stands for the instruction that it designates. Hence, l must be understood as a functional continuation, or in UTP terminology as a predicate, and not just as some variable containing execution locations, as if the locations were distinct from the predicates localised. Hence, execution locations in the context of UTP *must be confused* with the predicates that the locations serve to designate; and the fact that l is a 'variable' simply characterises that the instruction to be executed next may vary or change.

The solution to the limitations of l mentioned above is to design a value of the control variable that follows more tightly the structure of processes. This is what is done in [141].

In [141] (hereafter also WH02), Woodcock & Hughes use a *set-valued* control variable, denoted ls instead, which contains the continuations of all the steps that may be executed in parallel next. Thanks to ls, we may point-wise extend the UTP-CSP parallel composition operator. However, a number of changes must first be considered. Unlike HH98 steps, a WH02 step may now exit at many points at any one time, implying that a step may be entered simultaneously at multiple entry points. This is a little counter-intuitive but poses no great difficulties.

Definition 5.2.1 (WH02 Step [141])**.** *A predicate P is a* step *if $l \in \alpha P$ and*

$$P \;=\; P \lhd \exists\, l_0 \in \alpha lP \bullet l_0 \in ls \rhd II \qquad \square$$

Unfortunately, ls is not quite sufficient for our purpose. To see this, consider the following illustration. Let $P = seq(\langle s, P_1, h\rangle, \langle h, P_2, f\rangle)$, and let $ls = \{s, h\}$. The value of ls is still valid but does not reflect the structure of P. If the programmer was expecting parallel composition, sequential composition will be performed instead, which is an error and will not be detected. Let $Q = par(\langle s, Q_1, f\rangle, \langle t, Q_2, g\rangle)$, and let $ls = \{s\}$. Then Q will behave like Q_1, since Q_2 behaves like II (by definition). Again, if parallel composition was expected then only one step will be executed instead of two in parallel, which is an error and will not be detected.

Seeing that neither HH98 variable l nor WH02 variable ls are adequate, we have to design a new value for the control variable. $\mathcal{L}$ will denote the new control variable, and we discuss its formalisation in the next section.

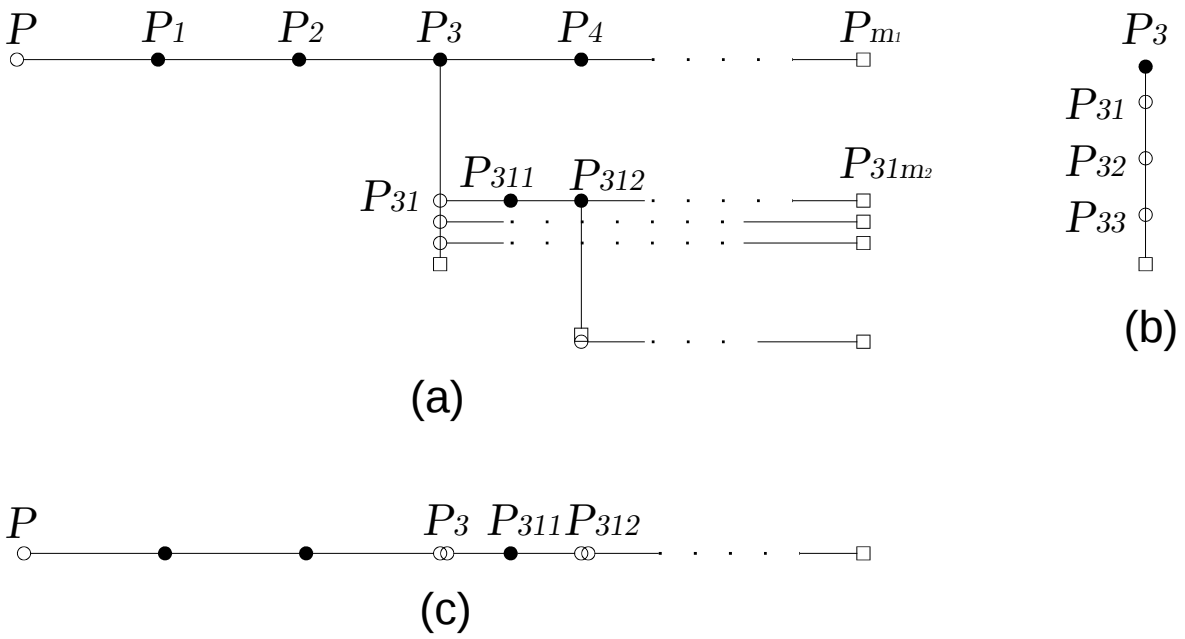

Figure 5.1: Example of a CFG for reactive processes

Design of the control variable $\mathcal{L}$

Parallel composition may be seen as a single block such that when entered sequentially, the steps that compose the block are executed in parallel, and when they have all exited, then the block is also exited. That is, entry into (resp. exit from) a block of parallel steps is identical to entry into (resp. exit from) a sequential block. Sequential and parallel blocks would hence differ in their respective execution order: for the first, only one step may be executed at a single (observation) time, whilst multiple steps may be executed at a single time for the second. In other words, parallel composition acts as an *envelop* w.r.t. its components. It has its own continuations, that differ from those of its constituents. Let $P = par(P_1, P_2)$, then the block denoted by P differs from its component blocks P_1 and P_2: P has its own entry and exit points that differ from those of P_1 and P_2. P will be called a *nesting* step, and P_1 and P_2 will be called *nested* steps.

A *control flow graph* (CFG) is a standard representation of programs with no parallel constructs, using a graph. A CFG and related concepts are appropriate for discussing the structure of UTP-CSP processes. Note that we are not interested in a graphical formalism, but only to use graphs as an adequate means for discussion. In what follows, we sketch what such a graph might look like.

Construction of a CFG for reactive processes. *Figure5.1(a)* shows an example of such a graph read in a left-right, then top-down, iterative manner, thus indicating the flow of control. P_i nodes may denote either single instructions, sequential blocks, or nested (parallel) blocks. Both the root node (P) and initial nodes (e.g. P_{31}, P_{32}, P_{33}) are indicated by empty circles. A nesting node (e.g. P_3) is indicated by a vertical line starting from the

node downwards, as shown in *Figure5.1(b)*. An empty square indicates termination for a horizontal line, whereas it simply serves as a visual aid to indicate the end of a vertical line. A flattened graph *Figure5.1(c)* shows how control goes through P_3, and then again through P_{312}. More information could have been added for loops and jumps, and bigger graphs may be conceived, but such are not our main interest. Rather, we may also annotate nodes with their continuations. The annotation procedure would then show how to evaluate the control variable.

Value of $\mathcal{L}$

Let $\mathcal{L}$ denote the control variable whose value we will be discussing. Then $\alpha\mathcal{L}P$ denotes the continuations of a step P.

To formalise the *nesting relation* between a parent and its children, we may partition the continuations set of every node into two subsets: αl, the continuations of the parent, and αls, the continuations of its children. We make the following important remark: the parent-child relation does not extend beyond two adjacent levels. Hence αls contains the continuations of nodes at the lower adjacent level only, e.g. for sequential blocks, $\alpha ls = \{\}$.

In what follows, we describe in detail the procedure for attributing continuations to nodes. That is also the procedure for computing the value of $\alpha\mathcal{L}$ for a given block.

Continuations are attributed hierarchically, in a bottom-up fashion. We make no difference between nodes denoting either single instructions or sequential blocks, and we will refer to them commonly as $\mathbf{lv_0}$ (read level-0) nodes. Such nodes do not introduce nesting, hence they have no children, i.e. $\alpha ls = \{\}$.

We then put in parallel $\mathbf{lv_0}$ nodes, exclusively, to form **lv1** nodes. Such nodes correspond to the nesting nodes mentioned earlier. The value of αls is given by the union of continuations αl of its constituents; e.g. $\alpha lsP_3 = \{\alpha lP_{31}, \alpha lP_{32}, \alpha lP_{33}\}$.

Again, putting exclusively $\mathbf{lv_1}$ nodes in parallel, or together with $\mathbf{lv_0}$ nodes, we obtain $\mathbf{lv_2}$ nodes. αls is the union of all the continuations of *adjacent* $\mathbf{lv_1}$ (and $\mathbf{lv_0}$) nodes, only. Hence, the value of αls for a $\mathbf{lv_2}$ node does not contain the continuations of those $\mathbf{lv_0}$ nodes that are nested to $\mathbf{lv_1}$ nodes; e.g. $\alpha lP_{312x} \nsubseteq \alpha lsP_3$, although $\alpha lP_{312} \subseteq \alpha lP_{31} \subset \alpha lsP_3$ & $\alpha lsP_{312} = \{..., \alpha lP_{312x}, ...\}$.

This illustrates what we said earlier about αls: it contains only the continuations of the lower adjacent levels. We reiterate this construction procedure for higher-levelled nodes.

The value of $\alpha\mathcal{L}$ may be obtained by iteration on the level of a node considered as the root (of the graph), as follows:

$\mathbf{lv_0}$ root, no children: αlP & $\alpha lsP = \{\}$ & $\alpha\mathcal{L}P = \alpha lP$

$\mathbf{lv_1}$ root or parent, $\mathbf{lv_0}$ children only: $\alpha lsP = \bigcup_i \alpha lP_i$ & $\alpha\mathcal{L}P = \alpha lP \cup \alpha lsP$

$\mathbf{lv_2}$ root or parent, at least one $\mathbf{lv_1}$ child: $\alpha lsP = \bigcup_i \alpha lP_i \ \& \ \alpha\mathcal{L}P = \alpha lP \cup (\bigcup_i \alpha\mathcal{L}P_i)$

$\mathbf{lv_n}$ root or parent, at least one $\mathbf{lv_{n-1}}$ child: $\alpha lsP = \bigcup_i \alpha lP_i \ \& \ \alpha\mathcal{L}P = \alpha lP \cup (\bigcup_i \alpha\mathcal{L}P_i)$

Note: the introduction of a *nesting* step is essentially what distinguishes the value of $\mathcal{L}$ from that of WH02' control variable *ls* from [141]. Its effect is to delegate the instantiation of parallel (nested) nodes to the nesting node, which is a dummy. Thanks to this, control flows as in sequential programs, since the dummy node hides away the parallel structure of programs. It is also thanks to the nesting node that we solve the limitations of *ls* discussed earlier. For example, using WH02 steps, it is possible to jump to a step without care for its nesting level. The presence of the dummy step resolves this by imposing that control must enter into the dummy step first before it can then enter into the parallel steps.

In what follows we describe the semantics of $\mathcal{L}$ formally.

5.2.2 Continuations semantics for programs with parallel constructs

HH98 steps (cf. §2.6) are programs that compute the control variable l. By analogy, we present programs that compute the control variable $\mathcal{L}$ instead. We follow the same methodology of Hoare and He [66, Chap. 6] that consists of starting with unstructured predicates (i.e. steps) and then adding more structure to obtain in turn target code programs, and then program blocks. In our case, after (re)defining steps, we shall restrict our programs to Reactive Processes and obtain, as a result, the theory of Reactive Process Blocks (cf. §5.2.3) i.e. reactive processes that contain the control variable $\mathcal{L}$.

Steps

We now describe predicates whose alphabet include a set of continuations denoted by $\alpha\mathcal{L}$. $\alpha\mathcal{L}$ is partitioned into two subsets: αl, which contains the continuations at the current level of execution, and αls, which contains the continuations at the adjacent lower level of execution, w.r.t. nesting.

At first, each level of execution may be considered without regard for nesting. Then, every step is entered horizontally, and exits horizontally. In a graph, a level corresponds to a single horizontal line that links nodes arranged from left to right, according to their execution order. There is a node that has no horizontal predecessor, called the root of the level. Each node on a line is adjoined a continuation. We say that a node is entered horizontally if we can draw a horizontal line from the root leading to it viz. the value of $\mathcal{L}$ corresponds to the node's continuation.

In the case of nesting, in a graph, there is a vertical line linking the higher level, at the top, with its adjacent lower levels, all arranged as parallel horizontal lines. The root of the graph has neither vertical nor horizontal predecessors (i.e. there is no vertical/horizontal line leading to the graph-root); the root of a lower level has no horizontal predecessor and should have at least one vertical predecessor. A lower level (or child) node may be entered only if its

parent has been entered first. That is, we can draw a vertical line from the parent node to the lower level horizontal line that contains the given child node, when traversing the graph of the step from its root to the given node. In other words, the value of $\mathcal{L}$ must hold both the parent and the child nodes continuations.

Definition 5.2.2 (Step(2))**.** *Let P be a predicate describing a step. Let $\alpha\mathcal{L}P$ denote its set of continuations, and let $\mathcal{L}$ be the control variable for its execution. We may partition the set $\alpha\mathcal{L}P$ into two subsets αlP and αlsP such that:*

- *αlP denotes the set of all the continuations of P at a single level of execution.*
- *αlsP denotes the set of all the continuations of P at the adjacent lower level of execution.*

Control may enter into a step horizontally with regard to its own execution level, or vertically with regard to nesting. In either case, a step may only be entered when the value of $\mathcal{L}$ coincides with one of the step's entry points. Otherwise the step does nothing. Formally,

$$P = P \lhd \mathcal{L} \in \alpha\mathcal{L}P \rhd II$$ □

Some operators induce/embed a nesting relation (cf. below, e.g. parallel assembly) whilst others do not.

Definition 5.2.3 (Nesting relation)**.** *Let P be a step, and* **op** *an operator on steps and which is closed.*
op *is said to* induce nesting *if, and only if, $\alpha l\,\mathbf{op}(P) \neq \alpha lP$ and $\alpha lP \subset \alpha ls\,\mathbf{op}(P)$: then, we say that $\mathbf{op}(P)$ is the* parent *of P, and is called a* nesting step*; or equivalently, we say that P is* nested *into $\mathbf{op}(P)$, and is called a* nested step.
Otherwise, i.e. if $\alpha l\,\mathbf{op}(P) = \alpha lP$, then **op** *does not induce nesting.* □

The value of $\alpha\mathcal{L}P$ may only be given by recursion over the nesting level of P.

Definition 5.2.4 ($\mathbf{lv_k}$-steps, $\alpha\mathcal{L}$)**.** *Let P be a step, then*

$$\alpha\mathcal{L}P \;\widehat{=}\; \alpha lP \cup \alpha lsP$$

where both αlP and αlsP are specified according to the level of the nested programs in the expression of P, as described subsequently.
We say that a program P is a $\mathbf{lv_0}$-step, denoted by $P = \mathbf{lv_0}(P)$, if, and only if, P has neither parent nor children, i.e. $\alpha lsP = \{\}$. Then

$$\alpha\mathcal{L}P \;\widehat{=}\; \alpha lP \cup \alpha lsP = \alpha lP$$

Let **op** *be a binary operator that induces nesting. Then:*

- *if P and Q are both $\mathbf{lv_0}$-steps, then we say that $\mathbf{op}(P, Q)$ is a $\mathbf{lv_1}$-step and*

$$\alpha l\,\mathbf{op}(P, Q) \;\widehat{=}\; \{\mathbf{nn}\} \qquad \alpha ls\,\mathbf{op}(P, Q) \;\widehat{=}\; \alpha\mathcal{L}P \cup \alpha\mathcal{L}Q = \alpha lP \cup \alpha lQ$$

- *if either P or Q is a $\mathbf{lv_1}$-step, or both are, then we say that $\mathbf{op}(P, Q)$ is a $\mathbf{lv_2}$-step and*

$$\alpha l\,\mathbf{op}(P, Q) \;\widehat{=}\; \{\mathbf{nn}\} \qquad\qquad \alpha ls\,\mathbf{op}(P, Q) \;\widehat{=}\; \alpha\mathcal{L}P \cup \alpha\mathcal{L}Q$$

- *if either P or Q is a $\mathbf{lv_k}$-step, or both are, then we say that $\mathbf{op}(P, Q)$ is a $\mathbf{lv_{k+1}}$-step and*

$$\alpha l\,\mathbf{op}(P, Q) \;\widehat{=}\; \{\mathbf{nn}\} \qquad\qquad \alpha ls\,\mathbf{op}(P, Q) \;\widehat{=}\; \alpha\mathcal{L}P \cup \alpha\mathcal{L}Q$$

where $\mathbf{op}(P, Q)$ is a nesting step and may have only one entry point, and only one exit point, both denoted by $\mathbf{nn}$ for convenience. □

Consequence 5.2.5.

1. $\mathbf{lv_0}$*-steps* do not *induce a nesting relation.*
2. $\mathbf{lv_0}$*-steps are* ⧉*-closed (cf. Theorem 2.6.4). Hence, every operator that may be defined in terms of* ⧉ *(such as* $\{\lhd b \rhd, \sqcap, ⨟\}$*)* does not *induce a nesting relation.* □

The relation with HH98 steps is obvious:

Example 5.2.6. *HH98 steps are $\mathbf{lv_0}$-steps.*

Proof. by definition. □

Sequential assembly (2). The sequential assembly is as defined by HH98. We simply redefine it here to account for the changes introduced.

Definition 5.2.7 (Sequential assembly(2))**.**

$$\begin{aligned} P ⧉ Q &\;\widehat{=}\; (P \lhd \mathcal{L} \in \alpha\mathcal{L}P \rhd Q) \lhd \mathcal{L} \in (\alpha\mathcal{L}P \cup \alpha\mathcal{L}Q) \rhd SKIP \\ \alpha\mathcal{L}(P ⧉ Q) &\;\widehat{=}\; \alpha\mathcal{L}P \cup \alpha\mathcal{L}Q \end{aligned}$$ □

Example 5.2.8. *From Figure5.1(a).*

1. $P = ⧉_{1 \le i \le m_1} P_i$
2. $P_{31} = ⧉_{1 \le j \le m_2} P_{31j}$ □

Parallel Assembly. Traditionally, control enters sequentially into a single step at any one time. However, when dealing with parallelism, control may enter sequentially into many steps at any one time. It is therefore possible for a step, upon exit, to indicate that many steps may be executed in parallel next (cf. WH02 [141]).

The selection of the next parallel steps may be delegated to a dummy step, or *nesting step*, which is hence responsible of splitting control. This greatly simplifies not only reasoning, but definitions as well.

In particular, thanks to the nesting step, we are able to 'guarantee by construction' that *none* of the component steps may be jumped into at random, and that *all* the component steps are *always* entered at the same time —it is necessary to enter the nesting step first. There is also no need for an explicit synchronisation at the exit, which is possible only when every step has finished its execution; and only when such is the case does the empty step terminates its own execution.

In sum, the principal effect of the parallel assembly of steps is the creation of a dummy step that somewhat hides away control inherent to component steps. Hiding of control is *not total* since we can still observe its effect on the final value of $\mathcal{L}$.

We may now define the parallel composition of steps, called *parallel assembly* and denoted by //. It states that the parallel assembly of two steps yields a third, nesting step.

Definition 5.2.9 (Parallel assembly)**.** *Let* $\mathbf{nn} \notin \alpha\mathcal{L}P$*, and* $\mathbf{nn} \notin \alpha\mathcal{L}Q$*.*

$$\begin{aligned} P//_M Q &\mathrel{\widehat{=}} (P \parallel_M Q) \lhd \{\mathbf{nn}\} \in \mathcal{L} \rhd II \\ M(\mathcal{L}) &\mathrel{\widehat{=}} \mathcal{L}' = 1.\mathcal{L} \cup 2.\mathcal{L} \\ \alpha\mathcal{L}(P//_M Q) &\mathrel{\widehat{=}} \{\mathbf{nn}\} \cup \alpha\mathcal{L}P \cup \alpha\mathcal{L}Q \end{aligned}$$ □

Example 5.2.10. *From Figure5.1(a),(b), we have:* $P_3 = P_{31} // P_{32} // P_{33}$ □

Instructions, blocks, program blocks

In this section, we principally add more structure to the steps defined in the previous section.

First, we redefine the notion of single instruction.

Definition 5.2.11 (Single instruction(2))**.** *Let INST be a* $\mathbf{lv_0}$*-step, i.e.* $\alpha ls INST = \{\}$*.*

$$m : INST \mathrel{\widehat{=}} INST \lhd \mathcal{L} = \{m\} \rhd II$$

is a single *instruction.* □

We may distinguish two types of machine code blocks, according to the assembly operator used for their composition: (purely) sequential blocks (which we also call proper blocks) are the sequential assembly of single instructions (called machine code block in HH98 [66]); and parallel blocks (or nesting blocks) are the parallel assembly of single instructions.

Definition 5.2.12 (Proper-, nesting- block)**.** *A* proper block*, say SeqB, is a program expressible as a sequential assembly of single instructions i.e.*

$$\begin{aligned} SeqB &\mathrel{\widehat{=}} m_0 : INST_0 \boxplus m_1 : INST_1 \boxplus ... \boxplus m_n : INST_n \\ \alpha l(SeqB) &\mathrel{\widehat{=}} \{m_i \mid 0 \leq i \leq n\} \\ \alpha ls(SeqB) &\mathrel{\widehat{=}} \{\} \end{aligned}$$

A nesting block*, say ParB, is a program expressible as a parallel assembly of single instructions i.e.*

$$\begin{aligned} ParB &\mathrel{\widehat{=}} m_0 : INST_0 \,//\, m_1 : INST_1 \,//\, ... \,//\, m_n : INST_n \\ \alpha l(ParB) &\mathrel{\widehat{=}} \{\mathbf{nn}\} \\ \alpha ls(ParB) &\mathrel{\widehat{=}} \{m_i \mid 0 \leq i \leq n\} \end{aligned}$$ □

Example 5.2.13. *From Figure5.1(a).*

1. *Let* $20\!:\!P_1$ *and* $27\!:\!P_2$ *be single instructions, then* $(20\!:\!P_1 \boxplus 27\!:\!P_2)$ *is a proper block.*
2. *Let* $22 : P_{32}$ *and* $74 : P_{33}$ *be single instructions, then* $(22 : P_{32} \,//\, 74 : P_{33})$ *is a nesting block.* □

We expect any instruction to always pass control via a single exit point that may lead either to a proper instruction or to a nesting one. The definition of target code below reflects that expectation.

Definition 5.2.14 (Proper-, nesting- target code)**.** *Let P be a step. Let S below denote the set of entry points of all the steps that will be executed in parallel next, and let F denote the corresponding set of exit points.*
If $\alpha lsP \neq \{\}$, then we say that any step of the form $\langle (s, S), P, (F, f)\rangle$ is in nesting *target code, and defined by*

$$\begin{aligned} \langle (s, S), P, (F, f)\rangle &\mathrel{\widehat{=}} (\mathcal{L} \in \{s\} \cup S)^{\top} \fatsemi P \fatsemi (\mathcal{L} \in F \cup \{f\})_{\perp} \\ &= \mathbf{var}\mathcal{L} := \{s\} \cup S \fatsemi P \fatsemi (\mathcal{L} \in F \cup \{f\})_{\perp} \fatsemi \mathbf{end}\,\mathcal{L} \end{aligned}$$

However, if P is a $\mathbf{lv_0}$-step i.e. $\alpha lsP = \{\} = S = F$; we say that the step is in proper *target code and we may write simply $\langle s, P, f\rangle$.* □

Notice from above that the entry and exit points of the nesting step are independent of those of the steps supposed to execute in parallel. Upon entry, $\mathcal{L}$ is updated with the continuation s to ensure normal entry into the nesting step itself, and also with the set S so that the parallel steps may be entered conjointly afterwards. Upon exit, the value of $\mathcal{L}$ is first determined by a given merge function (cf. parallel assembly Def. 5.2.9) that ensures that $\mathcal{L}' \in F$ upon exiting the parallel assembly, and then $\mathcal{L}$ should be updated with the continuation f to provide normal exit out of the nesting step itself.

In what follows, we define the nesting target code for the parallel composition operator $\|$ only.

The parallel composition of two steps simply yields a third, nesting step, which has its own distinct entry and exit points from those of the steps that are to be run in parallel. Each component step may start only when its continuation has been provided by the nesting step.

Definition 5.2.15 (Target code for parallel composition)**.**

$$\begin{pmatrix} \langle (s_1, S_1), P, (F_1, f_1)\rangle \parallel \\ \langle (s_2, S_2), Q, (F_2, f_2)\rangle \end{pmatrix} \;\widehat{=}\; \exists (s,f) \bullet \langle (s, \{s_1, s_2\}), P // Q, (\{f_1, f_2\}, f)\rangle$$

$$\alpha l(P // Q) \;\widehat{=}\; \{s, f\}$$

$$\alpha ls(P // Q) \;\widehat{=}\; (\{s_1, f_1\} \cup S_1 \cup F_1) \cup (\{s_2, f_2\} \cup S_2 \cup F_2) \qquad \square$$

We expect the possibility of jumping into nested parallel steps. However, such jumps may not be left unguarded. The least requirement we can impose is that the continuation of the parent must figure in the definition of the jump statement together with the continuations of the children nodes to jump into.

Definition 5.2.16 (Vertical jump)**.** $jump(f, F) \;\widehat{=}\; \mathcal{L} := \{f\} \cup F \lhd \mathcal{L} = \mathbf{n} \rhd II$ $\square$

Example 5.2.17. *From Figure5.1(a). Let $l_i \in \alpha l P_i$ denote the last instruction executed in the block P_i. e.g. let $P_4 = (63 : P_{41} \boxplus 64 : P_{42})$, then $l_4 \in \{63, 64\}$.*

1. *normal jump to the block P_2*

$$jump\, l_2 \fatsemi P \;=\; P_2 \;=\; jump\, (l_2, \{\}) \fatsemi P$$

2. *normal jump to the nesting instruction P_3*

$$jump\, l_3 \;=\; P_3$$

P_3 will be responsible for initialising its nested parallel children.

3. *vertical jump to the block P_{311}*

$$jump(l_3, \{l_{311}\}) \fatsemi P \;=\; P_{311}$$

4. unsuccessful *vertical jump to the block P_{311}*

$$jump\, l_{311} \fatsemi P \;=\; II$$

5. *vertical jump to the nesting instruction P_{312}*

$$jump(l_3, \{l_{312}\}) \fatsemi P \;=\; P_{312}$$

Let $P_{312} = P_{3121} // P_{3122}$, then P_{312} is responsible for giving values to l_{3121} and l_{3122}.

6. *vertical jump to P_{312}'s children*

$$jump(l_3, \{l_{312}, l_{3121}, l_{3122}\}) \fatsemi P \;=\; P_{3121} // P_{3122}$$

The values of l_{3121} and l_{3122} are defined by the jump statement, not P_{312}. All the parents must be listed for the jump to succeed. l_3 allows a vertical jump through P_3, and l_{312} allows a vertical jump through P_{312}. □

Placing a label to multiple steps at the same time for the purpose of running them in parallel may seem like an interesting feature at first, but it would only add pointless complications. It is sufficient for us to place labels in each program individually and then run the result (of each labelling procedure) in parallel.

In sum, in this section, we have defined the semantics of programs that may contain the control variable $\mathcal{L}$, thus extending the range of programs expressible using HH98 and WH02 to nested parallel programs. We have not discussed the case of Higher-order (HO) programs and this should be done, given that the theory of mobile processes for which we have built the continuations above relies on HO programming. We postpone such a discussion to the following section.

5.2.3 Reactive Process Blocks

In this section we present the construction and semantics of Reactive Process Blocks (or RPB), based on the results obtained previously.

RPB processes are meant to extend UTP-CSP processes with continuations. Since we are also interested in Higher-order programming, i.e. the possibility of calling a program from within another program, we shall consider the extension of UTP-CSP with HO programming defined by Tang & Woodcock [126] (cf. Chap. 3, §3.1.2).

RPB Alphabet

First, let us consider UTP-CSP processes as defined in [126]. The alphabet of a UTP-CSP process P is defined by

$$\alpha P = VarP \cup Obs \cup \mathcal{A}$$

where $Obs = \{\mathbf{o}, \mathbf{o}' \mid \mathbf{o} \in \{ok, wait, tr, ref\}\}$ is the set of observational variables; $\mathcal{A}$ the set of events that P may perform (including communications), and $VarP$ the set of variables that P may use. We may extend such an alphabet with both $\alpha\mathcal{L}P$, the continuations of P, and $\mathcal{L}$, the control variable. This yields the following alphabet for P

$$\alpha P = VarP \cup \{\mathcal{L}\} \cup Obs \cup \mathcal{A} \cup \alpha\mathcal{L}P$$

Such an extension poses no difficulty at all, remembering that the alphabet of a predicate is simply a collection of symbols (otherwise meaningless on their own). We will refer to processes with such an alphabet as *reactive steps*.

RPB Healthiness conditions

UTP-CSP processes are characterised by a monotonic and idempotent healthiness condition $\boldsymbol{CSP} = \boldsymbol{R} \circ \boldsymbol{CSP1} \circ \boldsymbol{CSP2}$.

CSP trivially applies to processes whose alphabet is extended as defined above. Nonetheless, this is not enough for characterising reactive steps. In order to achieve such a characterisation, it is necessary to regard the definition of steps given earlier as an additional healthiness condition that applies to UTP-CSP processes with $\mathcal{L}$ in their alphabet. We denote that healthiness condition by ***RPB1***, defined below:

$$\boldsymbol{RPB1}(P) = P \lhd \mathcal{L} \in \alpha\mathcal{L}P \rhd II_R$$

The following law trivially holds:

$$\boldsymbol{RPB1} \circ \boldsymbol{CSP}(P) = \boldsymbol{CSP} \circ \boldsymbol{RPB1}(P)$$

The control variable $\mathcal{L}$ and both the observational variables *ok* and *wait* allow reasoning about termination; in addition, $\mathcal{L}$ permits reasoning about control, while both *ok* and *wait* permit reasoning about intermediate stable states. We need to ensure that no contradiction arises from the definitions of each of these variables. Thus, we define the following laws to ensure the consistency of the definitions of $\mathcal{L}$, *ok* and *wait* variables.

Definition 5.2.18 (Consistency between $\mathcal{L}$, *ok* and *wait*)**.** *The variables wait and $\mathcal{L}$ must agree on the behaviour of a Step prior to its execution.*

$$\begin{aligned} \boldsymbol{A1} \quad & P \wedge wait \Leftrightarrow P \wedge \mathcal{L} \notin \alpha\mathcal{L}P \\ \text{(or equivalently)} \quad & P = P \wedge (wait \Leftrightarrow \mathcal{L} \notin \alpha\mathcal{L}P) \end{aligned}$$

The variables ok and $\mathcal{L}$ must agree on the start of the execution.

$$\begin{aligned} \boldsymbol{A2} \quad & P \wedge ok \Leftrightarrow P \wedge (\mathcal{L} \in \alpha\mathcal{L}P) \\ \text{(or equiv.)} \quad & P = P \wedge (ok \Leftrightarrow \mathcal{L} \in \alpha\mathcal{L}P) \end{aligned}$$

The variables ok and wait, and $\mathcal{L}$ must agree on valid intermediate states.

$$\begin{aligned} \boldsymbol{A3} \quad & P \wedge ok' \wedge wait' \Leftrightarrow P \wedge (\mathcal{L}' \in \alpha\mathcal{L}P) \\ \text{(or equiv.)} \quad & P = P \wedge (ok' \wedge wait' \Leftrightarrow \mathcal{L}' \in \alpha\mathcal{L}P) \end{aligned}$$

The variables ok and wait, and $\mathcal{L}$ must agree on the termination.

$$\begin{aligned} \boldsymbol{A4} \quad & P \wedge ok' \wedge \neg\, wait' \Leftrightarrow P \wedge \mathcal{L}' \notin \alpha\mathcal{L}P \\ \text{(or equiv.)} \quad & P = P \wedge (ok' \wedge \neg\, wait' \Leftrightarrow \mathcal{L}' \notin \alpha\mathcal{L}P) \end{aligned}$$

□

Definition 5.2.19. $\boldsymbol{A} \mathrel{\widehat{=}} \boldsymbol{A1} \circ \boldsymbol{A2} \circ \boldsymbol{A3} \circ \boldsymbol{A4}$ □

Since ***A1***, ***A2***, ***A3***, and ***A4*** are all conjunctive, the order of their composition is irrelevant.

It turns out that reactive steps that are $\boldsymbol{RPB1} \circ \boldsymbol{CSP}$ healthy are also ***A1*** and ***A2*** healthy. The proof of the latter is obtained easily if we remark that ***A1*** and $\boldsymbol{R3} \circ \boldsymbol{RPB1}$ are similar, and also that ***A2*** and $\boldsymbol{CSP1} \circ \boldsymbol{RPB1}$ are similar.

Theorem 5.2.20. $\boldsymbol{A1} \circ \boldsymbol{RPB1} \circ \boldsymbol{CSP} = \boldsymbol{RPB1} \circ \boldsymbol{CSP}$

Proof.

$\boldsymbol{A1}(P)$
$=\{\boldsymbol{A1}\,\text{def}\}$
$P \wedge (wait \Leftrightarrow \mathcal{L} \notin \alpha\mathcal{L}P)$
$=\{\text{prop calc}\}$
$P \wedge (wait \Rightarrow \mathcal{L} \notin \alpha\mathcal{L}P) \wedge (\mathcal{L} \notin \alpha\mathcal{L}P \Rightarrow wait)$
$=\{\text{prop calc}\}$
$P \wedge (\neg\, wait \vee \mathcal{L} \notin \alpha\mathcal{L}P) \wedge (\mathcal{L} \in \alpha\mathcal{L}P \vee wait)$
$=\{\text{prop calc}\}$
$P \wedge ((\neg\, wait \wedge \mathcal{L} \in \alpha\mathcal{L}P) \vee (wait \wedge \mathcal{L} \notin \alpha\mathcal{L}P))$
$=\{\text{prop calc}\}$
$(P \wedge \neg\, wait \wedge \mathcal{L} \in \alpha\mathcal{L}P) \vee (P \wedge wait \wedge \mathcal{L} \notin \alpha\mathcal{L}P)$

$\boldsymbol{R3} \circ \boldsymbol{RPB1}(P)$
$=\{\boldsymbol{R3}\,\text{def}\}$
$II_R \lhd wait \rhd \boldsymbol{RPB1}(P)$
$=\{\boldsymbol{RPB1}\,\text{def}\}$
$II_R \lhd wait \rhd (P \lhd \mathcal{L} \in \alpha\mathcal{L}P \rhd II_R)$
$=\{\text{cond symm}\}$
$II_R \lhd wait \rhd (II_R \lhd \mathcal{L} \notin \alpha\mathcal{L}P \rhd P)$
$=\{\text{cond assoc, cond idemp}\}$
$II_R \lhd wait \vee \mathcal{L} \notin \alpha\mathcal{L}P \rhd P$
$=\{\text{cond def, De Morgan's Law}\}$
$(P \wedge \neg\, wait \wedge \mathcal{L} \in \alpha\mathcal{L}P) \vee (II_R \wedge (wait \vee \mathcal{L} \notin \alpha\mathcal{L}P))$

$$
\begin{aligned}
&\boldsymbol{A1} \circ \boldsymbol{R3} \circ \boldsymbol{RPB1}(P) \\
=&\{\boldsymbol{A1}\,\text{def expanded form}\} \\
&(\boldsymbol{R3} \circ \boldsymbol{RPB1}(P) \wedge (\neg\ wait \wedge \mathcal{L} \in \alpha\mathcal{L}P)) \vee (\boldsymbol{R3} \circ \boldsymbol{RPB1}(P) \wedge (wait \wedge \mathcal{L} \notin \alpha\mathcal{L}P)) \\
=&\{\boldsymbol{R3} \circ \boldsymbol{RPB1}(P)\,\text{def expanded form, prop calc}\} \\
&(P \wedge \neg\ wait \wedge \mathcal{L} \in \alpha\mathcal{L}P) \vee (II_R \wedge wait \wedge \mathcal{L} \notin \alpha\mathcal{L}P) \\
=&\{\boldsymbol{R3} \circ \boldsymbol{RPB1}(P)\,\text{def expanded form}\} \\
&\boldsymbol{R3} \circ \boldsymbol{RPB1}(P)
\end{aligned}
$$

□

Theorem 5.2.21. $\boldsymbol{A2} \circ \boldsymbol{RPB1} \circ \boldsymbol{CSP} = \boldsymbol{RPB1} \circ \boldsymbol{CSP}$

Proof.

$$
\begin{aligned}
&\boldsymbol{A2}(P) \\
=&\{\boldsymbol{A1}\,\text{def}\} \\
&P \wedge (ok \Leftrightarrow \mathcal{L} \in \alpha\mathcal{L}P) \\
=&\{\text{prop calc}\} \\
&P \wedge (ok \Rightarrow \mathcal{L} \in \alpha\mathcal{L}P) \wedge (\mathcal{L} \in \alpha\mathcal{L}P \Rightarrow ok) \\
=&\{\text{prop calc}\} \\
&P \wedge (\neg\ ok \vee \mathcal{L} \in \alpha\mathcal{L}P) \wedge (\mathcal{L} \notin \alpha\mathcal{L}P \vee ok) \\
=&\{\text{prop calc}\} \\
&(P \wedge \neg\ ok \wedge \mathcal{L} \notin \alpha\mathcal{L}P) \vee (P \wedge ok \wedge \mathcal{L} \in \alpha\mathcal{L}P)
\end{aligned}
$$

$$
\begin{aligned}
&\boldsymbol{CSP1} \circ \boldsymbol{RPB1}(P) \\
=&\{\boldsymbol{CSP1}\,\text{def}\} \\
&\boldsymbol{RPB1}(P) \lhd ok \rhd tr \leq tr' \\
=&\{\boldsymbol{RPB1}\,\text{def}\} \\
&(P \lhd \mathcal{L} \in \alpha\mathcal{L}P \rhd II_R) \lhd ok \rhd tr \leq tr' \\
=&\{\text{cond assoc, cond idemp}\} \\
&P \lhd \mathcal{L} \in \alpha\mathcal{L}P \vee ok \rhd tr \leq tr' \\
=&\{\text{cond def, De Morgan's Law}\} \\
&(P \wedge (ok \vee \mathcal{L} \in \alpha\mathcal{L}P)) \vee (tr \leq tr' \wedge \neg\ ok \wedge \mathcal{L} \notin \alpha\mathcal{L}P)
\end{aligned}
$$

$$
\begin{array}{ll}
& \boldsymbol{A2} \circ \boldsymbol{CSP1} \circ \boldsymbol{RPB1}(P) \\
= & \{\boldsymbol{A2}\,\text{def expanded form}\} \\
& (\boldsymbol{CSP1} \circ \boldsymbol{RPB1}(P) \wedge (\neg\, ok \wedge \mathcal{L} \notin \alpha\mathcal{L}P)) \vee (\boldsymbol{CSP1} \circ \boldsymbol{RPB1}(P) \wedge (ok \wedge \mathcal{L} \in \alpha\mathcal{L}P)) \\
= & \{\boldsymbol{CSP1} \circ \boldsymbol{RPB1}(P)\,\text{def expanded form, prop calc}\} \\
& (tr \leq tr' \wedge \neg\, ok \wedge \mathcal{L} \notin \alpha\mathcal{L}P) \vee (P \wedge ok \wedge \mathcal{L} \in \alpha\mathcal{L}P) \\
= & \{\boldsymbol{CSP1} \circ \boldsymbol{RPB1}(P)\,\text{def expanded form}\} \\
& \boldsymbol{CSP1} \circ \boldsymbol{RPB1}(P)
\end{array}
$$

□

Since ***A3*** and ***A4*** both mention the final of value of $\mathcal{L}$, we may make a parallel with the expression $P \fatsemi (\mathcal{L} \notin \alpha\mathcal{L}P)_\perp$, which allows us to specify the final value of $\mathcal{L}$. That is, we expect our predicates to verify the equation $P = P \fatsemi (\mathcal{L} \notin \alpha\mathcal{L}P)_\perp$. We thus define a new healthiness condition:

$$\boldsymbol{RPB2} \quad P \;=\; P \fatsemi (\mathcal{L} \notin \alpha\mathcal{L}P)_\perp$$

It turns out that ***RPB1*** $\circ$ ***CSP*** healthy reactive steps that are ***RPB2*** healthy are also ***A3*** and ***A4*** healthy.

Theorem 5.2.22. $\boldsymbol{A3} \circ \boldsymbol{RPB2} \circ \boldsymbol{RPB1} \circ \boldsymbol{CSP}(P) = \boldsymbol{RPB2} \circ \boldsymbol{RPB1} \circ \boldsymbol{CSP}(P)$

Proof. If $P = P \wedge \mathcal{L}' \in \alpha\mathcal{L}P$, then $ok' = true$; otherwise, if $ok' = false$ then the value of $\mathcal{L}'$ is unobservable. If $wait' = true$ then

$$
\begin{array}{ll}
& \boldsymbol{RPB2}(P) \\
= & \{\boldsymbol{RPB2}\,\text{def, seq comp, assertion def}, \perp = CHAOS, ok' = true, wait' = true\} \\
& (P(ok', wait') \wedge II_R(ok, wait) \wedge \mathcal{L}' \notin \alpha\mathcal{L}P) \vee (P(ok', wait') \wedge CHAOS(ok, wait) \wedge \mathcal{L}' \in \alpha\mathcal{L}P) \\
= & \{\text{hypothesis}\, P = P \wedge \mathcal{L}'\} \\
& P(ok', wait') \wedge CHAOS(ok, wait) \wedge \mathcal{L}' \in \alpha\mathcal{L}P \\
= & \{CHAOS\,\text{def}\} \\
& P(ok', wait') \wedge II_R(ok, wait) \wedge \mathcal{L}' \in \alpha\mathcal{L}P \\
= & \{II_R\,\text{def}\} \\
& P(ok', wait') \wedge ok' \wedge wait' \wedge \mathcal{L}' \in \alpha\mathcal{L}P \\
= & \{\text{hypothesis}\} \\
& P(ok', wait') \wedge ok' \wedge wait'
\end{array}
$$

i.e.

$$\textbf{\textit{RPB2}} \circ \textbf{\textit{RPB1}} \circ \textbf{\textit{CSP}}(P) \wedge \mathcal{L}' \in \alpha\mathcal{L}P \;\Leftrightarrow\; \textbf{\textit{RPB2}} \circ \textbf{\textit{RPB1}} \circ \textbf{\textit{CSP}}(P) \wedge \mathit{wait}' \wedge \mathit{ok}'$$

□

Theorem 5.2.23. $\textbf{\textit{A4}} \circ \textbf{\textit{RPB2}} \circ \textbf{\textit{RPB1}} \circ \textbf{\textit{CSP}}(P) = \textbf{\textit{RPB2}} \circ \textbf{\textit{RPB1}} \circ \textbf{\textit{CSP}}(P)$

Proof. Similar to the previous one: if $P = P \wedge \mathcal{L}' \notin \alpha\mathcal{L}P$ ($\notin$, not $\in$), then $\mathit{ok}' = \mathit{true}$; otherwise, if $\mathit{ok}' = \mathit{false}$ then the value of $\mathcal{L}'$ is unobservable. This time however, we choose $\mathit{wait}' = \mathit{false}$. □

We summarise the previous results in the following definition and theorem.

Definition 5.2.24. *(**RPB**)* $\textbf{\textit{RPB}} \mathrel{\widehat{=}} \textbf{\textit{RPB2}} \circ \textbf{\textit{RPB1}} \circ \textbf{\textit{CSP}}$ □

Theorem 5.2.25. *Let P denote for any Step whose alphabet includes that of Reactive Processes. Then:*

$$\textbf{\textit{A}} \circ \textbf{\textit{RPB}}(P) = \textbf{\textit{RPB}}(P)$$

Proof. From Theorems 5.2.20 to 5.2.23. □

We may now define reactive steps formally:

Definition 5.2.26 (Reactive step)**.** *Any predicate whose alphabet includes that for reactive processes, and, additionally, both $\alpha\mathcal{L}$, and $\mathcal{L}$, and that is **RPB** healthy is called a* reactive step. □

Discussion. Recall Def. 2.6.13 (Chap. 2, §2.6) of blocks and proper blocks:

$$(P : S \Rightarrow F) \;\mathrel{\widehat{=}}\; P = (P \fatsemi (l \in F \cup \{\mathbf{n}\})_{\perp}) \lhd l \in S \cup \{\mathbf{n}\} \rhd \mathit{II})$$

$P : S \Rightarrow F$ defines a set of programs and not a single program as one might have expected. The healthiness condition $\textbf{\textit{RPB2}} \circ \textbf{\textit{RPB1}}$ defines exactly the same set of programs when l is used instead of $\mathcal{L}$.

The difference in formulation with HH98 has some importance. In effect, we may now reason in terms of a UTP theory instead of reasoning in terms of a particular notation. An immediate consequence is that we can start thinking about links between theories. For illustration, we may readily characterise sequential programs as steps that have no children i.e.

$$\textbf{\textit{RPBSeq}} \quad P = P \wedge \alpha\mathit{lsP} = \{\}$$

$\textbf{\textit{RPBSeq}} \circ \textbf{\textit{RPB}}$ defines a subset theory of that defined by **RPB**. □

Basic RPB predicates and operators

We now give the semantics of some basic predicates and operators. Since we are building a target language for high-level UTP-CSP processes (that do not contain $\mathcal{L}$), we need to specify our basic instructions. The definition of a target code given in the previous section makes it possible to define arbitrarily complex predicates even as single instructions. In what follows, we will consider a language with only two single instructions: assignment and action prefix.

The notation $m : INST$ may be considered as a predicate transformer, a function that takes a constant value m and a UTP-CSP process $INST$, and returns a reactive step with continuations $\{m\}$.

Example 5.2.27.

1. Assignment instruction

$$\begin{aligned} m : (x := e) &\mathrel{\widehat{=}} (x := e)_{+\mathcal{L}} \lhd \mathcal{L} = m \rhd II_R \\ \alpha\mathcal{L}(m : (x := e)) &\mathrel{\widehat{=}} \{m\} \end{aligned}$$

2. Simple action prefix instruction

$$\begin{aligned} m : (a \to SKIP) &\mathrel{\widehat{=}} (a \to SKIP)_{+\mathcal{L}} \lhd \mathcal{L} = m \rhd II_R \\ \alpha\mathcal{L}(m : (a \to SKIP)) &\mathrel{\widehat{=}} \{m\} \end{aligned}$$ □

For any $INST$ a UTP-CSP process with the following BNF

$$INST ::= x := e \mid a \to SKIP$$

it is now clearer how using the definitions for steps given in previous sections, one can build the existing RPB operators.

We may interpret the notation $\langle s, BINST, f\rangle$ as denoting a predicate transformer, a function that takes two constant values s and f, and a basic instruction $BINST$ with the following BNF

$$BINST ::= m : INST$$

and returns a reactive step (target code), with continuations $\{s, f\}$.

$$\begin{aligned} \langle s, BINST, f\rangle &\mathrel{\widehat{=}} \mathbf{var}\, \mathcal{L} := \{s\} \fatsemi s : (INST_{+\mathcal{L}} \fatsemi \mathcal{L} := \{f\}) \\ \alpha\mathcal{L}\, \langle s, BINST, f\rangle &\mathrel{\widehat{=}} \{s, f\} \end{aligned}$$

We may define in an analogue way first basic (sequential) blocks, basic parallel blocks, and then proper blocks (or reactive process blocks).

The following example is taken from [136]. The programming language occam-pi is used as the semantic domain for both serial and parallel integrators in [136], whereas Reactive Processes (and continuations) are used below.

Example 5.2.28 (An integrator)**.** *The basic interface of the integrator process is two channels, one input and one output. Given the input sequence x, y, z, the integrator will output running sums: $x, (x + y), (x + y + z)$ and so on.*

1. *The Serial integrator*

$$SIntegrate \;\widehat{=}\; total := 0 \fatsemi \mu X \bullet (in?x \to total := total + x \fatsemi out!total \to X)$$

could be translated into

$$(SIntegrate : S \Rightarrow F) = \langle m_1, Init, m_2 \rangle \fatsemi \mu X \bullet \begin{pmatrix} \langle m_2, Input, m_3 \rangle \fatsemi \langle m_3, Add, m_4 \rangle \fatsemi \\ \langle m_4, Output, m_2 \rangle \fatsemi X \end{pmatrix}$$

where

$$\begin{aligned} Init &= m_1 : total := 0 \\ Input &= m_2 : in?x \to SKIP \\ Add &= m_3 : total := total + x \\ Output &= m_4 : out!total \to SKIP \end{aligned}$$

2. *The Parallel integrator*

$$\begin{aligned} PIntegrate &\;\widehat{=}\; (Plus \parallel Delta \parallel Prefix) \setminus \{a, b, c\} \\ Plus &\;\widehat{=}\; (in?x \to SKIP \;||\; c?y \to SKIP) \fatsemi a!(x + y) \to Plus \\ Delta &\;\widehat{=}\; a?x \to (out!x \to SKIP \parallel b!x \to SKIP) \fatsemi Delta \\ Prefix &\;\widehat{=}\; c!0 \to \mu X \bullet (b?x \to c!x \to X) \end{aligned}$$

could be translated into

$$(PIntegrate : S \Rightarrow F) = (Plus : S \Rightarrow F \;//\; Delta : S \Rightarrow F \;//\; Prefix : S \Rightarrow F) \setminus \{a, b, c\}$$

$$(Plus : S \Rightarrow F) = \begin{pmatrix} \begin{pmatrix} \langle m_1, In(m_1, x)[in \leftarrow ch], m_3 \rangle \;// \\ \langle m_2, In(m_2, y)[c \leftarrow ch], m_3 \rangle \end{pmatrix} \fatsemi \\ \langle m_3, Out(m_3, x + y)[a \leftarrow ch], \mathbf{nn} \rangle \fatsemi Plus \end{pmatrix}$$

$$(Delta : S \Rightarrow F) = \begin{pmatrix} \langle m_{10}, In(m_{10}, x)[a \leftarrow ch], \mathbf{nn} \rangle \fatsemi \\ \begin{pmatrix} \langle m_{11}, Out(m_{11}, x)[out \leftarrow ch], m_{10} \rangle \;// \\ \langle m_{12}, Out(m_{12}, y)[b \leftarrow ch], m_{10} \rangle \end{pmatrix} \fatsemi \\ Delta \end{pmatrix}$$

$$(Prefix : S \Rightarrow F) = \left(\begin{array}{l} \langle m_{20}, Out(m_{20}, 0)[c \leftarrow ch], m_{21} \rangle) ; \\ \mu\, X \bullet \left(\begin{array}{l} \langle m_{21}, In(m_{21}, x)[b \leftarrow ch], m_{22} \rangle ; \\ \langle m_{22}, Out(m_{22}, x)[c \leftarrow ch], m_{20} \rangle) \, ; \, X \end{array} \right) \end{array} \right)$$

where

$$\begin{aligned} In(s, z) &= s : ch?z \rightarrow SKIP \\ Out(s, z) &= s : ch!z \rightarrow SKIP \end{aligned}$$

□

Assuming that every other operator is well-defined, we now turn to the case of higher-order (HO) programming.

HO variable declaration. A HO program or *procedure* is one that may be assigned as the value of a HO process variable. $\{|\ P\ |\}$ denotes the procedure that, when executed, behaves like process P. In UTP-CSP, the declaration of a HO variable h supposes that h may contain as values only procedures that have the same actions set $\mathcal{A}$. We follow this idea for continuations too. We assume that any HO variable h may only receive for value procedures that have the same continuations. This means that we have no need for modifying the existing definition [126], besides adding the latter postulate about continuations.

Summary. In this section we have defined continuations for reactive processes, with an emphasis on the semantics for the parallel composition operator. New healthiness conditions have been defined and the result is a theory of reactive process blocks which permits characterising continuations for reactive processes. In the next section we discuss the representation of the state for reactive processes.

5.3 Representation of the state for Reactive Processes

In this section, we discuss how the state of a UTP-CSP process may be represented and recorded into its trace. We shall not use that representation in our subsequent theory of strong mobility, however. The reasons for not using it will be discussed in greater detail in the next section. We may already state that the following representation may eventually not yield reactive processes simply because the dependency between consecutive states would of necessity be lifted into the trace. Nonetheless, the following presentation is useful for the discussions in the next section.

The state of a process is given by mapping the variables in its alphabet with their observed values. The alphabet of a process may be partitioned into two subsets: observational variables, e.g. *wait*, *tr*, and program variables. In this section, when we talk of the state of a process, we refer to program variables only. It is always possible to determine the state

of a process from an inspection of the values of its variables: below, we define the function $state : Var \to Val$ which realises and records such mappings.

The state of a process may change during the process's activation. The operation that changes the state of a process may be conceived of as an action whose effect is to update that state: we call it the *update action*. More precisely, we may consider that every process embeds such an action, which has the same effect as assignment (:=). This remark permits us to make an economy of notation by associating every assignment with a given update action. Let us denote by $\mathcal{U}$ the set of all update actions of a process. Then:

$$\begin{aligned} \mathcal{U}P &\mathrel{\widehat{=}} \{v := e \mid v \in \alpha P \bullet e \textit{ is a value}\} \\ &= \{v \mapsto e \mid v \in \alpha P \bullet e \textit{ is a value}\} \end{aligned}$$

The association assignment-update action may be formalised by simply modifying the definition of assignment: now assignment also modifies the trace by introducing therein the corresponding update action.

$$x := e \mathrel{\widehat{=}} x' = e \land tr' = tr \frown \langle x \mapsto e \rangle \land v' = v$$

Each update action denotes, for a process P, the set of mappings $var \mapsto val$ where $var \in \alpha P$ is a single variable. Equality of update actions is simply equality between their respective set of mappings, and inequality is defined by set inclusion.

The trace of a process must be slightly modified to account for update actions i.e. the type of tr must now be $(\mathcal{U} \cup \mathcal{A})^*$.

The state function may then be defined by:

$$\begin{aligned} state : \alpha P &\to Val \\ tr' &\mapsto last\, tr' \upharpoonright \mathcal{U} \end{aligned}$$

Summary. In this section we have discussed a way for introducing into the alphabet the function *state*(), which gives the mappings between variables and their values. In the next section we define the generic interrupt, the last element before giving the semantics of process mobility. We will also justify why using the state function *state*() is actually undesirable.

5.4 Generic interrupt

5.4.1 Preliminaries

The interrupt operator is one of the most complex operators that may be defined. In fact, there is not much literature on the interrupt operator in process algebra. In CSP in particular, existing semantics for the generic interrupt are *all* given in a *timed* model. Overall, there are two pieces of work on the generic interrupt in UTP.

Huang et al. [68, 69, 70] have proposed a semantics for interrupt programs in a discrete time model. An interrupt program is a pair (M, IH) composed of a *main* program M, which is *sequential*, together with a set IH of so called *interrupt handlers*, also sequential (interruptible) programs i.e. each program $P \in IH$ also has the form (P, IH_p). Each interrupt handler has its own *unique* interrupt event $i \notin \mathcal{A}M$ so that their proper denotation is actually $i_p \to P$. The main program M holds each interrupt event in a *queue*, denoted by a special variable q. M may enable and disable interrupts triggered by the environment, and may itself trigger interrupt requests. Programs are characterised by ***R1*** healthy predicates (called ***H1*** in [70]), and obey an additional healthiness condition relative to interrupt. Trace elements are triples (t, σ, μ), where t is the time (of occurrence of an event, or either an instruction), σ (a record of) the state of the program at the end of the execution, and μ is a program identifier: it indicates whether the state is that of the main program ($\mu = 0$) or of an interrupt handler ($\mu = 1$).

The state of a program is simply given by σ viz. $P(\sigma) = \sigma'$, and unlike traditional UTP-CSP semantics, it must be inferred from the trace. The interrupt mechanism works as follows: when interrupts are disabled, $M \triangle P$ behaves like M. When enabled, the interrupt handler takes over and then returns control to the main program. $M \triangle P$ works like $M \parallel P$ where the final state is decided by the merge function. Basically, the trace of M is partitioned into the trace before (or before-trace) and the trace after (after-trace) the interrupt (for each interrupt handler); then, the trace of an interrupt handler P is inserted between the before-trace and the after-trace of M, meanwhile time is moved forward in the after-trace of M such as to coincide with the last time in the inserted trace of P.

The language of Huang &al. is quite restrictive and would demand substantial work to be extended to the whole of reactive processes.

In [134, 135], Kun Wei defines a number of interrupt operators in a timed model of UTP-CSP, using reactive designs [35], [134] (cf. Def. 2.5.27). Based on the definition in [84] a reactive design for the catastrophic interrupt is defined; semantics for the generic interrupt $P \triangle Q$ are also provided, using a similar approach than Huang et al. but without the language limitation. An important difference with [70] is that the state of the interruptible process P is not recorded in the trace; hence, when an interrupt occurs, the state of P is discarded and only the state of the interrupting process Q is available at the end. The final trace, however, is a concatenation of the trace of P before interrupt, followed by a trace of Q.

Notwithstanding the fact that Kun Wei's semantics for the generic interrupt discard the state of P upon interrupt, the existence of a link between UTP Timed Reactive Designs and UTP Reactive Processes provides us with a method for defining the generic interrupt for reactive processes. In fact, going from timed to untimed reactive designs is enough since the latter are but untimed reactive processes. However, mixing both styles of semantics (i.e. reactive processes and reactive designs) does *not* have much interest, so we would prefer to prolong the transformation from untimed reactive designs to untimed reactive processes.

Unfortunately, this series of transformation seems not only tedious, but the result may be

difficult to exploit in proofs and also, mainly, for reasoning. Indeed, the transformations are not structure-preserving (viz. they are not homomorphisms). For illustration, compare the semantics of the catastrophic interrupt [84] with its equivalent timed reactive design [135]. Another issue is that explicitly saving the state into the trace may not be possible whilst preserving the semantics of reactive processes.

In effect, ***R2*** may be violated since the state component may not be arbitrary. This constraint appears more clearly from the semantics of Huang et al. than from the use of a specific update action for encapsulating the state (cf. §5.3). Indeed, the former shows more clearly the dependency between the last state and the current one; hence, unless we were to express the dependency relation between (update actions) explicitly, the traces semantics would be too permissive. Furthermore, there is no guarantee that we can successfully characterise the dependency in every case. For example, in [119] Shi et al. propose as a possible restriction that the last state in tr equals the first state in tr'. However such a restriction cannot suffice. Suppose that we expect for a given state σ that $\sigma' = f(\sigma)$. Then, we may change the value of σ such that the equation $\sigma' = f(\sigma)$ is no longer valid. The specification of f illustrates the specification of a dependency condition between states. Without introducing non-determinism, it is possible that f holds between certain states only, and not the whole state space. And we may conceive of more constricting dependency conditions.

CSP1 would also be invalid since except for trace expansion $tr \leq tr'$, it should not be possible to make any other observation about a process, when $ok = \mathit{false}$. Putting the state in tr means that we can observe that state, which contradicts ***CSP1***. The issue may be restated in terms of making the state visible in the trace. Then, hiding all the update actions might appear to be the solution, however, that would render the recording obsolete.

Finally, assuming that the above problems have been mitigated if not totally solved (i.e. we have successfully recorded the state of P into its trace), at this stage, it is not totally clear how we would make the state of P available to Q —given that they are running in parallel. The latter question is studied by neither K.Wei [135] nor Huang et al. [70], and their respective semantics do *not* actually answer the question.

The previous discussions suggested an approach to solving the problem of defining the generic interrupt for (Untimed) Reactive Processes, starting from an existing, related framework. Two candidate frameworks have been presented, and the infeasibility of that approach has been discussed. A second approach may be to start directly from reactive processes. In this case, we have two possibilities: (i) use time; (ii) not use time. In either case, the construction of the semantics must be guided by the constraint of making available the state of the interruptible process to the interrupting one.

The timed option may have the advantage of making analysis easier, since every existing definition (of the generic interrupt) in the literature uses time. The major inconvenience is that since we ultimately want a semantics for untimed reactive processes, we will have to abstract away from time. However, this approach lacks of efficiency. Suppose that we want

to define a system with process mobility in untimed CSP where only a single process may be moved. Then, we would have to transform that process into a timed version just so that it can be interrupted. A further inconvenience is that we need to account for continuations. Hence, we may end up with quite a complicated theory —with three levels of abstraction, time, untimed CSP, and continuations .

On the other hand, the links between theories indicate that an untimed definition is feasible, although it may not be as elegant as one based on a timed model. We thus formulate the following hypothesis:

> [HypothesisGenericNoTransform] It is possible to define the generic interrupt in untimed CSP without having to transform a (possibly) timed equivalent first.

We will present our results starting with a description of the problem (of formalising the generic interrupt operator) in UTP-CSP terms. From there we build an initial specification of the generic interrupt. We then refine that initial specification progressively until we eventually reach a satisfying definition. The refinement steps are rather informal although, in accord with the recommended methodology for building specifications (e.g. [66, Chap. 1, §1.4]), we freely mix both formal and informal notations and arguments as may seem suitable. Our proposition is quite novel and relies on the expansion law for parallel composition [66, Chap. 7, §7.5]. We may emit a second hypothesis in relation with our objective:

> [HypothesisNoBarrierNoShare] Using bulk synchronous parallelism as defined in UTP (expansion law for parallel composition —[66, Chap. 7, §7.5, **L9**]), it is possible to pass the state of an interruptible process P to its interrupting process Q if a barrier synchronisation is placed on the execution line of both P and Q, just after the occurrence of the interrupt event and before the occurrence of Q.

In the remainder of this section, unless otherwise stated, $1.tr$ will stand for the trace of P, $2.tr$ for the trace of Q, and tr for that of $P \parallel Q$. Recall:

$$
\begin{aligned}
\mathcal{A}(P \parallel Q) &\mathrel{\widehat{=}} \mathcal{A}P \cup \mathcal{A}Q \\
P \parallel Q &\mathrel{\widehat{=}} P(\mathbf{o}, 1.\mathbf{o}') \wedge Q(\mathbf{o}, 2.\mathbf{o}') \fatsemi M(1.\mathbf{o}, 2.\mathbf{o}, \mathbf{o}') \\
M &\mathrel{\widehat{=}} \left(\begin{array}{l} ok' = (1.ok \wedge 2.ok) \wedge \\ wait' = (1.wait \vee 2.wait) \wedge \\ ref' = (1.ref \cup 2.ref) \wedge \\ (tr' - tr) = ((1.tr - tr) \parallel (2.tr - tr)) \end{array} \right) \fatsemi SKIP
\end{aligned}
$$

5.4.2 Formalisation

The basic idea behind the definition of the generic interrupt is to consider it as a restricted form of parallel composition. In effect, $P \triangle Q$ behaves like $P \parallel Q$ with the following restrictions on the legal behaviours:

- If P has not been interrupted i.e. Q has not occurred, then $i \notin tr'$ and the behaviour of $P \triangle Q$ is entirely determined by P.
- If P has been interrupted i.e. Q has occurred, then $i \in tr'$ and there are two possibilities:
  1. interrupt occurred before P could activate its first instruction, then $i = head\ tr'$ and the behaviour of $P \triangle Q$ is entirely determined by Q;
  2. interrupt occurred during P's activation and before P's termination, then $i \in tail\ tr' \wedge i \neq last\ tr'$ and the behaviour of $P \triangle Q$ is determined by both the behaviour of P before interrupt, denoted by $P\,\mathbf{bef}\,i$, and the behaviour of Q.

That is:

$$[GenInterruptSpec1] \qquad P \triangle Q = \begin{cases} P & if\ i \notin tr' \\ Q & if\ i \in tr' \wedge i = head\ tr' \\ P\,\mathbf{bef}\,i \fatsemi Q & if\ i \in tr' \wedge i \in tail\ tr' \wedge i \neq last\ tr' \end{cases}$$

At this stage, the relation between $\triangle$ and $\|$ may not be quite obvious. We make this more precise hereafter. Remark that

$$[\triangle - \| - equiv] \qquad P \parallel Q = \begin{cases} P & if\ Q = SKIP\ or\ Q = STOP \\ Q & if\ P = SKIP\ or\ P = STOP \\ (P \parallel\parallel Q) & otherwise \end{cases}$$

Note: Interleaving $\parallel\parallel$ is used mainly for indication, and the possibilities of communication and synchronisation between P and Q (up to the occurrence of the interrupt event) *must be retained*.

Then, trace-wise, $Q = SKIP\ or\ Q = STOP \Rightarrow 2.tr' = \langle\rangle$ which means that $i \notin tr'$. Similarly, in the second case we have $P = SKIP\ or\ P = STOP \Rightarrow 1.tr' = \langle\rangle$, which means that $i = head\ tr'$.
In sum:

- if $i \notin tr'$ then $P \triangle Q = P = P \parallel Q$
- if $i = head\ tr'$ then $P \triangle Q = Q = P \parallel Q$
- the third case (i.e. $i \in tr' \wedge i \in tail\ tr' \wedge i \neq last\ tr'$) requires a little more analysis. We shall use the expansion law for || (cf. below), and suppose that $P = P_1 \fatsemi \mathbf{sync} \fatsemi ... \fatsemi \mathbf{sync} \fatsemi P_n$. For now, we consider that each P_i is atomic or not-interruptible.

Before discussing the third case in greater detail, we first give the general idea of what we are trying to achieve: If we can break a process into individual steps that may not be interrupted during their execution (but only before their execution starts), then interrupt may be determined by recursion over process terms as follows: [*GenInterruptSpec*2]

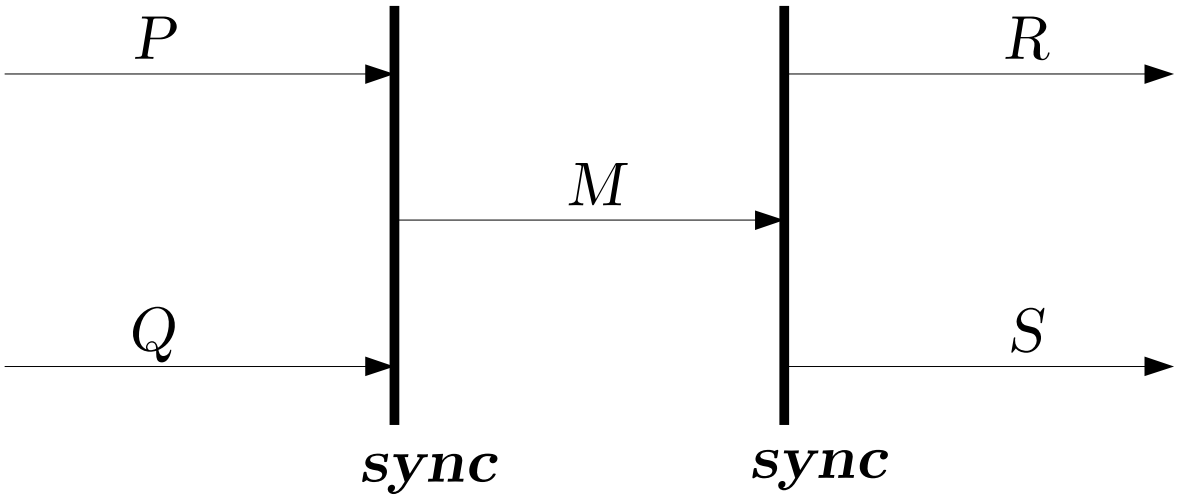

Figure 5.2: Illustration of Bulk Synchronous Paralellism (BSP)

- If P is atomic i.e. not-interruptible, then

$$P \triangle Q = \begin{cases} P & if \quad i \notin tr' \\ Q & if \quad i = head\ tr' \end{cases}$$

- Else If $P = P_1 \fatsemi P_2$, where both P_1 and P_2 are atomic, then

$$P \triangle Q = (P_1 \triangle Q) \,\Box\, (P_1 \fatsemi P_2 \triangle Q)$$

In what follows we discuss a way for achieving the sort of recursive definition described above. The specification technique relies on Bulk Synchronous Parallelism (BSP), which consists of placing a synchronisation barrier on the execution lines of parallel executing processes. At the barrier, the data of each process is made available to the other accordingly, and then every process resumes its execution. What we are trying to accomplish is to block further execution of P when it reaches the barrier, whilst still making available the state of P to Q.

Figure5.2 illustrates the BSP mechanism. A synchronisation barrier is represented by a bold vertical line, and annotated with the name of the corresponding synchronisation event, **sync**. Processes are represented by horizontal arrows whose head indicates the direction of the execution flow. Each horizontal line is annotated with the name of the corresponding process. The merge predicate M is represented in the middle, whereas parallel running processes are represented by corresponding parallel horizontal lines.

In algebraic terms, the BSP mechanism is determined by the expansion law for the parallel composition operator.

Recall - Expansion law [66, Chap. 7, §7.5, **L9**]:

The intended effect of synchronisation is most clearly explained by an algebraic law. Let M be the merge operation for *all* the shared variables m of the system. Let P describe the behaviour of one process up to its first **sync** action, and let Q denote the initial non-synchronising behaviour of the other process. Then the **sync** action invokes the merge operation M to consolidate the results of P and Q in their global store, so that results computed separately in m by each of them are available subsequently to both of them. The synchronisation action is retained to deal with the possibility that there might be three or more processes. This informal account is summarized in the following expansion law:

$$\textbf{L9} \qquad (P \fatsemi \textbf{sync} \fatsemi R) \parallel_M (Q \fatsemi \textbf{sync} \fatsemi S) = (P \parallel Q) \fatsemi \textbf{sync} \fatsemi (R \parallel_{\widetilde{M}} S)$$

Here the tilde over M is meant to indicate that M is executed not just once at the end of parallel composition but also at all the intermediate synchronisation point.

For a start, take $n = 2$ viz. $P = P_1 \fatsemi \textbf{sync} \fatsemi P_2$. The behaviour of P upon interrupt (after P_1) should be given by

$$[\mathit{GenInterruptSpec}3] \qquad (P_1 \fatsemi \textbf{sync} \fatsemi P_2) \triangle Q = (P_1 \fatsemi \textbf{sync} \fatsemi P_2) \Box (P_1 \fatsemi \textbf{sync} \fatsemi i \to Q)$$

[*GenInterruptSpec*3] simply states that, after P_1, either interrupt has *not* occurred, in which case we may observe P_2 only, or interrupt has occurred in which case we may observe Q instead.
Let $i \to \textbf{sync} \fatsemi Q$, then:

$$(P_1 \fatsemi \textbf{sync} \fatsemi P_2) \parallel (i \to \textbf{sync} \fatsemi Q) = (P_1 \parallel i \to \mathit{SKIP}) \fatsemi \textbf{sync} \fatsemi (P_2 \parallel Q)$$

This is not satisfactory: it enforces the synchronisation of both P and Q, and yet fails to eliminate P_2.

Consider instead $\mu Y \bullet (\textbf{sync} \fatsemi Y) \Box (i \to Q)$, then

$$\begin{aligned}(P_1 \fatsemi \textbf{sync} \fatsemi P_2) \parallel \mu Y \bullet (\textbf{sync} \fatsemi Y) &= (P_1 \parallel \mathit{SKIP}_{\alpha Q}) \fatsemi \textbf{sync} \fatsemi (P_2 \parallel \mu Y \bullet (\textbf{sync} \fatsemi Y)) \\ &= P_1 \fatsemi \textbf{sync} \fatsemi P_2\end{aligned}$$

and

$$(P_1 \fatsemi \textbf{sync} \fatsemi P_2) \parallel (i \to Q) \;=\; (P_1 \fatsemi \mathit{STOP}) \parallel i \to Q$$

Above, the synchronisation occurs outside $i \to Q$. Then, we have the choice between $P_1 \fatsemi \textbf{sync} \fatsemi P_2$ and something that we do not want i.e. $(P_1 \fatsemi \mathit{STOP}) \parallel i \to Q$. Indeed, although the latter is close enough to $(P_1 \fatsemi \textbf{sync} \fatsemi i \to Q)$, there is no synchronisation, hence

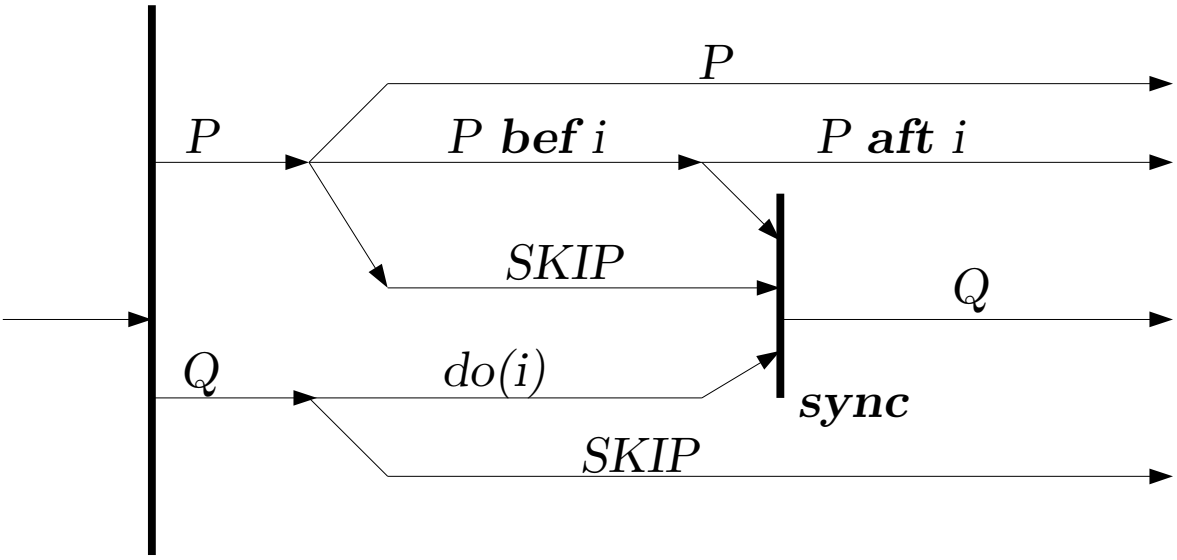

Figure 5.3: Interrupt mechanism with BSP - using a single barrier

the state of P upon interrupt, i.e. here P_1, may not be made available to Q.

Figure5.3 represents what should happen on the **sync** barrier. The first arrow indicates some prior execution; the first vertical line simply indicates the start of the executions of P and Q; it also indicates a possible initial merge. The initial labels P and Q are meant only for indication. The actual behaviour of each process is in fact indicated after the initial arrow. Diverging arrows emerging from a single point represent non-deterministic choice. For example, the line labelled with process P indicates that it may not synchronise at all and hence P may be observed; the line labelled by *SKIP* indicates that P may be interrupted before actually starting; and the line labelled by $P\,\mathbf{bef}\,i$ indicates that P may be interrupted in which case P synchronises on the **sync** barrier and then Q may be observed instead. $P\,\mathbf{aft}\,i$ is merely for indication; it should be considered as *unreachable*.

What we expect from using a synchronisation barrier may be described thus:

- when i does not occur and P_1 reaches **sync** first, the synchronisation is *vacuous*: this is the effect obtained when **sync** is outside $i \to Q$;
- on the other hand, when Q reaches **sync** first, the synchronisation is *blocking*: this is what we expected by placing **sync** after i, as in $i \to \mathbf{sync} \fatsemi Q$.

This suggests that we need two barriers instead of a single one: a skipping one, say **skyp**, and a blocking one, **sync** as before.

Let $R = \big((\mathbf{skyp} \fatsemi R) \;\Box\; (\mathbf{sync} \fatsemi \mathit{SKIP})\big)$. R is the process that continually offers P a vacuous **skyp** synchronisation, until

- either P terminates, in which case R should STOP, and then Q should also STOP;

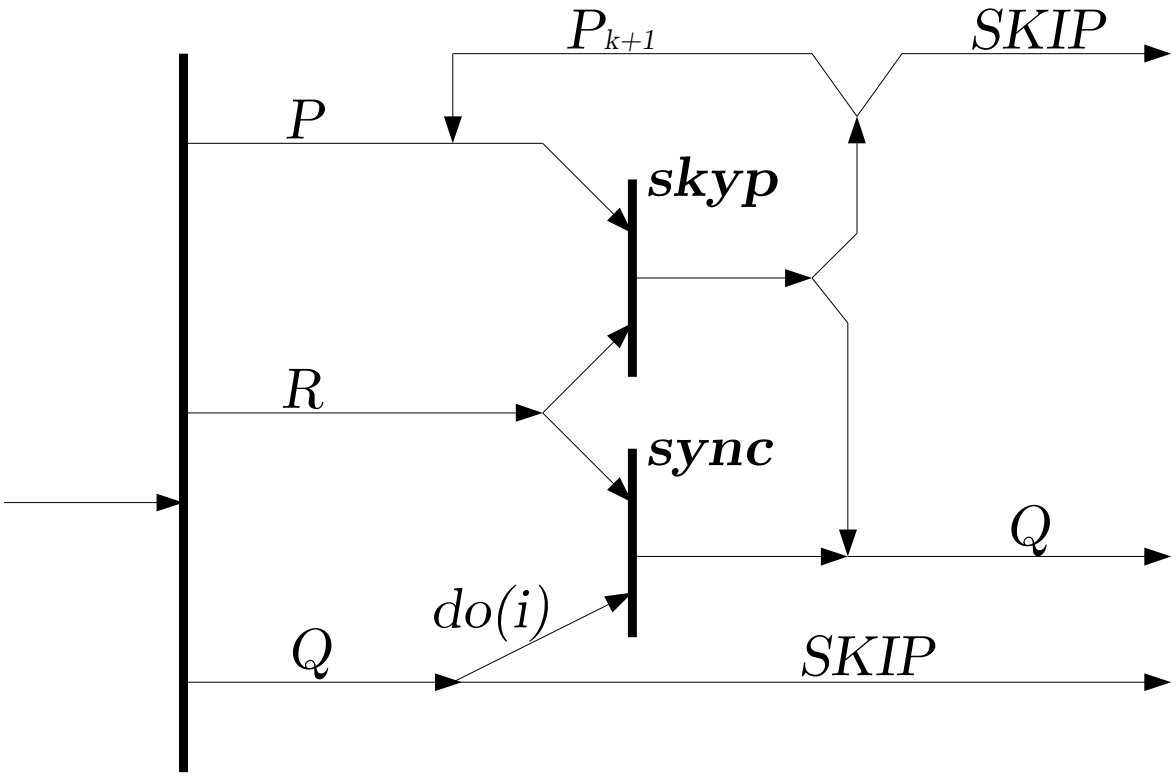

Figure 5.4: Interrupt mechanism with BSP - using two(2) barriers

- or Q synchronises with R on **sync** *before* P terminates thus leading R to termination viz. *SKIP*, in which case then P should *STOP*.

Figure5.4 represents the interrupt mechanism when two barriers are used.

Then:

$$(P_1 \fatsemi \mathbf{skyp} \fatsemi P_2) \parallel (\mathbf{skyp} \fatsemi R \ \Box \ \mathbf{sync} \fatsemi \mathit{SKIP}) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q)$$

$$= \left(\begin{array}{l} ((P_1 \fatsemi \mathbf{skyp} \fatsemi P_2) \parallel (\mathbf{skyp} \fatsemi R) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q)) \ \Box \\ ((P_1 \fatsemi \mathbf{skyp} \fatsemi P_2) \parallel (\mathbf{sync} \fatsemi \mathit{SKIP}) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q)) \end{array} \right)$$

Left-hand-side (lhs) of the choice: According to our initial assumption, P_1 must occur first, so it synchronises with R on **skyp** barrier, and then P_2 follows. So we end up with $P_1 \fatsemi \mathbf{skyp} \fatsemi P_2$. Right-hand-side (rhs) of the choice: Again P_1 must occur first but no synchronisation on **skyp** may occur, so P blocks after P_1 i.e. $P = P_1 \fatsemi \mathit{STOP}$. However, Q may synchronise on **sync**, hence, we do not obtain the expected interleaving. The issue is that we obtain the expected blocking of P_2 when **sync** occurs before **skyp**, whereas we would like the blocking to occur only after **skyp**, but before P_2. That happens because simply assuming that P_1 occurs first is no longer enough: we need to implement it, by enforcing a first synchronisation

on **skyp**. For that, we may redefine $R = \mathbf{skyp} \rightarrow ((\mathbf{skyp} \fatsemi R) \ \Box \ (\mathbf{sync} \fatsemi \mathit{SKIP}))$. Then:

$$
\begin{aligned}
& (P_1 \fatsemi \mathbf{skyp} \fatsemi P_2) \parallel (\mathbf{skyp} \rightarrow (\mathbf{skyp} \fatsemi R \ \Box \ \mathbf{sync} \fatsemi \mathit{SKIP})) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q) \\
= \\
& P_1 \fatsemi \mathbf{skyp} \fatsemi (P_2 \parallel (\mathbf{skyp} \fatsemi R \ \Box \ \mathbf{sync} \fatsemi \mathit{SKIP}) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q)) \\
= \\
& P_1 \fatsemi \mathbf{skyp} \fatsemi \left(\begin{array}{l} (P_2 \parallel (\mathbf{skyp} \fatsemi R) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q)) \ \Box \\ (P_2 \parallel (\mathbf{sync} \fatsemi \mathit{SKIP}) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q)) \end{array} \right)
\end{aligned}
$$

Lhs of choice: R blocks as it can no longer synchronise on **skyp**; similarly Q blocks as it cannot synchronise on **sync**. So, we obtain $P_1 \fatsemi \mathbf{skyp} \fatsemi P_2$.
Rhs of choice: either P_2 occurs first or i occurs first. Either way Q synchronises with R. We obtain hence $P_2 \parallel (i \rightarrow \mathbf{sync} \fatsemi Q)$, though we were expecting STOP in place of P_2. Yet, if P_2 were not the last step, we could get the expected effect. To see this, let $P = P_1 \fatsemi \mathbf{skyp} \fatsemi P_2 \fatsemi \mathbf{skyp} \fatsemi P_3$ instead. Then, the rhs becomes

$$(P_2 \fatsemi \mathbf{skyp} \fatsemi P_3) \parallel (\mathbf{sync} \fatsemi \mathit{SKIP}) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q)$$

Since P cannot synchronise on **skyp**, P_3 will behave like STOP. This suggests that every step must be guarded by a **skyp** synchronisation, which then acts as a *lock*.

See that $i \rightarrow \mathbf{sync} \fatsemi Q$ appears after every step, although Q itself is not defined recursively. The reason for this comes from the expansion. Since steps are atomic, i may occur only before the step starts executing; otherwise, the only possibility for i to occur again is before the next step, but after the current one. Process R is there to ensure that i never occurs during the execution of the current step by not offering **sync** although Q may be ready. At this stage, it suffices to assume that **sync** will then be selected first on the next step (if it did occur during the execution of the current step).

In conclusion:

$$
\begin{aligned}
& (P_1 \fatsemi \mathbf{skyp} \fatsemi P_2) \parallel (\mathbf{skyp} \rightarrow (\mathbf{skyp} \fatsemi R \ \Box \ \mathbf{sync} \fatsemi \mathit{SKIP})) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q) \\
= \\
& (P_1 \fatsemi \mathbf{skyp} \fatsemi P_2) \Box (P_1 \fatsemi \mathbf{skyp} \fatsemi (\mathit{STOP} \parallel i \rightarrow \mathit{SKIP}) \fatsemi \mathbf{sync} \fatsemi (\mathit{SKIP} \parallel Q)) \\
= \\
& (P_1 \fatsemi \mathbf{skyp} \fatsemi P_2) \Box (P_1 \fatsemi \mathbf{skyp} \fatsemi i \rightarrow \mathbf{sync} \fatsemi Q)
\end{aligned}
$$

On the other hand, for $P = P_1 \fatsemi \mathbf{skyp} \fatsemi P_2 \fatsemi \mathbf{skyp} \fatsemi P_3$, the lhs becomes

$$((P_2 \fatsemi \mathbf{skyp} \fatsemi P_3) \parallel (\mathbf{skyp} \fatsemi R) \parallel (i \rightarrow \mathbf{sync} \fatsemi Q))$$

The latter is simply our initial formulation with P_2 replacing P_1 and P_3 replacing P_2.

In sum, the case $(P_1 \fatsemi \mathbf{skyp} \fatsemi P_2)$ straightforward generalises to the case $P\,\mathbf{bef}\,i \fatsemi P\,\mathbf{aft}\,i$ where $P\,\mathbf{bef}\,i$ stands for the concatenation of all those steps that have successfully synchronised with the environment through **skyp**, and $P\,\mathbf{aft}\,i$ simply stands for the remainder of P that would have been executed had the last **skyp** synchronisation been successful, but was pre-empted by the occurrence of the interrupt event i.

In the next section we give the formal definitions of the concepts introduced thus far.

5.4.3 Semantics of the generic interrupt

In the previous section we gave an informal account of how the generic interrupt operator is meant to work. We shall now precise more succinctly the same.

Let P be an atomic process; let $R = (\mathbf{skyp} \to R) \,\Box\, \mathbf{sync} \to \mathit{SKIP}$. Then:

$$
\begin{aligned}
&(P \parallel \mathbf{sync} \to Q) \parallel ((\mathbf{skyp} \to R) \,\Box\, \mathbf{sync} \to \mathit{SKIP}) \\
=\;& \\
&(P \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to R) \,\Box\, (P \parallel \mathbf{sync} \to Q \parallel \mathbf{sync} \to \mathit{SKIP}) \\
=\;& \\
&(P \parallel \mathit{STOP} \parallel \mathit{STOP}) \,\Box\, \mathbf{sync} \to (P \parallel Q)
\end{aligned}
$$

The rhs of the choice is not satisfying: we need P to behave like STOP when **sync** is the first choice. For this consider $\mathbf{skyp} \to P$ instead, then

$$
\begin{aligned}
&(\mathbf{skyp} \to P \parallel \mathbf{sync} \to Q) \parallel (\mathbf{skyp} \to R \,\Box\, \mathbf{sync} \to \mathit{SKIP}) \\
=\;& \\
&(\mathbf{skyp} \to P \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to R) \,\Box\, (\mathbf{skyp} \to P \parallel \mathbf{sync} \to Q \parallel \mathbf{sync} \to \mathit{SKIP}) \\
=\;& \\
&\mathbf{skyp} \to (P \parallel \mathbf{sync} \to Q \parallel R) \,\Box\, \mathbf{sync} \to (\mathit{STOP} \parallel Q)
\end{aligned}
$$

The lhs of the choice is not satisfying: we need to block R upon P's termination. Consider $R = (\mathbf{skyp} \to \mathbf{skyp} \to R) \,\Box\, \mathbf{sync} \to \mathit{SKIP}$ instead, then

$$
\begin{aligned}
&(\mathbf{skyp} \to P \parallel \mathbf{sync} \to Q) \parallel ((\mathbf{skyp} \to \mathbf{skyp} \to R) \,\Box\, \mathbf{sync} \to \mathit{SKIP}) \\
=\;& \\
&(\mathbf{skyp} \to P \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to \mathbf{skyp} \to R) \,\Box\, (\mathbf{skyp} \to P \parallel \mathbf{sync} \to Q \parallel \mathbf{sync} \to \mathit{SKIP}) \\
=\;& \\
&\mathbf{skyp} \to (P \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to R) \,\Box\, (\mathbf{skyp} \to P \parallel \mathbf{sync} \to Q \parallel \mathbf{sync} \to \mathit{SKIP})
\end{aligned}
$$

$$
\begin{array}{l}
= \\
\quad \mathbf{skyp} \to (P \parallel \mathit{STOP} \parallel \mathit{STOP}) \,\Box\, (\mathit{STOP} \parallel \mathbf{sync} \to Q \parallel \mathbf{sync} \to \mathit{SKIP}) \\
= \\
\quad \mathbf{skyp} \to P \,\Box\, \mathbf{sync} \to Q
\end{array}
$$

The latter result corresponds to what we expected.

Let $P = \mathbf{skyp} \to P_1 \fatsemi \mathbf{skyp} \to P_2$. Then:

$$
\begin{array}{l}
\quad ((\mathbf{skyp} \to P_1 \fatsemi \mathbf{skyp} \to P_2) \parallel \mathbf{sync} \to Q) \parallel ((\mathbf{skyp} \to \mathbf{skyp} \to R) \,\Box\, \mathbf{sync} \to \mathit{SKIP}) \\
=\{\Box \text{ lhs}\} \\
\quad ((\mathbf{skyp} \to P_1 \fatsemi \mathbf{skyp} \to P_2) \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to \mathbf{skyp} \to R) \\
= \\
\quad \mathbf{skyp} \to ((P_1 \fatsemi \mathbf{skyp} \to P_2) \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to R) \\
= \\
\quad \mathbf{skyp} \to P_1 \fatsemi \mathbf{skyp} \to (P_2 \parallel \mathbf{sync} \to Q \parallel R)
\end{array}
$$

Above we have left out the case when **sync** is the first choice, for conciseness. The other case leads (last line) to the same problem we encountered in our first attempt earlier (atomic case, rhs problem). Again, we would need to guard P_2 so that it can be eliminated in case **sync** is the first choice.

The general pattern is that we need to guard each step individually on the entry of the **skyp** barrier in order to dismiss Q, and on exit, again on **skyp**, in order to save the last executed step. However, the exit **skyp** of the last step may not coincide with the entry **skyp** of the next one, in order to allow the possibility for Q to pre-empt the next step. If Q does not, then we execute the step and then synchronise on exit, leaving the possibility for Q to pre-empt the next one. We repeat this cycle until either P terminates, in which case R will block on the exit **skyp** of P, or Q interrupts P, in which case R will terminate, hence P will block on the entry **skyp** of its next step.

Let $\mathit{bsp}(X) \mathrel{\widehat{=}} \mathbf{skyp} \to \mathit{SKIP} \fatsemi X \fatsemi \mathbf{skyp} \to \mathit{SKIP}$. Then, if P is atomic:

$$
\begin{array}{l}
\quad (\mathit{bsp}(P) \parallel \mathbf{sync} \to Q) \parallel ((\mathbf{skyp} \to \mathbf{skyp} \to R) \,\Box\, \mathbf{sync} \to \mathit{SKIP}) \\
=\{\Box \text{ lhs}\} \\
\quad (\mathbf{skyp} \to P \fatsemi \mathbf{skyp} \to \mathit{SKIP}) \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to \mathbf{skyp} \to R) \\
= \\
\quad \mathbf{skyp} \to ((P \fatsemi \mathbf{skyp} \to \mathit{SKIP}) \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to R) \\
= \\
\quad \mathbf{skyp} \to P \fatsemi \mathbf{skyp} \to (\mathit{SKIP} \parallel \mathbf{sync} \to Q \parallel R)
\end{array}
$$

This means that the last step of P must not synchronise with R on termination, otherwise Q may still be observable. Let $P = bsp(P_1) \fatsemi \mathbf{skyp} \to P_2$

$$
\begin{aligned}
& ((bsp(P_1) \fatsemi \mathbf{skyp} \to P_2) \parallel \mathbf{sync} \to Q) \parallel ((\mathbf{skyp} \to \mathbf{skyp} \to R) \Box \mathbf{sync} \to SKIP) \\
= & \{\Box \text{ lhs}\} \\
& (bsp(P_1) \fatsemi \mathbf{skyp} \to P_2) \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to \mathbf{skyp} \to R) \\
= & \\
& (\mathbf{skyp} \to P_1 \fatsemi \mathbf{skyp} \to SKIP \fatsemi \mathbf{skyp} \to P_2) \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to \mathbf{skyp} \to R \\
= & \\
& \mathbf{skyp} \to ((P_1 \fatsemi \mathbf{skyp} \to \mathbf{skyp} \to P_2) \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to R) \\
= & \\
& \mathbf{skyp} \to P_1 \fatsemi \mathbf{skyp} \to (\mathbf{skyp} \to P_2 \parallel \mathbf{sync} \to Q \parallel R) \\
= & \\
& \mathbf{skyp} \to P_1 \fatsemi \mathbf{skyp} \to ((\mathbf{skyp} \to P_2) \Box \mathbf{sync} \to Q)
\end{aligned}
$$

That is what we expected.

The semantics of the generic interrupt are given subsequently.

Definition 5.4.1 (Generic interrupt)**.** *Let* $P \mathrel{\widehat{=}} (\fatsemi_{1 \le i < n} \bullet bsp_{\mathbf{skyp}}(P_i)) \fatsemi \mathbf{skyp} \to P_n$.

$$
P \triangle Q \mathrel{\widehat{=}} \left(\begin{array}{l} P \parallel \mathbf{sync} \to Q \parallel \\ \mu\, X \bullet ((\mathbf{skyp} \to \mathbf{skyp} \to X) \Box \mathbf{sync} \to SKIP) \end{array} \right) \setminus \{\mathbf{skyp}, \mathbf{sync}\} \qquad \Box
$$

A comment about the previous definition may be necessary, regarding the form of P. Let us write $bsp(P)$ instead, for ease of reference, i.e. $bsp(P) \mathrel{\widehat{=}} (\fatsemi_{1 \le i < n} \bullet bsp_{\mathbf{skyp}}(P_i)) \fatsemi \mathbf{skyp} \to P_n$. In a few words, the form of $bsp(P)$ infers that *interruptible processes* have a structure of their own.

Indeed, a way of seeing the generic interrupt operator $\triangle$ is as embedding a predicate transformer, notably *bsp*, which transforms a UTP-CSP process into one with a given form. Then, $bsp(P)$ would truly denote the transformation of P into an equivalent process that allows interrupting P.

$bsp(P)$, by introducing synchronisation events into a process P that could have first been entirely sequential makes it possible for that process to be interruptible also under the catastrophic interrupt operator (cf. Chap. 2, §2.5.1). Recall that the catastrophic interrupt requires the interruptible process to be in a waiting state before interrupt can occur.

Following the preceding remark, we may question the possibility of embedding synchronisation events such as **sync** and **skyp** within other interaction events, including the interrupt event i. Then, every interaction of P with the environment that may put P in a waiting state may be taken for **skyp**; and i may be taken for **sync**. Under the proposed embedding,

the catastrophic interrupt appears to be quite similar to the generic interrupt. However, this similitude does not mean that both operators are equivalent.

In [134], Kun Wei shows that the generic interrupt is more expressive than the catastrophic interrupt, concerning their provided definitions under Reactive Designs. Unfortunately we have not carried out a similar comparison between the semantics of the generic interrupt proposed in this thesis and the semantics of the catastrophic interrupt defined by McEwan & Woodcock [84]. Notwithstanding, the use of the parallel composition operator $\|$ permits to infer that our proposed definition is more expressive than the catastrophic interrupt simply from the fact it allows the possibility for P and Q to interact with each other.

More interestingly, this possibility of defining the catastrophic interrupt in terms of the generic interrupt further suggests that the generic interrupt may serve as a basis for defining a *canonic form* for any interrupt operator.

In what follows we prove certain laws for the generic interrupt defined above. These laws were given by Hoare in [65, §5.4] and give us a criterion for validating our definition. Of these laws, the step-law for the interrupt operator is generally considered as characteristic for the operator. We will therefore consider that the generic interrupt operator $\triangle$ is correct if it obeys the step-law for interrupt.

L1 (Step-law) $(a \to P) \triangle Q = (a \to (P \triangle Q)) \Box Q$

Proof.

$$
\begin{array}{ll}
 & (a \to P) \triangle Q \\
= & \{\triangle \text{ def}\} \\
 & \big((bsp(a \to P) \parallel \mathbf{sync} \to Q) \parallel ((\mathbf{skyp} \to \mathbf{skyp} \to R) \Box \mathbf{sync} \to SKIP)\big) \setminus \{\mathbf{skyp}, \mathbf{sync}\} \\
= & \{\| \text{ distrib}\} \\
 & \left(\begin{array}{l} (bsp(a \to P) \parallel \mathbf{sync} \to Q \parallel \mathbf{skyp} \to \mathbf{skyp} \to R) \Box \\ (bsp(a \to P) \parallel \mathbf{sync} \to Q \parallel \mathbf{sync} \to SKIP) \end{array} \right) \setminus \{\mathbf{skyp}, \mathbf{sync}\} \\
= & \{bsp(a \to P) = \mathbf{skyp} \to (a \to SKIP) \fatsemi \mathbf{skyp} \to bsp(P)\} \\
 & (a \to (P \parallel \mathbf{sync} \to Q \parallel R) \Box \mathbf{sync} \to Q) \setminus \{\mathbf{skyp}, \mathbf{sync}\} \\
= & \{\text{hiding def, } \triangle \text{ def}\} \\
 & a \to (bsp(P) \triangle Q) \Box Q
\end{array}
$$

$\Box$

L2 $(P \triangle Q) \triangle S = P \triangle (Q \triangle S)$

Proof. If P is a single, atomic step then $P \triangle Q = P \Box Q$. Then

$$
\begin{array}{ll}
 & (P \triangle Q) \triangle S \\
= & \{P \text{ is atomic}\} \\
 & (P \Box Q) \triangle S \\
= & \{\triangle \text{ distrib}\} \\
 & (P \triangle S) \Box (Q \triangle S) \\
= & \{P \text{ is atomic}\} \\
 & P \Box S \Box (Q \triangle S)
\end{array}
$$

Reciprocally

$$
\begin{array}{ll}
 & P \triangle (Q \triangle S) \\
= & \{P \text{ is atomic}\} \\
 & P \Box (Q \triangle S) \\
= & \{\text{ step-law}\} \\
 & P \Box ((Q \triangle S) \Box S) \\
= & \{\Box \text{ assoc}\} \\
 & (P \triangle Q) \triangle S
\end{array}
$$

If P is not atomic, we would obtain the same result as above by replacing P by $P \,\mathbf{bef}\, i$, accordingly. □

L3 $P \triangle STOP = P = STOP \triangle P$

Proof. From $P \parallel STOP = P$ and $(P_1 \fatsemi \mathbf{sync} \fatsemi P_2) \parallel \mathbf{sync} \fatsemi SKIP = P_1 \fatsemi P_2 = P$ □

L4A $P \triangle (Q \sqcap S) = (P \triangle Q) \sqcap (P \triangle S)$
L4B $(Q \sqcap S) \triangle P = (Q \triangle P) \sqcap (S \triangle P)$

Proof. From $P \parallel (Q \sqcap S) = (P \parallel Q) \sqcap (P \parallel S)$ □

L5 $CHAOS \triangle P = CHAOS = P \triangle CHAOS$

Proof. From $CHAOS \parallel P = CHAOS$ □

Example 5.4.2.

1. Suspendable serial integrator. Let

$$bsp(SIntegrate) \;\widehat{=}\; total := 0 \fatsemi \mu X \bullet (\mathbf{skyp} \rightarrow in?x \rightarrow total := total + x \fatsemi out!total \rightarrow X)$$

The process bsp(SIntegrate) $\triangle$ Q, Q a given interrupting process, describes a serial integrator that may be suspended only before realising an input.

In contrast, let us use the catastrophic interrupt instead (cf. Def. 2.5.26). The process SIntegrate $\triangle_{iev}$ Q, Q a given interrupting process, iev the associated interrupt event, describes a serial integrator that may be suspended either when waiting for input, or when waiting for output.

2. *Suspendable parallel integrator. Let*

$$\begin{aligned} bsp(PIntegrate) &\mathrel{\widehat{=}} (bsp(Plus) \parallel Delta \parallel Prefix) \setminus \{a, b, c\} \\ bsp(Plus) &\mathrel{\widehat{=}} \mathbf{skyp} \rightarrow (in?x \rightarrow SKIP \parallel c?y \rightarrow SKIP) \fatsemi a!(x + y) \rightarrow Plus \end{aligned}$$

The process bsp(PIntegrate) $\triangle$ Q, Q a given interrupting process, describes a parallel integrator that may be suspended only before realising an input. The semantics of bsp(PIntegrate) are quite convenient: blocking the process Plus triggers a blocking of both processes Delta and Prefix. Also, neither of the latter two have an internal state, otherwise both internal states would have been lost (according to the previous semantics). To prevent such loss it would have been necessary to insert instances of skyp in both Delta and Prefix also.

In contrast, the process PIntegrate $\triangle_{iev}$ Q, Q a given interrupting process, iev the associated interrupt event, describes a parallel integrator that may be suspended either when waiting for input (in channel), or when waiting for output (out channel). Communications on channels a, b, c are hidden, hence internal waiting on either of them may not allow the interrupt event iev to occur. Unlike bsp(PIntegrate) however, if either Delta or Prefix had an internal state, it would not have been possible to specify where to save that state. Hence Delta and Prefix would have to terminate for their internal state to be available, or else, they would have need to engage in an interaction with the environment. □

Semantics of skyp and sync

Def. 5.4.1 is incomplete, it says nothing of the semantics of **skyp** and **sync**. So far they have been considered to be *synchronisation actions* i.e. in $\mathcal{A}$. This notably permits them to play the blocking/ordering role that is expected of them in the semantics of the generic interrupt operator $P \triangle Q$. However, being synchronisations is not enough for them to allow passing the interrupt state of P to Q, although that is enough for characterising P's interrupt state.

In order to pass the interrupt state of P to Q, a simple solution consists of using communications instead of pure synchronisations. Then P stores a copy of its state in the environment on every **skyp** event, and Q reads that copy on a **sync** event.

Definition 5.4.3 (Generic interrupt (2))**.** *Let v denote all the variables of a process, including eventually the control variable $\mathcal{L}$, and excluding observational variables, e.g. ok, wait, tr. Let*

$1.v$ *for* P, $2.v$ *for* Q. *Let* $P \mathrel{\widehat{=}} (\fatsemi_{1 \leq i < n} \bullet bsp_{\mathbf{skyp}}(P_i)) \fatsemi \mathbf{skyp}!1.v \rightarrow P_n$.

$$P \triangle Q \mathrel{\widehat{=}} \left(\begin{array}{l} P \parallel \mathbf{sync}?1.vcopy \rightarrow Q \parallel \\ \mu\, X \bullet ((\mathbf{skyp}?y \rightarrow \mathbf{skyp}?y \rightarrow X) \Box \mathbf{sync}!y \rightarrow SKIP) \end{array} \right) \setminus \{\mathbf{skyp}, \mathbf{sync}\} \qquad \Box$$

Above, P saves its data twice, which is redundant. Since the second $\mathbf{skyp}?y$ is necessary only to obtain the expected ordering of events, it may be replaced by a simple synchronisation event.

In the rest of this chapter, we will use **skyp** (resp. **sync**) as an abbreviation for both $\mathbf{skyp}!1.v$ and $\mathbf{skyp}?y$ (resp. $\mathbf{sync}!y$ and $\mathbf{sync}?1.vstore$), according to the previous definition.

In the next section we present the semantics of strong mobility.

5.5 Semantics of process strong mobility

In this section, we present the semantics of process mobility as a particular interrupt operator where the interrupting or *moving* process is responsible for the movement of the interruptible or *movable* process.

Strong mobility requires interrupting the running process and saving its interrupt state, including the continuation of the next instruction as given by the control variable $\mathcal{L}$ (cf. §5.2). Hence, the semantics of strong mobility presented subsequently apply exclusively to movable processes that are Reactive Blocks (cf. §5.2.3).

However, it is not necessary that the moving process be also a reactive block. Since reactive blocks are also UTP-CSP processes, we only have to encapsulate or hide the control variable $\mathcal{L}$.

The following definition takes into account the fact that a process may be moved only when considered as the value of a higher-order process variable.

Definition 5.5.1 (Strong mobility)**.** *Let P be a reactive process block that may be moved during its activation. The mobility of P may be characterised by the postfix unary operators $\triangle_m$.*

$$\begin{aligned} P \triangle_m &\mathrel{\widehat{=}} \mathbf{proc}\, h_1 := \{\mid P \mid\} \fatsemi \xi(\langle h_1 \rangle) \triangle m_o P \\ m_o P &\mathrel{\widehat{=}} move!(h_1, P.st) \rightarrow SKIP \end{aligned}$$

where $P.st$ stands for the interrupt state of P; $\triangle$ stands for any known interrupt operator; and $\xi(P) \mathrel{\widehat{=}} \mathbf{var}\, \mathcal{L} := \mathbf{n} \fatsemi P \fatsemi \mathbf{end}\, \mathcal{L}$ (cf. [66, Chap. 6, §6.4, Def. 6.4.16]).
The resume operation on the target location would be:

$$m_i P \mathrel{\widehat{=}} move?(h_2, h_2.st) \rightarrow (jump\, h_2.\mathcal{L}) \fatsemi \langle h_2 \rangle (h_2.st) \qquad \Box$$

Example 5.5.2.

1. *Movable serial integrator. Let*

$$bsp(SIntegrate : S \Rightarrow F) = \left(\begin{array}{l} \langle m_0, \mathbf{skyp} \rightarrow SKIP, m_1 \rangle \fatsemi \langle m_1, Init, m_2 \rangle \fatsemi \\ \mu\, X \bullet \langle m_2, Input, m_3 \rangle \fatsemi \langle m_3, Add, m_4 \rangle \fatsemi \\ \langle m_4, Output, m_2 \rangle \fatsemi X \end{array} \right)$$

   The process $bsp(SIntegrate : S \Rightarrow F) \triangle_m$ *describes a movable serial integrator suspended with the generic interrupt* $\triangle$.

2. *The process* $(SIntegrate : S \Rightarrow F) \triangle_{iev,m}$ *describes a movable serial integrator suspended with the catastrophic interrupt* $\triangle_{iev}$.

3. *Movable parallel integrator. Let*

$$bsp(PIntegrate : S \Rightarrow F) = (bsp(Plus : S \Rightarrow F) \,//\, Delta : S \Rightarrow F \,//\, Prefix : S \Rightarrow F) \setminus \{a, b, c\}$$

$$(Plus : S \Rightarrow F) = \left(\begin{array}{l} \langle m_0, \mathbf{skyp} \rightarrow SKIP, \mathbf{nn} \rangle \fatsemi \\ (\langle m_1, In(m_1, x)[in \leftarrow ch], m_3 \rangle \,//\, \langle m_2, In(m_2, y)[c \leftarrow ch], m_3 \rangle) \fatsemi \\ \langle m_3, Out(m_3, x + y)[a \leftarrow ch], \mathbf{nn} \rangle \fatsemi Plus \end{array} \right)$$

   The process $bsp(PIntegrate : S \Rightarrow F) \triangle_m$ *describes a movable parallel integrator suspended with the generic interrupt* $\triangle$.

4. *The process* $(PIntegrate : S \Rightarrow F) \triangle_{iev,m}$ *describes a movable parallel integrator suspended with the catastrophic interrupt* $\triangle_{iev}$. □

5.6 Discussion

In what follows we discuss the results presented in this Chapter.

Continuations. The definition of continuations for UTP-CSP applies more generally to parallel programs. That is the first definition of continuations for CSP processes. Using a control variable l (resp. $\mathcal{L}$) may be misleading in the case when l is conceived of as just another variable. Rather, l must be conceived of as denoting the predicate whose label it contains. Then, using l instead of functions concurs to the simplicity of the proposed model. Indeed, it is very likely that trying to migrate this work to pure CSP [65], [106] may rather require formulations closer to the ones based on functions, e.g. [46] to [41].

This work may serve as a basis for reasoning about control flow for CSP programs, and also their compilation. Thanks to continuations, we could give a characterisation of sequential programs that is not syntactic, using the healthiness condition ***RPBSeq*** (cf. §5.2.3). This

extension of CSP with continuations finds an immediate application in the definition of strong mobility.

In [72], Jahnig et al. define continuations semantics for a CSP-like language. They do not actually deal with CSP itself, and the language that they consider is sequential. [3] discusses compilation and scheduling of real-time programs on the basis of UTP.

Interrupt. The semantics for the generic interrupt that we have proposed are perhaps the first attempt of a definition of the generic interrupt that does not rely on *time*, and that is based of Bulk Synchronisation Parallelism.

An advantage of the technique used is that we may characterise the interrupt state of a program without any further machinery. We are not aware of a similar result in the literature.

A second positive aspect of the definition is that it opens the way for the definition of other interrupt operators. Indeed, when process migration is not involved, we may readily define the operator that suspends a process and then returns control to that process, which may then resume seamlessly, locally.

We have already mentioned the possibility that our formulation may provide some ground for a canonical formulation of many, if not all other interrupt operators. This hypothesis requires further investigation. Notwithstanding, we have seen that interruptible programs have a structure of their own. That structure would also be worth investigating further as its existence suggests that providing semantics to interrupts may be done at a lower level abstraction than existing attempts.

A possible weakness of the proposed operator may be that of *state explosion*, for model checking. Certainly, the question here may be more that of the scale of the applications that the proposed definition may permit to formalise.

Another possible weakness is that the definition may rely on multiple merges, hence making program verification difficult.

In [26], Brookes discusses the issue of *fairness* when dealing with the expansion law for parallel composition, and shows notably that not every expansion is fair. Whilst fairness is not an issue in itself since we may always assume the existence of a fairness function that makes definitions fair, or either that a fair implementation is always possible, we may want to prove a statement such as: *if the interrupt event i occurs, then P will be interrupted.* A possible way of making such a proof possible might be by using the concept of *priority* in the definition of interrupts, such that an interrupt event is always chosen over some other event (with a lower priority) occurring at the same time. In particular, the *priority* mechanism would need to be implemented for R only since it is R that may resolve the choice between synchronising on **skyp** with P, or on **sync** with Q, whenever both synchronisation events occur at the same time. The priority mechanism may be useful also for discriminating amongst different interrupt events. The implementation in UTP of a scheduler with priority is discussed in [3].

Strong mobility. Strong mobility has been defined as a form of interrupt operator that involves the migration of the interruptible process. We are not aware of any other denotational

model for strong mobility, and we do not think that any has been defined using operational semantics either. We have already mentioned the work in [131] by Todoran, in which a first step towards a denotational semantics in the context of Object Oriented programming is presented. The Higher-order pi-calculus [112] (an operational semantics framework) is limited to weak mobility.

5.7 Strong process mobility vs. Channel mobility

As stated at the beginning of this thesis, mobility requires the definition of the entity that moves, and the space in which it moves. Trivially hence, what distinguishes process mobility from channel mobility is that processes and channels move in each of them, respectively. From a modelling point of view, it is interesting to determine if one can be used to encode the other. That is the question of interest in this section.

In the literature relative to the pi-calculus [86], [112], it is common to consider channel mobility and process mobility as dual notions. The duality comes from the following view. Let P be a process that may communicate through channel a with another process Q. Such a situation may be modelled by the process $(\{a\}, P) \parallel (\{a\}, Q)$, where we explicitly mention the interface of processes for the sake of the present discussion. Let R be a process that may communicate with Q through a channel b, but may not communicate with P. Such a situation may be modelled by the process $(\{a\}, P) \parallel (\{a, b\}, Q) \parallel (\{b\}, R)$. In the pi-calculus, the *scope* of a process determines with what other process it may communicate, e.g. Q is in the scope of P, and R is not.
Let Q pass its b channel to P. Such a situation may be modelled by the process $(\{a, b\}, P) \parallel (\{a\}, Q) \parallel (\{b\}, R)$. Let us use brackets $[]$ to represent the scope of P, then the situation just described may be pictured as follows:

$$\big[(\{a\}, P) \parallel (\{a, b\}, Q)\big] \parallel (\{b\}, R) \longrightarrow \big[(\{a, b\}, P) \parallel (\{a\}, Q) \parallel (\{b\}, R)\big]$$

R is now in the scope of P, as if R itself had moved: it has moved into P's scope.

In the pi-calculus, there is no explicit representation of *locations*, however. We may nonetheless infer that channels move from one process (viz. its interface) to another. Channel mobility implies process mobility, from one scope to another, whence the duality mentioned above. That is, interfaces are locations for channels, whilst scopes are locations for processes. Rigorously, the view that processes move because they change of scope does not correspond to the concept of process mobility; rather, that is simply *scope extrusion* or expansion.

Process mobility requires the movement of an *execution logic*, in the form of process code, from one process to another. In the scenario presented above, the computation realised by R never *left* R, to execute on P.

Cardelli et al. [32] propose another analogy between channel mobility and process

mobility, using a different concept of location.

Weak mobility, which is the movement of process code without any execution state requires that the receiving process has all of the resources for executing that code. In fact, code is just data, executable data. Thus weak mobility is but a form of data mobility, a form of message-passing. The main difference with strong mobility is that the latter involves an *interrupt mechanism*, which is local to the sending process. Granting successful suspension, what follows is just code mobility. Thus, process mobility is not equivalent to channel mobility since data mobility is not equivalent to channel mobility.

From a network topology view, strong mobility does not trigger channel mobility either. The assumption that the receiving environment has all the resources necessary for the moved process to execute includes channels as well. For example, commercial software that are meant to run on a network are installed on computers that already have the required network resources, especially channels. When a commercial software is removed, the computer resources are not removed as well. Thus if the software is communicated to another computer as part of the network logic, the software movement does not trigger a corresponding movement of the channels in its interface. Just like a software that requires $1Gb$ (Gigabyte) of memory may not be run on a computer that may provide $1Ko$ (Kilo octet) of memory only; just like a Web application will not run on a computer not connected to the Internet, so to send a process in an environment that does not already have the process's channels will result in the process not being able to run (its interface will not coincide with that of the higher-order variable meant to receive it.)

Finally, from a formal point of view, in UTP, both weak and strong mobility are modelled as operators, whilst channel mobility is a UTP theory. Also, static CSP is enough for passing processes, but may not be used for passing channels (cf. Chap. 4).

Chapter 6

Conclusion

The formal representation of the concept of mobility has been developed in UTP. Many forms of mobility may be distinguished according to what entity moves and the space in which it moves. Tang & Woodcock have provided a model for weak process mobility in [126]. We have extended their work with a discussion of the formalisation of weak process mobility, based on the informal description of code mobility of Fuggeta et al. [49] (cf. §3.1.2). The higher-order pi-calculus HOpi [112] also allows modelling weak mobility, though using operational semantics.

The literature on formal semantics for strong mobility is very scarce, and only Todoran [131] makes a step towards one. Their semantic domain is provided by a programming language however, instead of a mathematical domain.

The state of the art in formal semantics for channel mobility contrasts with that of strong mobility. Many frameworks have been propounded for reasoning about channel mobility, the most important one being the pi-calculus [86]. The semantics of the pi-calculus are operational, and channels are communicated through other channels as messages. Many variants of the pi-calculus have been developed, a comprehensive survey of which may be found in [117].

On the side of denotational semantics, much less work may be found. Work on extending FOCUS with mobility constitute the first attempt of extending an existing denotational framework, FOCUS [27], with mobility; e.g. [56], [58], [123]. FOCUS is concerned with data flow networks, hence these works may not permit to reason about refusals [66, §8.3].

There is no *direct* extension of CSP with mobility in the literature. Roscoe [109, §20.3] discusses an eventual extension of (the denotational semantics) of CSP with channel mobility, based on closed-world semantics. However, we have argued (cf. §3.2.6) that Rosoce's definition of closed-world is ambiguous. We have proposed a sound definition of the concepts of open- and closed-world semantics in §4.6.4.

In [107] and [108], Roscoe discusses the expressive power of CSP (the operational semantics), notably to express channel mobility. In [107], Roscoe defines so-called CSP-like operators, but we have pointed out (cf. §3.2.6) that that definition was problematic as there

is no definition of operational semantics that is independent of CSP, unlike for example, the framework of deSimone [120]. In light of [120], we have argued that the linearisation procedure propounded in [107] does not play a great role if any at all, in relating CSP operational semantics with that of other frameworks, notably the pi-calculus.

In [108], Roscoe defines some encodings of the pi-calculus on the basis of CSP. We have remarked (cf. §3.2.6) that that work did not rely on the linearisation of [107], and also that no relation is actually established between the operational semantics of the pi-calculus, and its proposed CSP models. Furthermore, many sets of names have been introduced in [108] that are difficult to relate to the traditional semantics of CSP. All these models rely on a renaming operator viz. the generalised relabelling operator (cf. Def. 3.2.13), which makes their semantics quite complex. As we have argued in §4.6.3, the problem of traces semantics for expressing channel mobility is not that channels must appear within specified trace scopes (i.e. parts/subsequences of the trace), but that they must belong to the specified interface. The latter remark applies also to the work of Bialkiewicz & Peschanski [18].

[18] proposes a CSP-like traces semantics for the scope extrusion mechanism of the pi-calculus. Their proposed localised traces is quite complex, however, making it difficult to relate to CSP.

Two other works have been presented which use a CSP-like language as their basis. Vajar et al. [132] have proposed an extension of CSP||B with channel mobility in which only the links to B machines may be moved. This clearly restricts the expressiveness of their language. Hoare & O'Hearn [67] have proposed a CSP-like language for channel mobility, using concepts from separation logic [91], [89], [105]. Their traces model is alphabetised, like the one that we have proposed. However, refusals are not studied in [67].

In general however, none of the traces models proposed in the Literature discusses the question of the *characterisation* of channels. That discussion is one of the contributions of this thesis, and has notably permitted us to introduce the concept of *capability* (cf. §4.2), to separate the notion of *knowledge of the existence* of a channel to the notion of *ownership*, modelled by the interface of a process. We have argued that without such a separation, existing models in the Literature are not sound. Thus, dynamic network processes must have a *static* capability and a dynamic interface.

Welch & Barnes [138] have proposed a CSP model for the channel mobility mechanism of the programming language occam-pi [136]. As discussed in §3.2.2, [138] model is not abstract enough, and we have proposed a way of simplifying it. From that simplification, a CSP model for occam-pi may readily be built based on the semantics provided in this thesis (cf. §4.3).

Besides mechanisms for passing channels, two new operators have been defined. Dynamic renaming (cf. §4.3.6) permits to rename eventual new names. The renaming operator is very useful for reusing processes. Hence, for example, it is possible to send the same channel to say three copies of the same process running in parallel, yet have each communicate with a distinct third process. This is the first semantics for that operator in either operational or denotational semantics.

Dynamic hiding (cf. §4.4) permits to hide eventual new names. This operator is useful to model networks whose internal topology may grow silently, which is the case for many large computer networks. The hiding operator is one of the most difficult to define in the presence of channel mobility, and this difficulty is reflected in its semantics, especially when compared with dynamic renaming. Indeed, we had to extend the theory of mobile processes in order for dynamic hiding to be expressible. The resulting theory is the theory of *silencing* processes. This is also the first semantics for the dynamic hiding operator in either operational or denotational semantics. Overall, our treatment of hiding allows disregarding issues related to internal and external interface, closed- and open-world semantics, issues which may greatly cloud understanding and reasoning.

Another contribution of this thesis is the definition of the link *iDgen* (viz. $dn2sn$), §4.5.2, which permits to transform a mobile process into a static process. This is the first link between a mobile framework and a static framework in the Literature. The technique used for defining *iDgen* may be used to relate other similar frameworks as well, notably CCS and the pi-calculus. Indeed, we may conjecture that pi-calculus processes may be modelled by snapshot-identified CCS processes when channel-passing operations are involved; otherwise, the pi-calculus is just CCS.

The definition of *iDgen* (viz. $dn2sn$) has required us to extend static CSP with the snapshot-identifier variable denoted by id, thus yielding the theory of (mobile channels static network) *simulation* processes. Such processes are indeed a simulation in the sense that they *do not* communicate channels between themselves, hence there is no change of their respective interfaces. The construction of simulation processes shows that any static framework may be used to *encode* channel mobility, and that a snapshot-identification mechanism is necessary to realise the encoding. The definition of *iDgen* implies that many, if not all the properties of mobile CSP may be verified using static (simulation) CSP. The definition of the inverse link $iDgen^{-1}$ would be an interesting topic for Ever since the development of both CSP and the pi-calculus, the question of their relation has been explored. We believe that this thesis will allow that comparison to move a step forward. Many elements for that comparison have been given throughout this thesis, notably in §4.6. Relating the two frameworks would be an interesting topic for further work. Such a relation may be established through defining a relation between CCS and the pi-calculus as suggested earlier.

The mechanisation of mobile CSP, and of *iDgen* (viz. $dn2sn$), and the definition of operational semantics for mobile CSP are also interesting for future work about channel mobility.

Returning to strong mobility, one of the main problems was the definition of continuations for CSP processes. Continuations permit reasoning explicitly about control flow, and notably allows defining jump features. Continuations have been defined for parallel programs in general (cf. §5.2.2), and then for CSP processes (cf. §5.2.3). This has led to the definition of the control variable denoted by $\mathcal{L}$, to contain the execution location (in the code of a process) of the defining instructions of a process. CSP processes were then extended with $\mathcal{L}$, thus

yielding the theory of *Reactive Process Blocks*.

Reactive process blocks have been used for defining strong mobility, as a particular kind of interrupt operator (cf. §5.5). These are the first semantics for that operator in CSP, and also in the Literature. Depending on what interrupt operator is used in the definition (catastrophic [84], or either generic), the interrupt state is more or less precise.

Although the catastrophic interrupt may be used for defining strong mobility, it is less expressive than the generic interrupt [135]. We have also seen a case where the catastrophic interrupt is less flexible than the generic interrupt (cf. e.g. 5.4.2). The semantics of the generic interrupt proposed in this thesis (cf. §5.4) is based on the Bulk Synchronous Parallelism mechanism, already used in UTP [66, Chap. 7]. The same principle underlies the process mobility mechanism of the occam-pi programming language [136]. The mechanisation of the generic interrupt may be pursued in future work. It would also be interesting to recast the semantics given in this thesis in the domain of Reactive Designs [37], which may notably permit a comparison with Wei's semantics [135].

Finally, applying the results provided in this thesis to more examples is an interesting challenge. Applications for channel mobility, using the pi-calculus, in the domain of security are currently undergoing, and many variants of the pi-calculus for that purpose have been developed. Channel mobility could possibly be used to provide models for pointers, garbage collection, and dynamic binding (in Object-Oriented Programming). It could also be used for reasoning about protocols for ad hoc networks, and also in the emerging domain of programmable networks.

Bibliography

[1] J.R. Abrial, *The B Book: Assigning Programs to Meaning*, CUP 1996.

[2] C.M. Angerer, T.R. Gross, *Parallel Continuation-Passing Style - A Compiler Representation for Incremental Parallelization*, PESPMA'10, 2010.

[3] A.E. Arenas, J.C. Bicarregui, *Applying Unifying Theories of Programming to Real-Time Programming*, in Journal of Integrated Design and Process Science, vol. 10, pp. 69-88, 2006.

[4] R.J.R. Back, J. von Wright, *Trace Refinement of Action Systems*, CONCUR'94, LNCS vol. 836, pp. 367-384, 1994. doi:10.1007/BFb0015020

[5] J.C.M. Baeten, *A Brief History of Process Algebra*, TCS-PA'05, vol. 335, pp. 131-146, 2005. doi:10.1016/j.tcs.2004.07.036

[6] M. Baldi, S. Gai, G.P. Picco, *Exploiting Code Mobility in Decentralized and Flexible Network Management*, MA'97, LNCS vol. 1219, pp. 13-26, 1997. doi:10.1007/3-540-62803-7_20

[7] M. Baldi, G.P. Picco, *Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications*, ICSE'97, pp. 146-155, IEEE, 1998. doi:10.1109/ICSE.1998.671111

[8] F.R.M. Barnes, P.H. Welch, A.T. Sampson, *Barrier Synchronisation for occam-pi*, PDPTA'05, vol. 1, pp. 173-179, 2005.

[9] F.R.M. Barnes, P.H. Welch, *Mobile Data Types for Communicating Processes*, PDPTA'01, vol. 1, pp. 20-26, 2001.

[10] F.R.M. Barnes, P.H. Welch, *Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment*, CPA'01, pp. 243-265, IOS Press, 2001.

[11] F.R.M. Barnes, P.H. Welch, *Prioritised Dynamic Communicating and Mobile Processes*, in Software, IEE Proc., vol. 150, pp. 121-136, 2003. doi:10.1049/ip-sen:20030182

[12] G. Barrett, *occam3 Reference Manual*, INOS Ltd., March 1992. Available at http://www.wotug.org/occam/documentation/oc3refman.pdf.

[13] J. Baumann, F. Kohl, K. Rothermel, M. StraBer, *Mole - Concepts of a Mobile Agent System*, World Wide Web, vol. 1, pp. 123-137, 1998. doi:10.1023/A:1019211714301

[14] K. Bergner, R. Grosu, A. Rausch, A. Schmidt, P. Scholz, M. Broy, *Focusing on Mobility*, Proc. of the 32nd Hawaii Internat. Conf. on Sys. Sci., IEEE, 1999. doi:10.1109/HICSS.1999.773061

[15] L. Bettini, R. de Nicola, *Translating Strong Mobility into Weak Mobility*, MA'01, LNCS vol. 2240, pp. 182-197, 2001. doi:10.1007/3-540-45647-3_13

[16] K. Bharat, L. Cardelli, *Migratory Applications*, MOS'96, LNCS vol. 1222, pp. 131-148, 1996. doi:10.1007/3-540-62852-5_11

[17] J.-A. Bialkiewicz, F. Peschanski, *Logic for Mobility: A Denotational Approach*, Logic, Agents and Mobility (LAM'09), 2009.

[18] J.-A. Bialkiewicz, F. Peschanski, *A Denotational Study of Mobility*, CPA'09, pp. 239-261, 2009. doi:10.3233/978-1-60750-065-0-239

[19] E. Bonnici, P.H. Welch, *Mobile Processes, Mobile Channels and Complex Dynamic Systems*, CEC'09, pp.232-231, IEEE, 2009. doi:10.1109/CEC.2009.4982953

[20] M. Boreale, *On the Expressiveness of Internal Mobility in Name-Passing Calculi*, TCS'98, vol. 195, pp. 205-226, 1998. doi:10.1016/S0304-3975(97)00220-X

[21] G. Boudol, *Notes on Algebraic Calculi of Processes*, Logics and Models of Concurrent Systems, vol. 13, pp. 261-303, NATO ASI Series, 1984. doi:10.1007/978-3-642-82453-1_9

[22] S.D. Brookes, *Idealized CSP: Combining Procedures with Communicating Processes*, MFPS'97, vol. 6, pp. 60-76, 1997. doi:10.1016/S1571-0661(05)80169-0

[23] S.D. Brookes, *Communicating Parallel Processes*, Symposium in Celebration of the work of C.A.R. Hoare, Oxford University, MacMillan, 2000.

[24] S.D. Brookes, A.W. Roscoe, D.J. Walker, *An Operational Semantics for CSP*, Submitted for publication(1986).

[25] S.D. Brookes, *On the Relationship of CCS and CSP*, Automata, Languages and Programming, LNCS vol. 154, pp. 83-96, 1983. doi:10.1007/BFb0036899

[26] S.D. Brookes, *Reasoning About Recursive Processes: Expansion is not Always Fair*, ENTCS, vol. 20, pp. 182-201, 1999. doi:10.1016/S1571-0661(04)80074-4

[27] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber, *The Design of Distributed Systems - An Introduction to FOCUS (revised)*, Tech. Rep., Uni. of Munich, 1993.

[28] M. Broy, *Equations for Describing Dynamic Nets of Communicating Systems*, Recent Trends in Data Type Specification, pp. 170-187, Springer, 1995. doi:10.1007/BFb0014427

[29] M. Broy, *A Model of Dynamic Systems*, ETAPS:FPS'14, LNCS vol. 8415, pp.39-53, 2014. doi:10.1007/978-3-642-54848-2_3

[30] C. Calcagno, P. O'Hearn, Hongseok Yang, *Local Action and Abstract Separation Logic*, LICS'07, pp. 366-378, IEEE, 2007. doi:10.1109/LICS.2007.30

[31] L. Cardelli, A.D. Gordon, *Mobile Ambients*, TCS, vol. 240, pp. 177-213, 2000. doi:10.1016/S0304-3975(99)00231-5

[32] L. Cardelli, *Abstractions for Mobile Computation*, in Secure Internet Programming, Security Issues for Mobile and Distributed Objects, LNCS vol. 1603, pp. 51-94, 1999. doi:10.1007/3-540-48749-2_4

[33] A. Carzaniga, G.P. Picco, G. Vigna, *Designing Distributed Applications with Mobile Code Paradigms*, ICSE'97, pp. 22-32, ACM, 1997. doi:10.1145/253228.253236

[34] A. Cavalcanti, A. Sampaio, J. Woodcock, *Refinement of Actions in Circus*, ENTCS'02, vol. 70, pp. 132-162, Elsevier, 2002. doi:10.1016/S1571-0661(05)80489-X

[35] A. Cavalcanti, J. Woodcock, *A Tutorial Introduction to CSP in Unifying Theories of Programming*, PSSE'04, LNCS vol. 3167, pp. 220-268, Springer-Verlag, 2004. doi:10.1007/11889229_6

[36] A. Cavalcanti, A. Sampaio, J. Woodcock, *Unifying Classes and Processes*, Software & Systems Modeling, vol. 4, pp. 277-296, 2005. doi:10.1007/s10270-005-0085-2

[37] A. Cavalcanti, W. Harwood, J. Woodcock, *Pointers and Records in the Unifying Theories of Programming*, UTP'06, LNCS vol. 4010, pp. 200-216, 2006. doi:10.1007/11768173_12

[38] A. Cavalcanti, M.-C. Gaudel, *A Note on Traces Refinement and the conf relation in the Unifying Theories of Programming*, UTP'08, LNCS vol. 5713, pp. 42-61, 2008. doi:10.1007/978-3-642-14521-6_4

[39] K. Chanchio, X.-H. Sun, *Communication State Transfer for the Mobility of Concurrent Heterogeneous Computing*, Trans. Comp., vol. 53, pp. 1260-1273, IEEE, 2004. doi:10.1109/TC.2004.73

[40] D. Chess, C. Harrison, A. Kershenbaum, *Mobile Agents: Are They a Good Idea?*, IBM Research Report, MOS'96 , LNCS vol. 1222, pp. 25-45, 1996. doi:10.1007/3-540-62852-5_4

[41] G. Ciobanu, E. Todoran, *Continuation Semantics for Concurrency*, Tech. Rep., Formal Methods Laboratory, the Institute of Comp. Sci. of the Romanian Academy, 2014.

[42] G. Cugola, C. Ghezzi, G.P. Picco, G. Vigna, *Analyzing Mobile Code Languages*, MOS'96, LNCS vol. 1222, pp. 91-109, 1996. doi:10.1007/3-540-62852-5_9

[43] O. Danvy, A. Filinski, *Representing control: a Study of the CPS Transformation*, Math. Struct. in Comp. Sci., vol. 2, pp. 361-391, 1992. doi:10.1017/S0960129500001535

[44] O. Danvy, *On Evaluation Contexts, Continuations, and the Rest of the Computation*, CW'04, pp. 13-23, ACM, 2004.

[45] T.I. Dix, *Exceptions and Interrupts in CSP*, Sci. of Comp. Progr., vol. 3, pp. 189-204, 1983. doi:10.1016/0167-6423(83)90010-2

[46] M. Felleisen, D.P. Friedman, B.F. Duba, J.Merrill, *Beyond Continuations*, Tech. Rep., Dpt. of Comp. Sci., Uni. of Indiana, 1987.

[47] M. Felleisen, M. Wand, D. Bruce, D.P. Friedman, B.F. Duba, *Continuations Semantics for Handling Full Functional Jumps*, LFP'88, pp.52-62, ACM, 1988. doi:10.1145/62678.62684

[48] M. Fiore, E. Moggi, D. Sangiorgi, *A Fully-Abstract Model for the pi-calculus*, LICS'96, pp. 43-54, IEEE, 1996. doi:10.1109/LICS.1996.561302

[49] A. Fuggetta, G.P. Picco, G. Vigna, *Understanding Code Mobility*, TSE'98, vol. 24, pp. 342-361, IEEE, 1998. doi:10.1109/32.685258

[50] C. Ghezzi, G. Vigna, *Mobile Code Paradigms and Technologies: A Case Study*, MA'97, LNCS vol. 1219, pp. 39-49, 1997. doi:10.1007/3-540-62803-7_22

[51] J.F. Giorgi, D. LeMetayer, *Continuation-Based Parallel Implementations of Functional Languages*, LFP'90, pp. 209-217, ACM, 1990. doi:10.1145/91556.91648

[52] M. Gordon, H. Collavizza, *Forward with Hoare*, in Reflections on the Work of C.A.R. Hoare, 2010. doi:10.1007/978-1-84882-912-1_5

[53] A.D. Gordon, *Notes on Nominal Calculi for Security and Mobility*, FOSAD'00, pp. 262-330, 2000. doi:10.1007/3-540-45608-2_5

[54] R. Gray, D. Kotz, S. Nog, D. Rus, G. Cybenko, *Mobile Agents for Mobile Computing*, Tech. Rep., Dpt. Of Comp. Sci., Darmouth College, Hanover, 1996.

[55] R. Grosu, K. Stolen, *A Denotational Model for Mobile P2P DFNs without Channel Sharing*, Tech. Rep., Uni. of Munich, Sep. 1996.

[56] R. Grosu, K. Stolen, *Specification of Dynamic Networks*, Tech. Rep., Uni. of Munich, Dec. 1996.

[57] R. Grosu, K. Stolen, M. Broy, *A Denotational Model for Mobile P2P Data Flow Nets with Channel Sharing*, Tech. Rep., Uni. of Munich, May1997.

[58] R. Grosu, K. Stolen, *Stream-Based Specification of Mobile Systems*, FAC'01, vol. 13, pp. 1-31, Springer, 2001. doi:10.1007/PL00003937

[59] W. Harwood, A. Cavalcanti, J. Woodcock, *A Theory of Pointers for the UTP*, ICTAC'08, vol. 5160, pp. 141-155, 2008. doi:10.1007/978-3-540-85762-4_10

[60] E. Hehner, *Predicative Programming part 1*, CACM'84, vol. 27, pp. 134-143, 1984. doi:10.1145/69610.357988

[61] E. Hehner, *Predicative Programming part 2*, CACM'84, vol. 27, pp. 144-151, 1984. doi:10.1145/69610.357990

[62] M. Hennessy, *A Fully-Abstract Denotational Semantics for the pi-calculus*, TCS'02, vol.278, pp. 53-89, 2002. doi:10.1016/S0304-3975(00)00331-5

[63] R. Hieb, R.K. Dybvig, *Subcontinuations*, LISP and Symbolic Computation, vol. 7, pp. 83-110, 1994. doi:10.1007/BF01019946

[64] C.A.R. Hoare, *Proof of Correctness of Data Representations*, Acta Informatica, vol. 1, pp. 271-281, 1972. doi:10.1007/BF00289507

[65] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[66] C.A.R. Hoare, J. He, *Unifying Theories of Programming*, Prentice-Hall, 1998.

[67] C.A.R. Hoare, P. O'Hearn, *Separation Logic Semantics for Communicating Processes*, ENTCS'08, vol. 212, pp. 3-25, Elsevier, 2008. doi:10.1016/j.entcs.2008.04.050

[68] Y. Huang, H. Zhu, Y. Zhao, J. Shi, S. Qin, *Investigating Time Properties of Interrupt-Driven Programs*, SBMF'12, LNCS vol. 7498, pp. 131-146, 2012. doi:10.1007/978-3-642-33296-8_11

[69] Y. Huang, Y. Zhao, J. Shi, H. Zhu, *A Denotational Model for Interrupt-Driven Programs*, ICSTW'13, pp. 15-20, IEEE, 2013. doi:10.1109/ICSTW.2013.9

[70] Y. Huang, J. He, H. Zhu, Y. Zhao, J. Shi, S. Qin, *Semantic Theories of Programs with Nested Interrupts*, FCS'14, vol. 9, pp 331-345, Higher Education Press, 2014. doi:10.1007/s11704-015-3251-x

[71] G. Hutton, J. Wright, *What is the Meaning of These Constant Interruptions? (Extended Version)*, JFP'07, vol. 17, pp. 777-792, Cambridge University Press, 2007. doi:10.1017/S0956796807006326

[72] N. Jahnig, T. Gothel, S. Glesner, *A Denotational Semantics for Communicating Unstructured Code*, FESCA'15, EPTCS, vol.178, pp. 9-21, 2015. doi:10.4204/EPTCS.178.22015

[73] J.B. Jensen, N. Benton, A. Kennedy, *High-Level Separation Logic for Low-Level Code*, POPL'13, vol. 48, pp. 301-314, ACM, 2013. doi:10.1145/2480359.2429105

[74] O. Jensen, R. Milner, *Bigraphs and Mobile Processes (revised)*, Technical Report UCAM-CL-TR-580, Uni. of Cambridge, UK, 2004.

[75] W. Kahl, *Refinement and Development of Programs from Relational Specifications*, ENTCS'03, vol. 44, pp. 51-92, 2003. doi:10.1016/S1571-0661(04)80932-0

[76] G. Kahn, *The Semantics of a Simple Language for Parallel Programming*, Information Processing, pp. 471-475, North Holland, 1974.

[77] D. Karkinsky, S. Schneider, H. Treharne, *Combining Mobility with State*, IFM'07, LNCS vol. 4591, pp. 373-392, 2007. doi:10.1007/978-3-540-73210-5_20

[78] P. Knudsen, *Comparing two Distributed Computing Paradigms - A Performance Case Study*, Phd Thesis, Dpt. of Comp. Sci., Uni. of Tromso, Norway, 1995.

[79] C.P. Kunze, S. Zaplata, W. Lamersdorf, *Mobile Process Description and Execution*, DAIS'06, LNCS vol. 4025, pp. 32-47, 2006. doi:10.1007/11773887_3

[80] L. Lamport, F.B. Schneider, *The "Hoare Logic" of CSP, and All That*, TOPLAS'84, vol. 6, pp. 281-296, ACM, 1984. doi:10.1145/2993.357247

[81] D. May, H.L. Muller, *Using Channels for Multimedia Communication*, IPPS:SPDP'99, pp. 93-98, IEEE, 1999. doi:10.1109/IPPS.1999.760441

[82] D. May, H.L. Muller, *Copying, Moving and Borrowing Semantics*, CPA'01, vol. 59, pp. 15-26, IOS Press, 2001.

[83] M. Mazzara, A. Bhattacharyya, *On Modelling and Analysis of Dynamic Reconfiguration of Dependable Real-Time Systems*, DEPEND'10, pp. 173-181, 2010.

[84] A. McEwan, J. Woodcock, *Unifying Theories of Interrupts*, UTP'08, LNCS vol. 5713, pp. 122-141, 2010. doi:10.1007/978-3-642-14521-6_8

[85] R. Milner, *Pi-nets: A Graphical Form of pi-Calculus*, ESOP'94, LNCS vol. 788, pp. 26-42, 1994. doi:10.1007/3-540-57880-3_2

[86] R. Milner, *Communicating and Mobile Systems: the pi-calculus*, Cambridge University Press, 1999.

[87] L. Moreau, C. Queinnec, *Partial Continuations as the Difference of Continuations - A Duumvirate of Control Operators*, PLILP'94, LNCS, vol.844, pp. 182-197, 1994. doi:10.1007/3-540-58402-1_14

[88] H.L. Muller, D. May, *A Simple Protocol to Communicate Channels over Channels*, Euro-Par'98, LNCS vol. 1470, pp. 591-600, 1998. doi:10.1007/BFb0057905

[89] P. O'Hearn, J.C. Reynolds, H. Yang, *Local Reasoning about Programs that Alter Data Structures*, CSL'01, LNCS, vol. 2142, pp. 1-19, 2001. doi:10.1007/3-540-44802-0_1

[90] P. O'Hearn, H. Yang, J.C. Reynolds, *Separation and Information Hiding*, POPL'04, vol. 39, pp. 268-280, ACM, 2004. doi:10.1145/982962.964024

[91] P. O'Hearn, *Resources, Concurrency and Local Reasoning*, TCS'07, vol. 375, pp. 271-307, 2007. doi:10.1016/j.tcs.2006.12.035

[92] P. O'Hearn, *A Primer on Separation Logic (and Automatic Program Verification and Analysis)*, Software Safety and Security, vol. 33, pp. 286-318, IOS Press, 2012. doi:10.3233/978-1-61499-028-4-286

[93] E.R. Olderog, C.A.R. Hoare, *Specification-Oriented Semantics for Communicating Processes*, Acta Informatica, vol. 23, pp. 9-66, 1986. doi:10.1007/BF002680751986

[94] F. Orava, J. Parrow, *An Algebraic Verification of a Mobile Network*, FAC'96, vol. 6, pp. 497-543, 1992. doi:10.1007/BF01211473

[95] J. Parrow, *An Introduction to the Pi-calculus*, In *Handbook of Process Algebra*, Chapter 8, pp. 479-543. Elsevier, 2001.

[96] F. Peschanski, *On Linear Time and Congruence in Channel-Passing Calculi*, CPA'04, pp. 39-54, IOS Press, 2004.

[97] F. Peschanski, H. Klaudel, R. Devillers, *A Decidable Characterisation of a Graphical Pi-calculus with Iterators*, in Infinity, vol. 39, pp. 47-61, 2010.

[98] A. Philips, L. Cardelli, *A Graphical Representation for Biological Processes in the Stochastic pi-Calculus*, Trans. on Comput. Syst. Biol., LNBI vol. 4230, pp. 123-152, 2006. doi10.1007/11905455_7

[99] G.D. Plotkin, *A Structural Approach to Operational Semantics*, JLAP'04, vol. 60, pp. 17-139, 2004.

[100] A. Popescu, *Weak Bisimilarity Coalgebraically*, CALCO'09, LNCS vol. 5728, pp. 157-172, 2009. doi:10.1007/978-3-642-03741-2_12

[101] A. Popescu, *A Fully-Abstract Coalgebraic Semantics for the pi-calculus under Weak Bisimilarity*, Tech. Rep., Uni. of Illinois, USA, 2009.

[102] A. Rausch, *Towards a Formal Foundation for Dynamic Evolutionary Systems*, in Proceedings of the Workshop on Architecture-Centric Evolution (ACE 2005), the 19th European Conference on Object-Oriented Programming (ECOOP 2005), Jul 2005.

[103] J.C. Reynolds, *The Discoveries Of Continuations*, LISP and Symbolic Computation, vol. 6, pp. 233-247, 1993. doi:10.1007/BF01019459

[104] J.C. Reynolds, *Separation Logic: A Logic for Shared Mutable Data Structures*, LICS'02, pp. 55-74, IEEE, 2002. doi:10.1109/LICS.2002.1029817

[105] J.C. Reynolds, *An Overview of Separation Logic*, IFIP'05, LNCS vol. 4171, pp. 460-469, 2008. doi:10.1007/978-3-540-69149-5_49

[106] A.W. Roscoe, *The Theory and Practice of Concurrency*, Prentice-Hall, 1998.

[107] A.W. Roscoe, *On the Expressiveness of CSP*, 2011 draft, Available at cs.ox.ac.uk/ros11.pdf

[108] A.W. Roscoe, *CSP is Expressive Enough for pi*, in Reflections on the Work of C.A.R. Hoare, History of Computing 2010, pp. 371-404, 2010. doi:10.1007/978-1-84882-912-1_16

[109] A.W. Roscoe, *Understanding Concurrent Systems*, Prentice-Hall, 2010.

[110] A. Sampaio, J. Woodcock, A. Cavalcanti, *Refinement in Circus*, FME'02, LNCS vol. 2391, pp. 451-470, 2002. doi:10.1007/3-540-45614-7_26

[111] D. Sangiorgi, *π-calculus, Internal mobility and Agent-passing Calculi*, TCS'96, vol. 167, pp. 235-274, 1996. doi:10.1016/0304-3975(96)00075-8

[112] D. Sangiorgi, D. Walker, *The pi-calculus: A Theory of Mobile Processes*, Cambridge University Press, 2001.

[113] T. Santos, A. Cavalcanti, A. Sampaio, *Object-Orientation in the UTP*, UTP'06, LNCS vol. 4010, pp. 18-37, 2006. doi:10.1007/11768173_2

[114] M. Satyanarayanan, *Fundamental Challenges in Mobile Computing*, PODC'96, pp. 1-7, ACM, 1996. doi10.1145/248052.248053

[115] S. Schneider, *Concurrent and Real-Time Systems - The CSP Approach*, John Wiley & Sons, Ltd, 2000.

[116] S. Schneider, H. Treharne, B. Vajar, *Introducing mobility into CSP||B*, in Automated Verification of Critical Systems (AVoCS), 2007.

[117] G. Serugendo, M. Muhugusa, C.F. Tschudin, *A Survey of Theories for Mobile Agents*, World Wide Web, vol. 1, pp. 139-153, 1998. doi:10.1023/A:1019219916118

[118] A. Sherif, *A Framework for Specification and Validation of Real-Time Systems using Circus Actions*, PhD thesis, Center of Informatics - Federal University of Pernambuco, Brazil, 2006.

[119] L. Shi, Y. Zhao, Y. Liu, J. Sun, J.S. Dong, S. Qin, *A UTP Semantics for Communicating Processes with Shared Variables*, ICFEM'13, pp. 215-230, 2013. doi:10.1007/978-3-642-41202-8_15

[120] R. de Simone, *Higher-Level Synchronising Devices in MEIJE-SCCS*, TCS'85, vol. 37, pp. 245-267, 1985. doi:10.1016/0304-3975(85)90093-3

[121] G. Smith, *A formal framework for modelling and analysing mobile systems*, ACSC 2004, vol. 26, 2004.

[122] I. Stark, *A Fully-Abstract Domain Model for the pi-calculus*, LICS'96, pp. 36-42, IEEE, 1996. doi:10.1109/LICS.1996.561301

[123] K. Stolen, *Specification of Dynamic Reconfiguration in the Context of Input/Output Relations*, FMOODS'99, pp. 259-272, Springer, 1999. doi:10.1007/978-0-387-35562-7_20

[124] C. Strachey, C.P. Wadsworth, *Continuations: A Mathematical Semantics for Handling Full Jumps*, Higher-Order and Symbolic Computation, vol. 13, pp. 135-152, 2000. doi:10.1023/A:1010026413531

[125] X. Tang, J. Woodcock, *Travelling Processes*, MPC'04, LNCS vol. 3125, pp. 381-399, 2004. doi:10.1007/978-3-540-27764-4_20

[126] X. Tang, J. Woodcock, *Towards Mobile Processes in UTP*, SEFM'04, pp. 44-53, IEEE, 2004. doi:10.1109/SEFM.2004.10045

[127] B. Thomsen, *A Calculus of Higher Order Communicating Systems*, POPL'89, pp. 143-154, ACM, 1989. doi:10.1145/75277.75290

[128] E. Todoran, N.S. Papaspyrou, *Continuations for Parallel Logic Programming*, PPDP'00, pp. 257-267, ACM, 2000. doi:10.1145/351268.351297

[129] E. Todoran, *Metric Semantics for Synchronous and Asynchronous Communication: A Continuation-based Approach*, WDS'99, ENTCS'00, vol. 28, pp. 101-127, 2000. doi:10.1016/S1571-0661(05)80632-2

[130] E. Todoran, N.S. Papaspyrou, *Continuations for Prototyping Concurrent Languages*, Tech. Rep.t CSD-SWTR-1-06, National Technical Uni. of Athens, Softw. Eng. Lab., 2006.

[131] E. Todoran, *Mobile Objects and Modern Communication Abstractions: Design Issues and Denotational Semantics*, ISPDC'11, pp. 191-198, IEEE, 2011. doi:10.1109/ISPDC.2011.36

[132] B. Vajar, S. Schneider, H. Treharne, *Mobile CSP||B*, AVoCS'09, 2009. doi:10.14279/tuj.eceasst.23.338

[133] K. Wei, J. Woodcock, A. Burns, *Timed circus: Timed CSP with the miracle*, ICECCS'11, pp. 55-64, IEEE, 2011. doi:10.1109/ICECCS.2011.13

[134] K. Wei, *New Circus Time*, Tech. Rep., Dpt. of Comp. Sci., Uni. of York, UK, 2013.

[135] K. Wei, *Reactive Designs of Interrupts in Circus Time*, ICTAC'13, LNCS vol. 8049, pp. 373-390, 2013. doi:10.1007/978-3-642-39718-9_22

[136] P.H. Welch, F.R.M. Barnes, *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, 2004.

[137] P.H. Welch, F.R.M. Barnes, *Mobile Barriers for occam-pi: Semantics, Implementation and Application*, CPA'05, vol. 63, pp. 289-316, IOS Press, 2005.

[138] P.H. Welch, F.R.M. Barnes, *A CSP Model for Mobile Channels*, CPA:CSE'08, vol. 66, pp. 17-33, IOS Press, 2008. doi:10.3233/978-1-58603-907-3-17

[139] M.A. Wermelinger, *Specification of Software Architecture Reconfiguration*, PhD Thesis, Universidade Nova de Lisboa, Faculdade de Ciencias e Tecnologia, Departamento de Informatica, Lisboa, 1999.

[140] J. Woodcock, A. Cavalcanti, *The Semantics of Circus*, ZB'02, LNCS vol. 2272, pp. 184-203, 2002. doi:10.1007/3-540-45648-1_10

[141] J. Woodcock, A. Hughes, *Unifying Theories of Parallel Programming*, ICFEM'02, LNCS vol. 2495, pp. 24-37, 2002. doi:10.1007/3-540-36103-0_5

[142] J. Woodcock, A. Cavalcanti, *A Tutorial Introduction to Designs in Unifying Theories of Programming*, IFM'04, LNCS vol. 2999, pp. 40-66, 2004. doi:10.1007/978-3-540-24756-2_4

[143] J. Woodcock, A.J. Wellings, A. Cavalcanti, *Mobile CSP*, SBMF'15, LNCS vol. 9526, pp. 39-55, 2015. doi:10.1007/978-3-319-29473-5_3

[144] H. Yang, P. O'Hearn, *A Semantic Basis for Local Reasoning*, FOSSACS'02, LNCS, vol. 2303, pp. 402-416, 2002. doi:10.1007/3-540-45931-6_28