SPECIFICATION AND VERIFICATION ISSUES IN A PROCESS LANGUAGE

THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF NEWCASTLE UPON TYNE

By

Giuseppe Pappalardo

December 1995

PAGINATED
BLANK PAGES
ARE SCANNED AS
FOUND IN
ORIGINAL
THESIS

NO
INFORMATION
MISSING

To Enza

Abstract

While specification formalisms for reactive concurrent systems are now reasonably well-understood theoretically, they have not yet entered common, widespread design practice. This motivates the attempt made in this work to enhance the applicability of an important and popular formal framework: the CSP language, endowed with a failure-based denotational semantics and a logic for describing failures of processes.

The identification of behaviour with a set of failures is supported by a convincing intuitive reason: processes with different failures can be distinguished by easily realizable experiments. But, most importantly, many interesting systems can be described and studied in terms of their failures. The main technique employed for this purpose is a logic in which process expressions are required to satisfy an assertion with each failure of the behaviour they describe. The theory of complete partial orders, with its elegant treatment of recursion and fixpoint-based verification, can be applied to this framework. However, in spite of the advantages illustrated, the practical applicability of standard failure semantics is impaired by two weaknesses.

The first is its inability to describe many important systems, constructed by connecting modules that can exchange values of an infinite set across ports invisible to the environment. This must often be assumed for design and verification purposes (e.g. for the many protocols relying upon sequence numbers to cope with out-of-sequence received messages). Such a deficiency is due to the definition of the hiding operator in standard failure semantics. This thesis puts forward a solution based on an interesting technical result about infinite sets of sequences.

Another difficulty with standard failure semantics is its treatment of divergence,

the phenomenon in which some components of a system interact by performing an infinite, uninterrupted sequence of externally invisible actions. Within failure semantics, divergence cannot be abstracted from on the basis of the implicit fairness assumption that, if there is a choice leading out of divergence, it will eventually be made. This 'fair abstraction' is essential for the verification of many important systems, including communication protocols. The solution proposed in this thesis is an extended failure semantics which records refused traces, rather than just actions. Not only is this approach compatible with fair abstraction, but it also permits, like ordinary failure semantics, verification in a compositional calculus with fixpoint induction. Rather interestingly, these results can be obtained outside traditional fixpoint theory, which cannot be applied in this case. The theory developed is based on the novel notion of 'trace-based' process functions. These can be shown to possess a particular fixpoint that, unlike the least fixpoint of traditional treatments, is compatible with fair abstraction. Moreover, they form a large class, sufficient to give a compositional denotational semantics to a useful CSP-like process language.

Finally, a logic is proposed in which the properties of a process' extended failures can be expressed and analyzed; the methods developed are applied to the verification of two example communication protocols: a toy one and a large case study inspired by a real transport protocol.

Acknowledgements

I am very grateful to my supervisor, Santosh Shrivastava, for his continuing guidance and help. He has been exceptionally kind, patient and generous with me throughout these years of study.

I feel deeply indebted to Maciej Koutny, who read many parts of this work with great care and perception, providing precious and enlightening comments. Maciej has given me prompt and invaluable help and sympathy, exactly when I needed them.

Pippo Scollo introduced me to this beautiful and intriguing field. The example of his enthusiasm and rigour helped me to overcome many difficult moments.

The friendship of Teresa Cartellà, Antonella Di Stefano, Luigi Mancini, Geppino Pucci and Giuseppe Sarné has also helped me in many, emotional and material, ways throughout these years.

The cooperation and support of the Faculty of Engineering at Reggio Calabria, Italy, and particularly of its head, Vincenzo Coccorese, allowed me to pursue this work together with my academic duties.

Finally, I am grateful to my wife, parents and sister with her husband. The best thing I can say about them is that this work would have never been completed without their love and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 About This Chapter

Many authors have employed in noticeably different ways some key notions like *abstraction*, *specification*, *implementation*, *verification* and *correctness*, *behaviour* and *process*. Thus, before embarking on a work where these play a fundamental role, it seems appropriate to attempt a clarification both of concepts and terminology, without any pretence to novelty, but only for the sake of laying the groundwork for later study. This is the subject of the first sections, up to 1.5.

The contents and results of this thesis are then outlined in the concluding Section 1.6.

1.2 Abstraction in Specification

1.2.1 System Description

The description of a physical system, including a computer system, presupposes a twofold conceptual effort.

Firstly, the system must be perceived as separate from the environment surrounding it.[1] This entails identifying the *interface* of the system, i.e. the boundary

[1] In fact, this idea of separation would be better described by the term 'object', rather than 'system', which in modern natural languages is suggestive of a set of interacting parts. The

through which it interacts with the environment. Since interaction across an interface implies an exchange of information, it is also referred to as *communication.*

The other main intellectual activity involved in system description is *abstraction* —from detail viewed as irrelevant. Abstraction in system description takes two typical forms. The first consists in ignoring particular classes of system properties; these include, in the approach of this work, any property irrelevant as to what a system does (e.g., in most cases, colour or weight). The other form of abstraction is to omit describing how a system accomplishes what it does; this topic will be examined in detail in Section 1.2.2.

Thus, a system will be described in terms of what it does or, technically speaking, its *behaviour* (systems for which this is appropriate are called *reactive* after [Pnueli, 1986]). More precisely, the behaviour of a system is identified with the interaction that it is observed engage in with the environment, across the interface between them. However, this does not determine the exact nature of behaviour yet, but rather relates it to a hypothetical observer asked to describe it. In the following, this observer's choices or limitations may be borne in mind as a justification for the ways abstraction is further exercised in the description of behaviour. In this respect, four principal abstractions will be employed in this work:

1. indivisible (or atomic) actions;

2. abstraction from priority and probability in choice;

3. abstraction from time duration;

4. interleaving concurrency.

By the first abstraction, a description conceptually subdivides the interaction observed, and the associated information flow, into indivisible instances. Each instance causes the exchange of an amount of information that is represented by a value of a suitable set. These values are referred to as *actions.* Note that different interaction instances may result in the exchange of the same value, i.e. in the

---

fortune of 'system' in computer science stems from the typically structured nature of the objects this discipline deals with.

same action; in other words, an action may occur several times. Most interaction mechanisms relevant to computer technology (from buttons and wires to hardware registers and procedure calls) can be easily modelled by actions.

Descriptions that will be considered also abstract from behaviour aspects related to priority and probability in choices, and the measure of time. Thus, the duration of actions and intervals between them is ignored; only the order in which actions occur is described. This order must be total because descriptions are also assumed to eschew the representation of true concurrency: the simultaneous occurrence of actions is replaced by all their possible interleavings.

It now seems appropriate to give a brief assessment of the four abstractions introduced. While they are probably the most common and established in the literature (starting from [Dijkstra, 1965]), they seem to enjoy different degrees of consensus. Action atomicity can usually be assumed at some level, however low (be it digital electronics or quantum mechanics); it has also been shown that groups of actions may be rigorously viewed as a unique action at a higher abstraction level [Lamport, 1986]. For the class of applications considered in this work, priority and probability issues are usually abstracted from, but have received a growing attention recently [Vardi, 1985; Pnueli & Zuck, 1986; Baeten, Bergstra, & Klop, 1987; Larsen & Skou, 1989; Christoff, 1990; Jou & Smolka, 1990; Tofts, 1990; v. Glabbeek, Smolka, Steffen, & Tofts, 1990; Tofts, 1994]. Abstraction from time duration has been overcome by many researchers, in many different frameworks, in order to describe real-time systems; in particular, the works [Milner, 1983; Moller & Tofts, 1990; Gerth & Boucher, 1987; Reed & Roscoe, 1988; Groote, 1990; Baeten & Bergstra, 1991; Davies & Schneider, 1993] are worth citing here because they aim at extending non-timed methodologies similar to those of this work. Timed approaches also provide a way of representing concurrency without recourse to interleaving; another alternative (see e.g. [Best, 1985]) is to model concurrency as a *partial order* over actions (leaving truly concurrent actions out of the order relation); true concurrency can also be described by means of nets (see e.g. [Reisig, 1985]).

The four abstractions adopted certainly limit the properties that can be conveyed by the description of a system. On the other hand, they allow a description,

and the facts proved about it, to apply to a wider set of systems. Moreover, they afford a simpler formal modelling of systems, and an easier and rich theory. It seems therefore fair to conclude, generalizing a view of [Pnueli, 1986], that the best tradeoff among these contrasting issues should be suggested by the application or the area of interest. In general, however, the abstractions introduced above and adopted in this work have proved 'very successful' and 'popular', in the words of [Best, 1990].

1.2.2 Specification, Implementation and Abstraction

The development of a computer system depends on the ability to describe it. In some sense, system development may even be viewed as an activity that starts from a description of a system in terms of the customer's requirements, and ends with an executable description of it in a programming language (or perhaps as a digital circuit, depending on the target technology). Commonly, the initial, requirement description is referred to as a *specification*, and the final, executable one as an *implementation* of the system.

Since the implementation is in fact *the* system, it should be consistent with, or *satisfy*, the description represented by the specification. Thus the specification should not give more information about the system than the implementation does; in fact, it normally provides less information and is therefore more abstract a description than the implementation. A notion of satisfaction of a specification by an implementation should therefore conceal the additional detail present in the implementation, by mapping it onto suitable aspects of the more abstract specification.

Which abstraction level is desirable for a specification may not be thoroughly obvious for the application studied, even after taking into due account the class of properties that have to be formalized. Clearly, such considerations heavily influence the abstraction level adopted, but this will also ultimately depend on personal taste or inclination. The goal to be pursued, however, is to strike a balance between giving sufficient, but not excessive, information to specification users, and allowing developers among them enough leeway in choosing the most effective implementation solutions.

Related Concepts and Terminological Issues

In the literature, the basic concepts outlined above are expressed in many ways, which are worth recalling briefly. Consider a specification and an implementation that satisfies it. In this context, 'to implement' may be used as synonymous with 'to satisfy'. Moreover, the implementation is said to be *correct* with respect to the specification; establishing correctness is the goal of *verification.*[2] Finally, with reference to its richer amount of detail, the implementation is said to *refine* the specification.

Since the abstraction gap between realistic specifications and implementations may be rather noticeable, refinement must in practice be carried out stepwise. Typically, each step makes a distinct implementation choice, e.g. an algorithm, a representation of data or a decomposition into modules (each of which may then be refined separately). Thus, stepwise refinement gives rise to a series of decreasingly abstract system descriptions, which begin from the specification and culminate in the final implementation. In fact, any two adjacent descriptions in the series may be relatively viewed as a specification and an implementation respectively, according to the notion of satisfaction associated with the refinement step[3] relating them.

The latter observation, that any description may serve as a specification, has led some authors to use these terms as synonyms, perhaps adding a qualification to distinguish the intended use of a specification. E.g., [Sannella, 1988] calls 'high-level specification' a description intended to convey requirements, and 'executable specification' the final implementation. In [Pnueli, 1986], refinement starts from 'requirement specification' and continues through an an intermediate (architectural) 'system specification'.

In this work, any two descriptions such that one is a refinement of the other may be termed specification and implementation respectively. Two cases are typical though not exclusive, mutually or of others.

---

[2]In practice, correctness of an implementation with respect to a specification may also be ascertained by *testing.*

[3]This step may be devised out of intuition and then proved correct by verification, or selected with suitable criteria from a library of refinement patterns known to be correct in advance. The latter approach, first advocated in [Burstall & Darlington, 1977], will not be pursued in this work.

1. *Tightening* refinement: the specification loosely describes the behaviour of a system, and the implementation removes (part of) this looseness. E.g. while the specification of a channel may require it to output data in the order in which they are input, the implementation may also insist that any input should be immediately followed by an output.

2. *Decomposition* refinement: the implementation describes a set of systems, and a way of combining them into one whose behaviour satisfies the specification.

Finally, a terminological remark seems appropriate: the terms 'specification' and 'implementation' denote not only instances of a description, but also the activity of producing these instances. The converse can be said about the word 'refinement'.

1.3 Formalisms for Reactive Systems

There is substantial agreement in the computer science community that precise descriptions and trusted reasoning must be formal, i.e. based on the methods of mathematics and logic. However, it has been the subject of much controversy whether the higher level of confidence attained by formal methods is worth the complexity they bring about in the specification and verification stages of system development. This work clearly presupposes an affirmative answer, but will not try to put the case for it; the interested reader is referred e.g. to [Meyer, 1985] and [Hall, 1990], which also highlights the role formal methods may have in other development phases (like implementation and testing). A thorough survey on the role of formal methods in system development can be found in [Wing, 1990], which is also a rich source of further reference.

A formal approach to specification and verification requires that mathematical concepts should be employed to represent implementations, specifications and the satisfaction relationship between them. It should be expected that accomplishing this will involve recourse to the two main forms of abstraction identified earlier, in Sections 1.2.1 and 1.2.2 respectively: (1) only behavioural properties (subject

to the specific abstractions assumed in Section 1.2.1) will be formalized, and (2) formalization should reflect the abstraction gap between a specification and an implementation satisfying it. These topics are examined in the following Sections 1.4 and 1.5.

1.4 Formalizing Implementations

1.4.1 Processes and Process Expressions

It would seem natural to begin by seeking a set *Proc*, whose elements, referred to as *processes*, are suited to representing implementations. A process should therefore provide, at least, a suitable formalization of system behaviour.

On the other hand, an implementation is intended to describe not just the behaviour of a system, but also how that behaviour can be obtained. A member of a set without sufficient structure does not appear to be suitable for capturing this notion. Instead, there would appear to be a need for a formal (process) language PL, with constants, denoting elementary behaviour patterns, and operators, used to construct expressions denoting complex behaviour. Of course, different expressions may happen to denote the same behaviour, reflecting the intuition that a behaviour may be implemented in different ways.

In the following, the term 'process' will be reserved for mathematical entities providing a precise formalization of the intuitive idea of behaviour; by this it is meant that (mathematically) different processes should denote (intuitively) different behaviours. Thus, processes cannot be the *process expressions* of the process language PL, nor can the set *Proc* coincide with PL. Rather, process expressions should be given an interpretation as processes, in order to define which behaviour they denote. The dichotomy between processes and process expressions is an instance of the classical one between syntax and semantics or, more philosophically, form and function.

There are many approaches to the problem of selecting a suitable *Proc* set and relating it to PL. In all of them, however, actions from a set *Act* can be expected to play a role in that, as assumed in Section 1.2.1, they represent the smallest

instances in which behaviour may take place. The two approaches employed in this work are described in Sections 1.4.2 and 1.4.3.

1.4.2 Transitional Process Semantics

Labelled Transition Systems

Transitional process semantics rests on the idea that any behaviour transforms into another behaviour after performing an action, which can be either *observable*, i.e. a member of Act, or *unobservable*, denoted by the symbol ι and also called *internal* or *silent*. The set $Act \cup \{\iota\}$ of all actions (observable or not) will be denoted by Act_ι.

Thus, formally, any behaviour is to be modelled by an element π of a suitable set $Stat$ endowed with a relation $\longrightarrow \subseteq Stat \times Act_\iota \times Stat$. That $(\pi, \alpha, \pi') \in \longrightarrow$ or, more suggestively, $\pi \xrightarrow{\alpha} \pi'$ is understood to mean that the behaviour represented by π may perform action α and transform into the behaviour represented by π'; sometimes, π' is said to be an *α-derivative* of π. A similar terminology applies to the other transition relations introduced later.

The structure $(Stat, Act_\iota, \longrightarrow)$ is called a *labelled transition system* (LTS) over $Stat$, the set of *states*, and Act_ι, the set of *labels*; a triple (π, α, π') in $\longrightarrow$ is called a *(α)-transition* from state π to the *successor* state π'. Note that the exact nature of states is relevant for behaviour representation only up to isomorphism; for it is clear that if a behaviour is representable within a LTS $(Stat, Act_\iota, \longrightarrow)$, so it is within $(Stat', Act_\iota, \longrightarrow')$ provided $Stat$ and $Stat'$ are isomorphic with respect to $\longrightarrow$ and $\longrightarrow'$.

It will be postulated that states of a LTS are adequate for representing behaviour, in the sense that two systems that behave as though they contained the same LTS in the same state are indistinguishable.

However, LTS states cannot be adopted as processes in the sense introduced earlier, because the converse is not true: different states may well represent the same behaviour. An easy example of this is provided by infinitely many distinct states π, $\pi_0, \pi_1, \ldots$ having only the transitions:

$$\pi \xrightarrow{a} \pi \qquad\qquad \pi_0 \xrightarrow{a} \pi_1 \xrightarrow{a} \ldots$$

A partial solution, and a step towards the discovery of a set *Proc*, requires a sharper insight into the nature of behaviour. It will now be postulated that a behaviour is completely determined by the order in which its actions occur, and its *branching structure*, i.e. the choices it makes among the actions available to it at a certain stage. This information can be easily recovered from a state of a LTS: it suffices to 'unwind' transitions from it and its successors into a rooted, unordered *derivation* tree, with edges that are labelled by actions, and nodes that are anonymous (not labelled by states). Thus, two states π, σ generating the same derivation tree—in symbols $\pi \hat{\sim} \sigma$—will be assumed to denote the same behaviour. This also suggests that (for a fixed Act_ι) a sufficient condition for a LTS to describe every possible behaviour is that its states generate every possible derivation tree.[4] For the rest of Section 1.4.2, a fixed LTS $(Stat, Act_\iota, \longrightarrow)$ satisfying the previous condition will be assumed.

Bisimulation Equivalence

Even $Stat/\hat{\sim}$, the set obtained partitioning *Stat* with the equivalence relation $\hat{\sim}$, is unsuitable as a candidate for *Proc*. Intuitive arguments whereby different derivation trees may represent the same behaviour, originally due to Robin Milner, can be found in his book [Milner, 1989]. Hence, we have still to pursue the goal of finding an equivalence relation $\overset{\circ}{\sim}$ that equates LTS states iff they describe behaviours indistinguishable to observation or experimentation. This equivalence will be assumed as *the* observation equivalence, and the set *Proc* will then be identified with $Stat/\overset{\circ}{\sim}$. As noted previously, this amounts to tying the exact, formal nature of behaviour to the abilities intuitively attributable to the observer that describes it.

The solution of [Milner, 1989] is to define three, increasingly coarse, equivalence relations over *Stat*.

The characteristic property of the strongest one, $\sim$, is that LTS states π, σ satisfying $\pi \sim \sigma$ *bisimulate* each other, i.e. belong to a *(strong) bisimulation* relation $\mathcal{B}$ such that:

[4] Thus, the cardinality of *Stat* for such a 'most general' LTS should be at least that of the set of derivation trees.

(1) whenever $\pi \xrightarrow{\alpha} \pi'$, for any $\alpha \in Act_\iota$, then also $\sigma \xrightarrow{\alpha} \sigma'$ for some σ', and π' and σ' are in $\mathcal{B}$;

(2) $\mathcal{B}$ is symmetric.

Strong bisimulation equivalence $\sim$ is defined to be the largest $\mathcal{B}$ enjoying properties (1) and (2). It gets closer to the sought $\overset{\circ}{\sim}$ but cannot yet be it. For, while states that bisimulate each other do not appear to be distinguishable by any plausible experiment, state pairs may be exhibited that are not in $\sim$ and still could not possibly be distinguished. The reason lies in the observers' inability to perceive every possible occurrence of the internal action ι.

In fact, any activity observed coming from a system may be interspersed with arbitrarily many internal actions. To take this into account, define:[5]

$$\sigma \overset{\langle\rangle}{\Longrightarrow} \sigma' \quad \text{iff} \quad \text{either } \sigma' = \sigma \qquad (1\text{-}1)$$
$$\text{or } \sigma \xrightarrow{\iota} \sigma_1 \ldots \xrightarrow{\iota} \sigma_n \xrightarrow{\iota} \sigma' \text{ for some } \sigma_1, \ldots, \sigma_n \ (n \geqslant 0)$$

and, for $a \in Act$:

$$\sigma \overset{a}{\Longrightarrow} \sigma' \quad \text{iff} \quad \sigma \overset{\langle\rangle}{\Longrightarrow} \sigma_1 \xrightarrow{a} \sigma_2 \overset{\langle\rangle}{\Longrightarrow} \sigma' \text{ for some } \sigma_1, \sigma_2. \qquad (1\text{-}2)$$

Then, after letting $\hat{a} = a$ for $a \in Act$, $\hat{\iota} = \langle\rangle$, we may replace $\xrightarrow{\alpha}$ by $\overset{\hat{\alpha}}{\Longrightarrow}$ in (1) above in order to obtain the definitions of *weak bisimulation* and, accordingly, *weak bisimulation equivalence* $\approx$.

A slight problem with $\approx$ is that $\pi \approx \sigma$ does not guarantee that the behaviours denoted by π and σ can replace each other within a larger behaviour, leaving this one unchanged. The remedy is to modify the definition of $\pi \approx \sigma$ slightly, so that initial ι actions of π must be matched by initial ι actions of σ and viceversa. The resulting equivalence relation is therefore finer (identifies less) than $\approx$; as it enjoys substitutivity, it is called *weak bisimulation congruence* ($\sim$ is also a congruence and is thus also called *strong bisimulation congruence*). In the following, the qualifiers 'strong' or 'weak' and 'bisimulation' will be omitted when they are clearly supplied by the context.

[5] $\langle\rangle$ denotes the *empty* sequence.

Other Observation Equivalences

The observer associated with bisimulation equivalence is quite a powerful one: it must be capable of *global testing* [Abramsky, 1987] of all the different options available to a behaviour at any stage. Normally, no observer is thought to be more powerful than this one, which amounts to admitting that the sought ideal observation equivalence $\overset{\circ}{\sim}$ should not be finer than bisimulation equivalence. The only notable exception is represented by *branching bisimulation* equivalence [v. Glabbeek & Weijland, 1989], which is a slight variation on bisimulation anyway.

On the other hand, many authors have proposed notions of behaviour and observation equivalences corresponding to less powerful (discriminating) observers. These equivalences are therefore coarser than bisimulation equivalence.

Surveys and comparisons of observation equivalences can be found in [De Nicola, 1987] and [v. Glabbeek, 1990; v. Glabbeek, 1993], where each of the best known ones is explained in terms of a specific kind of system interface and a class of experiments that can be performed on it. The equivalences considered in [v. Glabbeek, 1990; v. Glabbeek, 1993], if compared in terms of how many identifications they induce, form a lattice with bisimulation equivalence at the top and, among the minimal ones, *failure equivalence*, which is related to the approach of [Brookes, Hoare, & Roscoe, 1984]. Formally, states π and σ are failure equivalent if they possess the same failure sets, as defined later, in Section 1.4.3.

Transitional Observation Equivalence-Based Semantics for a Process Language

As discussed earlier, in the transitional approach processes may be thought of as equivalence classes of an equivalence relation, e.g. $\approx$, over the states of a LTS. Therefore, a technique for interpreting process expressions of the language PL as processes is to define a relation $\longrightarrow \subseteq \mathrm{PL} \times Act_\iota \times \mathrm{PL}$, hence a LTS $(\mathrm{PL}, Act_\iota, \longrightarrow)$ in which states are process expressions. Then, $\approx$ may be applied to process expressions as well, and a process expression $p \in \mathrm{PL}$ is interpreted with, or given as *meaning*, the set

$$\{q \in \mathrm{PL} \mid p \approx q\}$$

which is the equivalence class of p, i.e. a process.

As the language PL will be inductively defined, the definition of the relation $\longrightarrow \subseteq \mathrm{PL} \times Act_\iota \times \mathrm{PL}$ can be expected to reflect the structure of process expressions in PL. This will be obtained, following an approach introduced in [Plotkin, 1982] and generalized in [Baird, Istrail, & Meyer, 1988], by a set of inference rules; in the typical case, a transition of process expression $op(p_1, \ldots, p_N)$, where op is an operator of PL, is inferred from transitions of $p_1, \ldots, p_n$.

1.4.3 Observation Set Process Semantics

In this approach to process semantics, the set *Proc* is given explicitly; processes are sets of *observations*, which in turn are elements of a set *Obs*. The intention is that observations should represent the outcomes of experiments that an observer may conduct on a system; the system is then identified with the set of all possible outcomes (of all experiments). Typically, the observations that compose a process are made up of actions, and provide at least enough information to recover the *traces*, i.e. the sequences of actions that can be performed by the system described by the process. Below let $\langle a_1, \ldots, a_n \rangle$ denote the sequence of actions $a_1, \ldots, a_n$ ($\langle\rangle$ is the empty sequence) and st the concatenation of sequences s and t.

Not every observation set can be a process; intuitively, a process should at least contain the 'empty observation' and be closed with respect to shorter observations: if observation x can be made only after a shorter one x', then a process containing x should also contain x'. E.g. if observations are simply taken to be traces, then processes must be non-empty, prefix-closed trace sets.

Many observation set approaches have been proposed. The work [v. Glabbeek, 1990] surveys the most interesting ones, and devises ingenious experiments capable of detecting the kind of observations underlying each approach. An important example is represented by the failure model of CSP [Brookes, Hoare, & Roscoe, 1984]. In it, an observation is a failure, i.e. a pair (s, X) where $s \in Act$ and $X \in \mathsf{p}Act$ (the set of finite subsets of Act); this is intended to express that a system can be observed perform the trace s and then refuse all the actions in the finite *refusal set* X. A process is a failure set satisfying the constraints defined below.

Definition 1.1 According to [Brookes, Hoare, & Roscoe, 1984], a *process* is a set $P \subseteq Act^* \times \mathsf{p}Act$ satisfying the following constraints:

1. non-emptiness: $(\langle\rangle, \varnothing) \in P$;
2. trace prefix-closure: if $(st, \varnothing) \in P$, then $(s, \varnothing) \in P$;
3. refusal subset-closure: if $(s, X) \in P$ and $Y \subseteq X$, then $(s, Y) \in P$;
4. trace-refusal consistency: if $(s, X) \in P$ and $(s, X \cup \{a\}) \notin P$, then $(s\langle a\rangle, \varnothing) \in P$. □

The previous constraints are easy to justify appealing to intuition. More formally, assume (as in Section 1.4.2) that a LTS $(Stat, Act_\iota, \longrightarrow)$ provides enough information about every behaviour; complete the definitions (1-1) and (1-2) of the $\Longrightarrow$ relation by letting, for $s \in Act^*$, $s = \langle a_1, a_2, \ldots, a_n\rangle$ $(n > 0)$:

$$\pi \stackrel{s}{\Longrightarrow} \pi' \text{ iff } \pi \stackrel{a_1}{\Longrightarrow} \pi_1 \stackrel{a_2}{\Longrightarrow} \ldots \pi_{n-1} \stackrel{a_n}{\Longrightarrow} \pi' \text{ for some } \pi_1, \pi_2, \ldots, \pi_{n-1}.$$
$$\pi \stackrel{s}{\Longrightarrow} \text{ iff } \pi \stackrel{s}{\Longrightarrow} \pi' \text{ for some } \pi'.$$

Note that the LTS $(Stat, Act^*, \Longrightarrow)$ also captures enough information about observable behaviour. In accordance with the intuitive meaning of failures, it is possible to define the failure set of a state π as in [Brookes, Hoare, & Roscoe, 1984]:

$$failures(\pi) = \{(s, X) \in Act^* \times \mathsf{p}Act \mid \exists \pi' : \pi \stackrel{s}{\Longrightarrow} \pi' \text{ and } \forall x \in X : \pi' \not\stackrel{x}{\Longrightarrow}\} \quad (1\text{-}3)$$

and it is immediate to verify that it satisfies the constraints of Definition 1.1. Conversely, given a failure set P satisfying those constraints, it is possible to construct a LTS with a state π, i.e. a behaviour, such that $failures(\pi) = P$ [Brookes, Hoare, & Roscoe, 1984].

Observation Set Semantics for a Process Language

An observation set semantics for a process language PL is defined by a *meaning* function, mapping every process expression p of PL onto an observation set, called the *meaning* or *denotation* of p. Observation set semantics is therefore of the *denotational* kind.

In Section 1.4.2 we discussed how a process expression p can be viewed as a state of a LTS. As such, p can be easily ascribed an observation set $obsset(p)$, exploiting the LTS transition relations $\longrightarrow\!\!\triangleright$ or $\Longrightarrow\!\!\triangleright$, as done e.g. for failures in equation (1-3). This allows $obsset(\,)$ to be taken as a meaning function for a denotational semantics over PL.

In fact, it is easy to give an equivalent transitional style semantics in which process expressions are instead interpreted as equivalence classes. For, if two states, or process expressions, p and q are defined to be equivalent whenever $obsset(p) = obsset(q)$, then $obsset(p)$ characterizes the equivalence class (meaning) of p with respect to this new equivalence relation.

Adopting observation sets as processes can also support a compositional denotational semantics for process expressions. In this case, the meaning function $\mathcal{O}[\![\,]\!]$ should map a complex process expression p of the form $op(p_1, \ldots, p_n)$ onto an observation set $\mathcal{O}[\![p]\!]$, determined by the denotations $\mathcal{O}[\![p_1]\!], \ldots, \mathcal{O}[\![p_n]\!]$ of the operands, in a way that reflects the intended meaning of the operator op. Moreover, it is often possible to compare observation sets by set inclusion or in related ways that give *Proc* the status of a *complete partial order*, and to interpret language operators as continuous functions over *Proc*. In this setting, powerful techniques are available for introducing recursively defined process expressions and reasoning about them (see e.g. [Loeckx & Sieber, 1987]). This has been accomplished e.g. in [Brookes, Hoare, & Roscoe, 1984] for the CSP language with the failure semantics.

A more general and formal treatment of what has been described here as observation set semantics is given in [Olderog & Hoare, 1986].

Relating Transitional, Equivalence-Based and Observation Set Semantics

Figure 1.1 depicts the relationship between the above-mentioned approaches to semantics. For this relationship to be consistent, a process expression p should be mapped onto the same observation set by both the techniques introduced for this purpose: the compositional meaning function $\mathcal{O}[\![\,]\!]$ and the $obsset(\,)$ function applied to p viewed as a state of a LTS. Thus, it is required:

$$\text{for all } p \text{ in PL: } \mathcal{O}[\![p]\!] = obsset(p) \tag{1-4}$$

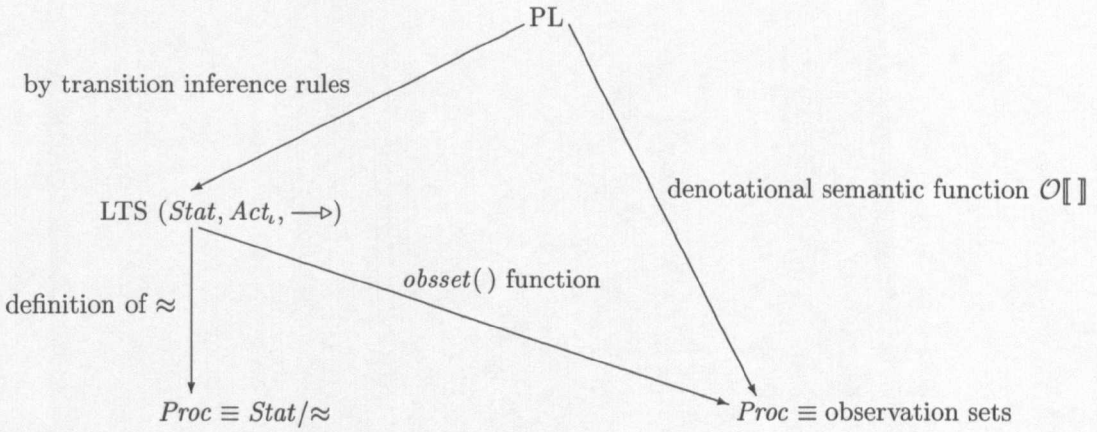


Figure 1.1: Relation between process semantics

Since bisimulation equivalence $\approx$ is about the strongest meaningful observation equivalence available, other consistency requirements could be that:

$$\text{for all } p, q \text{ in PL: } p \approx q \text{ implies } obsset(p) = obsset(q) \qquad (1\text{-}5)$$
$$\text{for all } p, q \text{ in PL: } p \approx q \text{ implies } \mathcal{O}[\![p]\!] = \mathcal{O}[\![q]\!]$$

Of course, a sufficient condition for the latter property to hold is that the previous two (1-4) and (1-5) hold. E.g. if observations are failures, defined as in equation (1-3), it is well-known (see e.g. [De Nicola, 1987]) that (1-5) holds (however the converse does not); (1-4) is stated in [Brookes, 1983a] (for a slightly different notion of failures).

1.4.4 Algebraic Process Semantics

There is undoubtedly a degree of arbitrariness in building process semantics upon a particular notion of equivalence or a particular sort of observations. This seems to be confirmed by the wealth of process semantics proposed. The remedy adopted in ACP (Algebra of Communicating Processes [Bergstra & Klop, 1984; Baeten & Weijland, 1990]) is to found the semantics on a set of axioms that represent equations between process expressions. These describe equalities that actual processes and process operations are expected to satisfy, but do not prescribe a particular

choice for them: any choice which is is consistent with (or, technically, is a *model* of) the equations will do.

In this way, however, it is the choice of equations and process language operators that becomes critical: somehow, it is as though the arbitrariness has been moved from the semantic to the syntactic level. This may be viewed as advantageous as far as the choice of equations is concerned: it is instructive to realize how including or excluding some equations, which amounts to accepting or refusing certain identifications between processes, corresponds to accepting or refusing models based on particular observation equivalences or observation sets. An example of this is given in Section 2.6.2. However, the generality of the ACP approach is also tied to the process operators chosen, and no such set is universally adopted by all authors.[6] Moreover, other algebraic process semantics could replace equations with more complex logical formulae, and make an even stronger claim to being general.

A process semantics independent even of the operators and the logic in which axioms are expressed can be given by resorting to *institutions* [Goguen & Burstall, 1984].

The algebraic approach to semantics will not be pursued in this work.

1.5 Formalizing Specifications and Satisfaction

A rigorous general description of specification formalisms can be found e.g. in [Larsen, 1990]. The discussion here is informal and geared to the view introduced in Section 1.2.2, that any two behaviour descriptions of which the first is more detailed, but consistent with (*satisfies*) the second, represent an implementation and a specification respectively.

In an attempt to organize the numerous formal specification techniques proposed, most authors distinguish between those rendered by logical formulae and those rendered by formal devices (like LTSs, automata etc.). In our view, it is

[6] In contrast, when the algebraic approach is applied to the functional description of a particular software system (rather than the behavioural description of the set of all systems), the operators are either legitimately selected by the system designer or universally accepted, e.g. `push()` and `pop()` for a stack.

perhaps more appropriate to classify a specification, primarily, according as the behaviour it *aims* to describe is a single one or any member of a set. This alternative, indeed, is often more a matter of aim or convention (between specifiers and users), than form of the specification (like its being a LTS or a logical formula). Thus, e.g., the informal specification 'data input are output in the same order' may be taken to describe any behaviour that fulfils it (including one that does nothing), or the behaviour that fulfils it precisely, by performing any permitted action at any stage.

It is often the case that a specification approach may be indifferently understood either way, simply by choosing the right semantic level. Four possible approaches to formal specification will now be examined in the light of this and the other considerations above.

First Approach

In this approach, a process expression p is regarded as a specification of a single[7] behaviour—that denoted by p in the chosen semantics. E.g. within the observation equivalence semantic framework, p can be taken to specify its $\approx$-equivalence class.

Implementations are also formalized by process expressions: a process expression q satisfies p if it denotes the same behaviour, (i.e. in the example cited if $p \approx q$). The added detail provided by q (if any) is its structure as a process expression (of course, also p has a structure, which is however irrelevant to its use as a specification).

Specifications of this kind are therefore rather demanding: no leeway is allowed for the behaviour of implementations.

Second Approach

The first framework may be modified by stipulating that q satisfies p if the behaviour denoted by q approximates that denoted by p. The formalization is often

[7]Even in this case, a specification p may be regarded as describing any member of a set, i.e. $\{q \mid q$ denotes the same behaviour as $p\}$, but this is a set of distinct process expressions, not behaviours.

based on a partial order $\leqslant$ over states of a LTS (e.g. simulation [Larsen, 1987]); satisfaction of p by q is then rendered as $q \leqslant p$.

An obviously equivalent alternative presentation is to view the specification p as describing any behaviour that approximates that denoted by p; satisfaction is then viewed as set membership: $q \in \{p' \mid p' \leqslant p\}$.

In any case, the information that the implementation q adds to that specified by p is not only structure, as in the first approach, but also knowledge of the particular behaviour approximating that denoted by p.

Third Approach

This case occurs when a specification is formalized with a formula S of a logic interpreted over a set U whose elements denote behaviour (in one of the ways described in Section 1.4). S is assumed to describe any of the behaviours denoted by an element of U upon which S is true. Implementations may be taken to be either, directly, elements of U or, indirectly, process expressions equipped with a mapping onto U (generally through a LTS). An implementation I satisfies the specification S if S is true for the element of U associated with I. The additional information I provides is: (i) which behaviour it denotes among those allowed by S, and (ii), if I is a process expression, how this behaviour can be obtained.

We have mentioned that either process expressions or more direct denotations of behaviour (e.g. states of a LTS) may serve as implementations.[8] However, it is clear that interesting implementations must ultimately provide detailed information on how they are to be constructed, which can be ensured only by recourse to process expressions.

Finally, let us consider two examples of the third approach. (i) Satisfaction of a branching temporal logic formula may be defined over a state or a state path, which may be thought of as generated by a fixed LTS (of a restricted kind) [Emerson & Halpern, 1986]. (ii) The interpretation of Hennessy-Milner logic [Hennessy & Milner, 1985] is given over a state of a LTS.

[8] In fact this applies whenever the specification does not already identify a unique behaviour (as it does in the first approach above).

Fourth Approach

The last case occurs when behaviour is understood as an observation set, as discussed in Section 1.4.3. Then, a specification is a logical formula S interpreted over an observation; S is aimed at describing any behaviour which makes S true with each observations that can be made on it. An implementation I may be an observation set or (with the usual difference of added detail) a process expression mapped onto an observation set by some semantics. I satisfies S if every observation in the set associated with I makes S true.

In general, logical formalisms of the fourth kind can be expected to be less expressive than those of the third. In practice, however, they have proved to suffice for many application areas, and in many different formal settings (Lamport [Lamport, 1980] has even suggested the view (criticized in [Emerson & Halpern, 1986]) that they are preferable for reasoning about concurrent systems). This can be seen as a consequence of the observation set philosophy: once behaviour has been identified with a set of experiment outcomes, it becomes natural to express requirements on a system as properties that must be confirmed by every experiment.

An important advantage of formalisms of the fourth kind is that specifications typically take a conjunctive form, and each conjunct can be verified separately and to a large extent independently of the others. E.g. in a CSP-based **sat** formalism like that employed in this work, the analyses of input-output relation and deadlock-freedom (each expressed with a different set of conjuncts) exhibit a high degree of independence (or 'orthogonality').

In this work only the fourth approach to formalization will be employed. As discussed in [Pnueli, 1986], this framework is well-suited to nearly all design phases, ranging from early requirement specification, up to detailed system specification, which immediately precedes actual coding.

1.6 Objectives and Contents of This Work

While specification formalisms of the previously described kinds are now sufficiently well-understood theoretically, they have not yet succeeded to enter common, widespread design practice. This may be partly due to a failure to realize

their cost-effectiveness, and to an obvious resistance by designers who have not been trained to their use. However, our past experience with formal specification and verification of OSI systems [Carchiolo, Di Stefano, Faro, Pappalardo, & Scollo, 1986; Carchiolo, Di Stefano, Faro, & Pappalardo, 1989] has convinced us that there are also margins to try to improve the applicability of current formal techniques. For this attempt, the framework selected in this work is a process language similar to CSP, with a failure-based denotational semantics and a logic for describing failures of processes. Two crucial areas where the applicability of this formalism might be enhanced are:

1. the description of networks of systems that can exchange data from an infinite set;

2. *fair abstraction* from loops of internal actions, and a calculus for reasoning about the properties of processes affected by such loops;

These issues will be clarified and analyzed in Section 1.6.2 below; the solutions proposed will then be expounded in Section 2.6.1 and Chapter 3, which proposes a new, *extended failure* semantics. Finally, Chapter 4 is devoted to the development of a logic in which properties of the extended failures of a process can be expressed and verified; these ideas will be applied to the verification of two example communication protocols: a toy one and a large case study inspired by a real transport protocol. But let us try first to provide a motivation for preferring the chosen formal framework.

1.6.1 The Formalism Employed: A Motivation

In the formalism considered in this work, processes are identified with failure sets (of a suitable kind). This identification is supported by convincing reasons, ranging from the theoretical[9] to the intuitive: processes that are equivalent in some meaningful sense cannot possess different failures, or they would easily be distinguished

[9] In an appropriate setting, failure equivalence is the largest congruence that guarantees equality of maximal traces [Bergstra, Klop, & Olderog, 1988]. Moreover, if observers are assumed to be a sort of processes, then failure equivalence amounts to indistinguishability by these observers [De Nicola & Hennessy, 1984].

by a suitable observer. But the most important justification is, as usual, pragmatic: the work of Hoare [Hoare, 1985] and other researchers has clearly shown that many interesting systems can be described and studied in terms of their failures. The main technique for this purpose is a formalism of the fourth kind considered in Section 1.5; it is based on a "**sat** logic" in which process expressions may be required to satisfy an assertion with every failure of the behaviour they denote. Of course, setting up the **sat** logic requires just that every process expression is ascribed a set of failures, e.g. through a LTS, and not necessarily a denotational failure semantics. The advantage of the latter approach is that failure sets form a *complete partial order* (cpo), and process language operators can be defined through continuous functions over this cpo. This facilitates a smooth treatment of recursively defined process expressions, which can be interpreted as least fixpoints of continuous functions, and reasoned about by introducing powerful fixpoint induction rules in a calculus for the verification of **sat** properties.

1.6.2 Problems with Failure Semantics

Although standard failure semantics affords the advantages illustrated above, its practical applicability is impaired by two weaknesses that will be examined in the remainder of this section.

Hiding Infinite Action Sets

Many important systems are constructed by connecting modules that can exchange values of an infinite set across ports invisible to the environment. While values actually exchanged during real operation will in practice belong to a finite set, for design and verification purposes it is often necessary or more convenient to think that these values are drawn from an infinite set.

As an illustration, consider the many protocols relying upon sequence numbers to cope with messages received out of sequence. These protocols are usually designed and proved correct by assuming that infinitely many sequence numbers are available. In practice, sequence numbers are implemented as N-bit integers, but this only works if suitable bounds are assumed on message lifetime. However,

specifying this assumption in order to verify that it guarantees correctness requires the ability to describe quantitative time, which is eschewed in this work and brings about considerable complexity anyway (see [Shankar & Lam, 1987] for an example).

Unfortunately, the potential exchange of infinitely many values at an internal port cannot be satisfactorily described in standard failure semantics. This deficiency is due to the definition of the *hiding* operator. This problem and a possible solution will be analyzed in Section 2.6.1.

Divergence and Fair Abstraction

Intuitively, *divergence*, also called *livelock*, is a phenomenon in which some components of a system interact by performing an infinite, uninterrupted sequence of externally invisible actions. Thus, divergence *may* arise when system components reach a stage after which they persistently have an option to interact with each other, although options to interact with the environment may also be available. Divergence *does* in fact arise when the former option is always taken; it is instead averted by *fairness*, the requirement that, if the latter options are infinitely often available, they should infinitely often be chosen (in the words of [Apt, Francez, & Katz, 1988]).[10]

How these ideas should be reflected by a formal technique is a complex issue, with a strongly philosophical flavour. Here a pragmatic approach is taken:

Req A formalism will just be required to allow a potentially diverging system to be proved correct, if it intuitively appears to be so under some informal fairness assumption.

Temporal logic clearly satisfies this criterion in a direct way, in that it can either express fairness or be interpreted over a fair computational model (as in [Clarke, Emerson, & Sistla, 1986]). This of course stems from its interpretation being based on infinite sequences of observations (computation states) [Gabbay, Pnueli, Shelah, & Stavi, 1980; Lehmann, Pnueli, & Stavi, 1981; Lamport, 1980; Emerson &

[10]This is actually a rather specific form of fairness: a thorough analysis of its many forms is given in [Apt, Francez, & Katz, 1988].

Halpern, 1986]. There is however a price to be paid for this (unless a real numbers interpretation is adopted [Barringer, Kuiper, & Pnueli, 1986]): either finite or infinite stuttering (dummy state repetition) has to be treated unsatisfactorily, which would seem to indicate that divergence-related phenomena have a rather fundamental complexity.

On the other hand, to quote [Brookes, Hoare, & Roscoe, 1984], 'it seems impossible to define a notion of fairness such that a fair process can be distinguished from an unfair one by any finite observation'. This implies that in semantics based on failure or bisimulation equivalences (and intermediate ones) fairness cannot be specified (but see [Parrow & Gustavson, 1984] for an infinite observation equivalence). Likewise, knowledge of all the finite observations on a system does not reveal whether it would actually diverge if part of its actions were regarded as internal; in general, this knowledge can only suggest the *possibility* of divergence.

As an illustration, consider two systems P and Q that can only perform infinite sequences of actions from $\{a, b\}$ and never stop: P performs all such sequences, while Q performs only those where a occurs infinitely often. Then, if b is made internal, P will diverge, while Q will not. However, P and Q have exactly the same set $\{a, b\}^*$ of finite sequences: all that can be said from this set is that both P and Q could diverge.

It would therefore seem that, in general, all that finite observation semantics can do, concerning divergence, is either record its possibility or abstract from it. The latter alternative is referred to as *fair abstraction*, for it presupposes the assumption that, in practice, fairness will never allow potential divergence to become actual (as though systems like P in the example above were simply not realizable). It should be noted that in this way fairness, while beyond the discriminating power of semantics, finds nevertheless a formal expression. With fair abstraction, also formalisms based on finite observation can fulfil the criterion **Req** stated above: they enable systems to be proved correct by abstracting from potential divergence in accordance with an (implicit) fairness assumption.

Whether fair abstraction or potential divergence representation is preferable largely depends on the kind of applications one has in mind.

An example is provided by a replicated process control system within which

faulty replicas of a controller and an actuator engage in infinite chatter. A reasonable correctness requirement for this replicated system is that such divergence phenomena should not be allowed unless they are already possible for the ideal, non-replicated version of the system. Formalizing this presupposes a representation of potential divergence, as e.g. in the CSP model employed for this purpose in [Mancini & Pappalardo, 1988; Koutny, Mancini, & Pappalardo, 1991; Koutny, Mancini, & Pappalardo, 1993].

A contrasting example is a communication system built around a lossy medium and employing retransmission. Since, in general, no fixed upper bound on consecutive losses can be assumed, this system is a potentially divergent one. It is only medium fairness that ensures that messages are eventually delivered, and it is only through fair abstraction that this fairness can be taken into account in finite observation formalisms.

Often, versions of finite observation semantics in which potential divergence can be represented have been viewed as superior. This was mainly due to their being discovered after their fair abstraction counterparts and as elaborations of these. In our view, superiority is rather to be judged from practical utility, and in this respect fair abstraction, with its application to communication protocols, definitely has an edge. This is in agreement with the opinion stated in [Milner, 1989] (pages 148–149). In any case, it would be desirable at least that every finite observation semantics could be employed both in a fair abstraction and in a potential divergence version.

While this is possible for bisimulation equivalence [Walker, 1987], it will be shown in Section 2.6.2 that fair abstraction does not blend well with a large class of equivalences, ranging from simulation equivalence [Larsen, 1987] to failure equivalence. Of course, this is not to be taken as an argument against fair abstraction, but rather as a motivation for seeking a semantics that combines the advantages of failures (cf. Section 1.6.1) with those of fair abstraction. A candidate such semantics was proposed in [Bergstra, Klop, & Olderog, 1987], but, as argued in Section 2.6.2, it does not seem to be immune from problems either.

The solution we put forward in Chapter 3 is an extended failure semantics which records refused traces, rather than just actions. This approach will turn out

to be compatible with fair abstraction and also permit, like ordinary failure semantics, verification in a compositional **sat** calculus with fixpoint induction. Rather interestingly, these results have been obtained outside the traditional cpo-fixpoint theory. This is necessary because fair abstraction requires that the extended failure model should be structured as a cpo without a bottom, which prevents recursively defined process expressions from being interpreted as least fixpoints. The theory developed has therefore been based on the novel notion of 'trace-based' process functions. These can be shown to possess a particular fixpoint that, unlike the least fixpoint of traditional treatments, is compatible with fair abstraction. Moreover, they form a large class, sufficient to give a compositional denotational semantics to a widely applicable CSP-like process language.

Chapter 2

Two Process Languages

2.1 About This Chapter

After introducing some notation in Section 2.2, a CCS-like and a CSP-like process language are presented. The former, PL_+, is described in Section 2.3, together with its transitional semantics and a small set of equational laws. The latter, $PL^{\oplus}_{\uplus}$, is described in Section 2.4, where its CSP-style compositional denotational failure semantics is also outlined. The two languages, and the underlying approaches, are briefly compared in Section 2.5. It should be pointed out that later chapters focus on $PL^{\oplus}_{\uplus}$, for which a new failure semantics and **sat** calculus will be defined and applied at length; the essential reason for introducing also PL_+ is to place the important material of Section 2.6.2 in the same operational-transitional setting in which it is treated by other authors, like e.g. [Bergstra, Klop, & Olderog, 1988; Bergstra, Klop, & Olderog, 1987]) (a comparison and discussion would otherwise tend to be rather cumbersome).

Finally, Section 2.6 deals with two objectives that the standard failure semantics of [Brookes, Hoare, & Roscoe, 1984; Hoare, 1985] cannot ensure: hiding of infinite action sets and fair abstraction. A solution for infinite hiding is put forward in Section 2.6.1. The fair abstraction issue is analyzed in detail in Section 2.6.2 (its solution will be tackled in Chapter 3).

2.2 Notation

Some notational conventions that will be employed throughout the rest of this work are collected here. Most of them are in fact standard, so this section is intended mainly to be used as a reference, as the need arises.

Tuples

We define as usual the set D^Λ of tuples of elements of D over an (arbitrary) index set Λ. When a meta-variable ranges over D^Λ, this may be emphasized with a Λ subscript, as in x_Λ.

Definition 2.1 Define D^Λ to be $(\Lambda \to D)$ (the set of functions from Λ to D). Moreover, let $x_\Lambda \in D^\Lambda$. Then:

1. for all $\lambda \in \Lambda$, x_λ or $(x_\Lambda)_\lambda$ are alternative notations for $x_\Lambda(\lambda)$
2. if $d_\lambda \in D$ for all $\lambda \in \Lambda$, then $\langle d_\lambda \mid \lambda \in \Lambda \rangle$ is the member x_Λ of D^Λ such that $x_\lambda = d_\lambda$. □

Sequences

A (finite) sequence over D is either the empty sequence $\langle\rangle$ or a tuple in $D^{\{1,\ldots,N\}}$, for some $N \geqslant 1$. Sequences will be ranged over by s, t, u, v, w, z. The following conventions apply.

1. *Length*: $\#s$, the length of s, is 0 if s is $\langle\rangle$, N if $s \in D^{\{1,\ldots,N\}}$.
2. *Element*: for $1 \leqslant i \leqslant \#s$, $s(i) \in D$ is said to be the ith element of s. In the following, as in common programming languages, the notation $s[i]$ will be preferred to $s(i)$.
3. *Sequence Notation*: $\langle d_1, \ldots, d_N \rangle$ $(d_i \in D)$ is the sequence s such that $\#s = N$ and $s[i] = d_i$ for $1 \leqslant i \leqslant \#s$.

   For $K \geqslant 0$, $d \in D$, d^K is the trace s such that $\#s = K$ and $s[i] = d$ for $1 \leqslant i \leqslant K$.

4. *Prefix*: $s \leqslant t$ holds true if $\#s \leqslant \#t$ and $s[i] = t[i]$ for $1 \leqslant i \leqslant \#s$; $s < t$ holds if $s \leqslant t$ and $s \neq t$.

5. *Concatenation*: the sequence $s{\cdot}t$ is defined by $\#(s{\cdot}t) = \#s + \#t$, $(s{\cdot}t)[i] = s[i]$ for $1 \leqslant i \leqslant \#s$ and $(s{\cdot}t)[i{+}\#s] = t[i]$ for $1 \leqslant i \leqslant \#t$.

   The '$\cdot$' operator will nearly always be omitted: we write st for $s{\cdot}t$.

   Concatenation may be extended to take one sequence set operand and return a sequence set: $sT = \{st \mid t \in T\}$ and $Ts = \{ts \mid t \in T\}$; note that $s\varnothing = \varnothing s = \varnothing$.

6. *Head, last, tail*: if $\#s > 0$, define $head(s) = s[1]$, $last(s) = s[\#s]$, and $tail(s) = t$ such that $t[i] = s[i+1]$ for $1 \leqslant i \leqslant \#t = \#s{-}1$. Of course $s = head(s) \cdot tail(s)$. On the empty string argument, these functions may be assumed to return a fresh error value, when convenient.

7. *Subsequence*: if $1 \leqslant i \leqslant j \leqslant \#s$, $s[i\,..\,j]$ is the sequence t such that $\#t = j - i + 1$ and $t[k] = s[i + k - 1]$ for $1 \leqslant k \leqslant j - i + 1$; moreover $s[i..]$ is $s[i\,..\,\#s]$.

8. *Hiding*: $s\backslash E$, for $E \subseteq D$, is the sequence obtained from s by deleting elements that are in E.

9. *Projection*: $s\restriction E$, for $E \subseteq D$, is the sequence $s\backslash(D - E)$.

10. *Elements for sequences*: if $d \in D$, d may replace $\langle d \rangle$ in contexts where a sequence over D is expected; e.g. ds for the concatenation $\langle d \rangle s$, or even $d_1 \ldots d_k$ $(k > 1)$ for $\langle d_1, \ldots, d_k \rangle$.

11. *Sequences for sets*: in contexts where a set is expected, s also denotes $\{s[i] \mid 1 \leqslant i \leqslant \#s\}$, the set of the elements of sequence s.

Miscellaneous

The notation $\mathsf{p}S$ will stand for the set of finite subsets of set S.

Substitution of expression e_n for variable x_n $(1 \leqslant n \leqslant N)$ in a logical formula S will be denoted:

$$\textbf{let } x_1 := e_1, \ldots, x_N := e_N \textbf{ in } S$$

Process correctness properties will be verified within suitable formal systems, with derivations that assume the valid statements about common data types.

All other proofs are conducted informally, within elementary set and number theory. However, for the sake of conciseness, we shall sometimes take the licence to use logical quantifiers and connectives as shorthands for their natural language counterparts.

2.3 The Process Language PL_+

The first process language considered, PL_+, is a mixture of CCS [Milner, 1989] (with its explicit modelling of nondeterminism by internal actions), and CSP [Brookes, Hoare, & Roscoe, 1984; Hoare, 1985] (with its non-directed actions and synchronized parallel composition). This makes PL_+ quite similar, in form and motivation, to the standard process language LOTOS [ISO, 1989] and on the whole closer to CCS than CSP.

2.3.1 Syntax

The Basic PL_+

First of all, process expressions of PL_+ are built with actions from Act_ι (ranged over by α, while Act is ranged over by a, b). A, B will be used to range over subsets of Act. Where no confusion arises, a singleton $\{a\}$ may also be denoted simply by a.

stop is a process expression denoting a totally inactive behaviour and therefore referred to as *inaction.*

Process constants (or simply constants, ranged over by κ, η) are also needed, in order to denote a process by an identifier, as necessary for recursive definitions.

Process expressions are ranged over by p, q. Process expression tuples over the index set Λ are ranged over by p_Λ, q_Λ.

The language PL_+ of process expressions is the smallest set containing *stop*, the process constants and such that, if $p, p_1, \ldots, p_N$ are in PL_+ and p_Λ is in $(\mathrm{PL}_+)^\Lambda$, then the following are members of PL_+:

$\sum_\Lambda p_\Lambda$, also written $\sum p_\Lambda$, $\sum_{\lambda\in\Lambda} p_\lambda$ or, if $\Lambda = \{\lambda_1, \lambda_2\}$, $p_{\lambda_1} + p_{\lambda_2}$;

$\alpha \cdot p$, for every $\alpha \in Act_\iota$;

$p[f]$, for every injective $f\colon Act \to Act$;

$\|(p_1{:}B_1, \ldots, p_N{:}B_N)$, for any $B_n \subseteq Act$, $1 \leqslant n \leqslant N$; this expression is also written $(p_1{:}B_1 \,\|\, \ldots \,\|\, p_N{:}B_N)$ or $\|_{n=1}^N p_n{:}B_n$;

$p\backslash B$, for every $B \subseteq Act$;

(p). Parentheses may be spared by assuming that operator precedence decreases from postfix operators through $\alpha\cdot$, then $+$, and, lowest, infix $\|$.

A unique defining equation of the form $\kappa := p$ must be assumed to be given for every constant κ.

Some technical notions (the first three from [Milner, 1989]) will be useful.

Definition 2.2 A constant κ is *weakly* (*strongly*) *guarded* in a process expression q if it always occurs within some subexpression $\alpha \cdot q'$ (respectively $a \cdot q'$) of q.

Moreover, κ is *sequential* in q if every subexpression of q (except q and κ) in which κ occurs is of the form $\alpha \cdot q'$ or $\sum_\Lambda q_\Lambda$.

As usual $p[q/\kappa]$ is the result of replacing κ with q in p. Simultaneous indexed substitution is also admitted. □

The intended meaning of the above constructs is as follows. As in CCS, *action prefix* $\alpha \cdot p$ performs α and then behaves like p, and *choice* (or *sum*) $\sum_\Lambda p_\Lambda$ may choose to behave as any p_λ for $\lambda \in \Lambda$. *Renaming* $p[f]$ behaves like p with observable actions renamed by f. As in CSP, *hiding* $p\backslash B$ behaves like p with every action in B occurring silently, turned into ι. Again as in CSP, *parallel composition* $\|(p_1{:}B_1, \ldots, p_N{:}B_N)$ behaves like $p_1, \ldots, p_N$ operating and interacting concurrently, each with the corresponding action set B_n enforced as an interface (acting like the alphabets of [Hoare, 1985]). Thus an action $a \in Act$ is performed by

$\|(p_1{:}B_1, \ldots, p_N{:}B_N)$ iff it is in $\bigcup_{n=1}^{N} B_n$ and is performed simultaneously by every p_n such that $a \in B_n$; no synchronization is required for ι actions.

The Value-Passing PL_+

For applications, it is often convenient to describe interactions that involve the exchange of values from a *universe* set D. The ability to do so is already in the basic PL_+, but some notational conveniences are useful.

For this purpose, a set of *channels* (ranged over by c, d) is presupposed such that, if c is a channel and v a value, then $c!v \in Act$; action $c!v$ is said to *occur* at c and to *exchange* v. With some abuse, channels and actions will sometimes be confused; in particular, in appropriate contexts, sets of channels may figure in lieu of the set of actions occurring at them. A set of parametric process constants, each with an arity, is also needed; it is assumed that, if κ is such a constant with arity N and $v_1, \ldots, v_N$ are values, then $\kappa_{v_1,\ldots,v_N}$ is a process constant of the basic language. Finally, a set of value expressions or terms (ranged over by e) is assumed; as usual, terms are formed from constants, operators and variables (ranged over by x), Constants and operators are assumed to be standard both in name/arity and interpretation, which is therefore fixed; thus, when an assignment of values to the variables of a term e is provided, e evaluates (in the standard way) to a value of D. A boolean expression is a term that, given an arbitrary assignment, evaluates to a member of $Bool = \{true, false\}$, which is assumed to be a subset of D.

Value passing process expressions are formed like the basic ones, with the addition of parametric constants $\kappa(e_1, \ldots, e_N)$ (where κ has arity N and e_n is a term) and, for any value passing process expression p:

$c!e \cdot p$, for every channel c and term e;

$c?x \cdot p$, for every channel c and variable x;

$b \rightarrow p$, for every boolean expression b.

All the newly introduced constructs have the same binding power as action prefix. For every parametric constant κ with arity n, there must be a defining equation $\kappa(x_1, \ldots, x_N) := p$, where p can only contain the value variables $x_1, \ldots, x_N$.

Intuitively, if e is a variable-free term and v its value, the behaviour of $c!e \cdot p$ is to perform action $c!v$ and continue like p.

With some licence, the *multiple prefix* $c?x \cdot p$ can be thought to describe a behaviour that may perform any action $c!x$, for x in the universe D, and continue like p. A further convenience is to write $c?x:T \cdot p$, where T is an identifier denoting a set $D_T \subseteq D$ to which the choice of x is restricted.

Finally, if the *conditional* b is variable-free, $b \rightarrow p$ behaves like p if b evaluates to true, or like *stop* if b evaluates to false.

The above notation is slightly adapted from [Milner, 1989], which also shows how an extended process expression can be easily translated into the basic language, provided all its value variables are bound (in the sense of logic); the binding occurrences of variable x are of the form $c?x$.

A direct value-passing semantics is given in [Hennessy, 1991].

2.3.2 Semantics of PL_+

As discussed at length in Section 1.4.2, the first step in giving a meaning to expressions of the basic PL_+ is to make them the states of a LTS through a set of transition inference rules. Those given below for PL_+ are adapted from those of [Milner, 1989]. Action prefix and sum are straightforward:

$$\frac{p_\lambda \xrightarrow{\alpha} p'}{\sum_\Lambda p_\Lambda \xrightarrow{\alpha} p'} \ (\lambda \in \Lambda) \qquad\qquad \frac{}{\alpha \cdot p \xrightarrow{\alpha} p}$$

Renaming is also simple, especially if the renaming function $f : Act \rightarrow Act$ is extended by defining $f(\iota)$ to be ι.

$$\frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$$

Hiding is easy too, but should not be confused with CCS restriction, which would not have the first rule:

$$\frac{p \xrightarrow{a} p'}{p\backslash B \xrightarrow{\iota} p'\backslash B} \ a \in B \qquad\qquad \frac{p \xrightarrow{\alpha} p'}{p\backslash B \xrightarrow{\alpha} p'\backslash B} \ \alpha \notin B$$

For parallel composition, it is useful to extend the relation $\longrightarrow\!\triangleright$ by defining $\xrightarrow{\langle\rangle}\!\triangleright$ to be the identity relation. Also, let $a\restriction B$ be a if $a \in B$, $\langle\rangle$ otherwise, Then observable transitions of parallel composition obey the rule:

$$\frac{\forall n \in \{1,\ldots,N\}: p_n \xrightarrow{a\restriction B_n}\!\triangleright\, q_n}{\|_{n=1}^{N} p_n{:}B_n \xrightarrow{a}\!\triangleright\, \|_{n=1}^{N} q_n{:}B_n} \quad (a \in \bigcup_{n=1}^{N} B_n)$$

Letting $\varepsilon_{nk}(\iota)$ be ι if $n = k$ and $\langle\rangle$ otherwise, the internal transition rule can be expressed:

$$\frac{\forall n \in \{1,\ldots,N\}: p_n \xrightarrow{\varepsilon_{nk}(\iota)}\!\triangleright\, q_n}{\|_{n=1}^{N} p_n{:}B_n \xrightarrow{\iota}\!\triangleright\, \|_{n=1}^{N} q_n{:}B_n} \quad (1 \leqslant k \leqslant N)$$

Finally, process constants obey the rule:

$$\frac{p \xrightarrow{\alpha}\!\triangleright\, p'}{\kappa \xrightarrow{\alpha}\!\triangleright\, p'} \quad (\kappa := p)$$

Following Section 1.4.2, the semantics of PL_+ is completed by introducing the following relations over it: strong (bisimulation) congruence, (bisimulation) observation equivalence and (weak bisimulation) congruence.

2.3.3 Equational Laws for PL_+

A collection of laws for strong and weak bisimulation congruence is now given. All of them can be derived practically in the same way as in [Milner, 1989], where the issue of completeness of an equational calculus is also tackled.

As in [Milner, 1989], in view of its importance, weak bisimulation congruence between process expressions of PL_+ (cf. Section 1.4.2) will be simply denoted by the equality symbol (this of course does not apply anywhere the language PL_+ is not employed); this notion of equality is not to be confused with syntactic equality (denoted with $\equiv$, where necessary). Strong bisimulation congruence will continue to be denoted with the symbol '$\sim$'.

The laws are given with (hopefully) suggestive labels for future reference in proofs.

Strong congruence implies the weak one:

STRONG If $p \sim q$, then $p = q$.

The internal action ι may be manipulated with the axioms:

iCANC $\alpha \cdot \iota \cdot p = \alpha \cdot p$

iLAW1 $p + \iota \cdot p = \iota \cdot p$

iLAW2 $\alpha \cdot (p + \iota \cdot q) = \alpha \cdot (p + \iota \cdot q) + \alpha \cdot q$

The main properties of sum (note also $stop \sim \sum\langle\rangle$) are:

ASSO+ $p_1 + (p_2 + p_3) \sim (p_1 + p_2) + p_3$

IDEM+ $p + p \sim p$

COMM+ $p_1 + p_2 \sim p_2 + p_1$

NEUT+ $p + stop \sim p$

The *expansion law* $\mathbf{EXP}_{op}$ describes how operator op distributes over action prefix and sum. For renaming and hiding it splits into two.

$\mathbf{EXP}^{\cdot}_{[]}$ $(\alpha \cdot p)[f] \sim f(\alpha) \cdot p[f]$

$\mathbf{EXP}^{+}_{[]}$ $(\sum p_\Lambda)[f] \sim \sum\langle p_\lambda[f] \mid \lambda \in \Lambda\rangle$

$\mathbf{EXP}^{\cdot}_{\backslash}$ $(\alpha \cdot p)\backslash B \sim \iota \cdot (p\backslash B)$, if $\alpha \in B$;
$(\alpha \cdot p)\backslash B \sim \alpha \cdot (p\backslash B)$ otherwise.

$\mathbf{EXP}^{+}_{\backslash}$ $(\sum p_\Lambda)\backslash B \sim \sum\langle p_\lambda\backslash B \mid \lambda \in \Lambda\rangle$

The expansion law for parallel composition could be expressed formally, based on the corresponding transition rule, but is perhaps more understandable informally:

$\mathbf{EXP}_{\|}$ If each p_n $(1 \leqslant n \leqslant N)$ is a sum of action prefixes, then

$$\|_{n=1}^{N} p_n{:}B_n \sim \sum\{\alpha \cdot \|_{n=1}^{N} q_n{:}B_n \mid \alpha, q_1, \ldots, q_N \text{ are as in (1) or (2)}\}$$

where:

1. $\alpha = \iota$, $\iota \cdot q_k$ is a summand of p_k, and q_n is p_n for $n \neq k$ $(1 \leqslant n \leqslant N,\ 1 \leqslant k \leqslant N)$;

2. $\alpha \in \bigcup_{n=1}^{N} B_n$, and if $\alpha \in B_n$ $(1 \leqslant n \leqslant N)$, then $\alpha \cdot q_n$ is a summand of p_n, otherwise q_n is p_n.

The next group of rules deals with tuples κ_Λ of distinct process constants such that, for every $\lambda \in \Lambda$, $\kappa_\lambda := p_\lambda$ and p_λ contains no process constant but those in κ_Λ. The constants may be seen as solutions of their defining equations:

FIX $\kappa_\Lambda \sim p_\Lambda$

where $\sim$ and $=$ have been extended to tuples. Uniqueness of this solution (also up to $=$) depends on suitable side conditions (introduced in Definition 2.2):

UNIQ$_\sim$ If $q_\Lambda \sim p_\Lambda[q_\Lambda/\kappa_\Lambda]$ then $q_\Lambda \sim \kappa_\Lambda$, provided $\kappa_{\lambda'}$ is weakly guarded in p_λ for all $\lambda, \lambda' \in \Lambda$.

UNIQ$_=$ If $q_\Lambda = p_\Lambda[q_\Lambda/\kappa_\Lambda]$ then $q_\Lambda = \kappa_\Lambda$, provided $\kappa_{\lambda'}$ is strongly guarded and sequential in p_λ for all $\lambda, \lambda' \in \Lambda$.

2.4 The Process Language $PL^{\oplus}_{\uplus}$

The other process language considered, $PL^{\oplus}_{\uplus}$, practically coincides (except for minor syntactic features) with CSP [Brookes, Hoare, & Roscoe, 1984]. The main difference between $PL^{\oplus}_{\uplus}$ and PL_+ is that internal actions are not modelled explicitly; instead, $PL^{\oplus}_{\uplus}$ has an operator to express *internal* choice, which cannot be controlled by the environment.

2.4.1 Syntax

Not surprisingly, only observable actions of Act are employed in the construction of process expressions of $PL^{\oplus}_{\uplus}$. The notational conventions applicable carry over from those established for PL_+. Process constants, inaction, renaming, hiding and parallel composition have syntax and intended meaning as in PL_+. The choice and action prefix of PL_+ are instead replaced by assuming that, if p_1, p_2 are in $PL^{\oplus}_{\uplus}$, p_Λ is in $(PL^{\oplus}_{\uplus})^\Lambda$ and p_A is in $(PL^{\oplus}_{\uplus})^A$ for $A \subseteq Act$, then the following are members of $PL^{\oplus}_{\uplus}$:

$\uplus_\Lambda p_\Lambda$, also written $\uplus_{\lambda\in\Lambda} p_\lambda$ or, if $\Lambda = \{\lambda_1, \lambda_2\}$, $p_{\lambda_1} \uplus p_{\lambda_2}$;

$p_1 \oplus p_2$;

$A; p_A$, also written $a; p$ if $A = \{a\}, p_a = p$;

The process expression $\uplus_\Lambda p_\Lambda$ is referred to as *internal* or *non-deterministic choice* (among the behaviours p_λ for $\lambda \in \Lambda$). The process expression $p_1 \oplus p_2$ is instead (environment-driven) *external choice* (between the behaviours of p_1 and p_2). Finally, *multiple action sequence* $A;p_A$ denotes a behaviour that may deterministically choose to begin by any action a in A and continues like p_a; but, in addition, any p_a, for $a \in A$, is allowed to make internal progress while waiting for one of the initial actions in A. This feature is adopted from [Brookes, Hoare, & Roscoe, 1984] and entails that the behaviour of $A; p_A$ cannot be straightforwardly expressed in $\mathrm{PL}^{\oplus}_{\uplus}$ with a combination of the sum and action prefix of PL_+.

A value passing version of $\mathrm{PL}^{\oplus}_{\uplus}$ can be introduced along the lines employed for PL_+ (note that multiple action sequence is already in the basic language).

2.4.2 Transitional Semantics

It would be easy to define a relation $\longrightarrow$ for $\mathrm{PL}^{\oplus}_{\uplus}$, via a set of inference rules analogous to those for PL_+. However, for future developments, it is enough to provide some results only for the derived transition relation $\Longrightarrow$ over $\mathrm{PL}^{\oplus}_{\uplus} \times Act^* \times \mathrm{PL}^{\oplus}_{\uplus}$ defined as in Section 1.4.3. It can be proved in a straightforward manner that the $\Longrightarrow$ transitions are exactly those derivable from the following rules.

Relation $\Longrightarrow$ is, in a sense, reflexive:

$$\frac{}{p \overset{\langle\rangle}{\Longrightarrow} p}$$

and transitive:

$$\text{if } p \overset{s}{\Longrightarrow} p' \text{ and } p' \overset{t}{\Longrightarrow} q, \text{ then } p \overset{st}{\Longrightarrow} q$$

Rules for constants, renaming, hiding and parallel composition are clearly related to the corresponding ones for $\longrightarrow$ (as given earlier for PL_+):

$$\frac{p \overset{s}{\Longrightarrow} p'}{\kappa \overset{s}{\Longrightarrow} p'} \ (\kappa := p)$$

$$\frac{p \stackrel{s}{\Longrightarrow} p'}{p[f] \stackrel{f(s)}{\Longrightarrow} p'[f]} \qquad \frac{p \stackrel{s}{\Longrightarrow} p'}{p\backslash B \stackrel{s\backslash B}{\Longrightarrow} p'\backslash B}$$

$$\frac{\forall n \in \{1,\ldots,N\}\colon p_n \stackrel{s\restriction B_n}{\Longrightarrow} q_n}{\|_{n=1}^{N} p_n{:}B_n \stackrel{s}{\Longrightarrow} \|_{n=1}^{N} q_n{:}B_n} \quad (s \in (\bigcup_{n=1}^{N} B_n)^*)$$

The rules for internal and external choice, and multiple action sequence reflect the intended meaning of these operators.

$$\frac{p_\lambda \stackrel{s}{\Longrightarrow} p'}{\uplus_\Lambda p_\Lambda \stackrel{s}{\Longrightarrow} p'} \quad (\lambda \in \Lambda)$$

$$\frac{p \stackrel{s}{\Longrightarrow} p'}{p \oplus q \stackrel{s}{\Longrightarrow} p'} \quad (s \neq \langle\rangle) \qquad \frac{q \stackrel{s}{\Longrightarrow} q'}{p \oplus q \stackrel{s}{\Longrightarrow} q'} \quad (s \neq \langle\rangle)$$

While waiting for an external choice to be resolved, its operands are allowed by the next rule to make internal progress (even independently, because $q \stackrel{\langle\rangle}{\Longrightarrow} q$ for all q):

$$\frac{p \stackrel{\langle\rangle}{\Longrightarrow} p', q \stackrel{\langle\rangle}{\Longrightarrow} q'}{p \oplus q \stackrel{\langle\rangle}{\Longrightarrow} p' \oplus q'}$$

Internal progress is also possible for multiple action sequence:

$$\frac{\forall a \in A\colon p_a \stackrel{\langle\rangle}{\Longrightarrow} p'_a}{A; p_A \stackrel{\langle\rangle}{\Longrightarrow} A; p'_A} \qquad \frac{p_a \stackrel{s}{\Longrightarrow} p'}{A; p_A \stackrel{as}{\Longrightarrow} p'} \quad (a \in A)$$

Since $PL^{\oplus}_{\uplus}$ can be mapped, like PL_+, onto a LTS, its process expressions can be compared by any observation equivalence for LTSs.

2.4.3 Compositional Denotational Semantics

As noted in Section 1.4.3, an alternative to transitional semantics is to identify a process with a set of observations, and, letting *Proc* (ranged over by P, Q) stand for the set of processes, try to define a compositional meaning function $\mathcal{O}[\![\ ]\!]\colon PL^{\oplus}_{\uplus} \to Proc$.

This definition is based on the association of every Λ-ary operator *op* of $PL^{\oplus}_{\uplus}$ with a function $Op\colon Proc^{\Lambda} \to Proc$. For simplicity, but without harm for practical applications, it will be assumed that κ_Γ is a tuple of distinct constants that are

the only ones occurring in process expressions, and (as usual) there is a unique definition $\kappa_\gamma := q_\gamma$ for every $\gamma \in \Gamma$. First, every expression p is compositionally mapped onto a function $\hat{\mathcal{O}}[\![p]\!]: Proc^\Gamma \to Proc$:

Definition 2.3 The semantic functional $\hat{\mathcal{O}}[\![\,]\!]: \mathrm{PL}^{\oplus}_{\uplus} \to (Proc^\Gamma \to Proc)$ is inductively defined by letting, for $P_\Gamma \in Proc^\Gamma$:

$$\hat{\mathcal{O}}[\![\, op(p_\Lambda)]\!](P_\Gamma) = Op(\langle \hat{\mathcal{O}}[\![p_\lambda]\!](P_\Gamma) \mid \lambda \in \Lambda \rangle) \quad (op \text{ a } \Lambda\text{-ary operator}) \quad (2\text{-}1)$$

$$\hat{\mathcal{O}}[\![\kappa_\gamma]\!](P_\Gamma) = P_\gamma \qquad (\gamma \in \Gamma) \quad (2\text{-}2)$$

Process functions obtained by application of $\hat{\mathcal{O}}[\![\,]\!]$ are said to be $\mathrm{PL}^{\oplus}_{\uplus}$-*generated*:

Definition 2.4 A function $F: Proc^\Gamma \to Proc$ is said to be $\mathrm{PL}^{\oplus}_{\uplus}$-generated if F is $\hat{\mathcal{O}}[\![p]\!]$ for some process expression p in $\mathrm{PL}^{\oplus}_{\uplus}$.

A function tuple $\langle F_\gamma: Proc^\Gamma \to Proc \mid \gamma \in \Gamma \rangle$, viewed as a function $F: Proc^\Gamma \to Proc^\Gamma$, is said to be $\mathrm{PL}^{\oplus}_{\uplus}$-generated if every F_γ is $\mathrm{PL}^{\oplus}_{\uplus}$-generated. □

The semantic function proper $\mathcal{O}[\![\,]\!]: \mathrm{PL}^{\oplus}_{\uplus} \to Proc$ can now be defined:

Definition 2.5 Let every $\mathrm{PL}^{\oplus}_{\uplus}$-generated function $F: Proc^\Gamma \to Proc^\Gamma$ possess a particular fixpoint $\mathrm{fix}\, F$. Then (recalling $\kappa_\gamma := q_\gamma$ for $\gamma \in \Gamma$) define, for every process expression p:

$$\mathcal{O}[\![p]\!] = \hat{\mathcal{O}}[\![p]\!](\mathrm{fix}\, \langle \hat{\mathcal{O}}[\![q_\gamma]\!] \mid \gamma \in \Gamma \rangle) \qquad \square$$

In sum, whatever the observation set model *Proc*, defining a compositional denotational semantics for $\mathrm{PL}^{\oplus}_{\uplus}$ over *Proc* amounts to defining: (i) a function $Op: Proc^\Lambda \to Proc$ for every Λ-ary operator op of $\mathrm{PL}^{\oplus}_{\uplus}$, and (ii) a fixpoint for $\mathrm{PL}^{\oplus}_{\uplus}$-generated process functions.

Standard Failure Semantics

In [Brookes, Hoare, & Roscoe, 1984], failures are employed as observations. It is worth recalling, from Section 1.4.3, that a failure is a trace-refusal pair in $Act^* \times \mathrm{p}Act$, and that a failure set suitable as a process must satisfy non-emptiness, prefix closure, inclusion closure and trace-refusal consistency (Definition 1.1). Let then

Proc denote the set of such failure sets. In accordance with the scheme outlined above, for every operator of $PL^{\oplus}_{\uplus}$, a suitable process function is now introduced, which, for simplicity, is still denoted like the operator. Note that these functions are well-defined over any failure sets (in $Act^* \times \mathsf{p}Act$) and process-preserving [Brookes, Hoare, & Roscoe, 1984].

$$
\begin{aligned}
STOP &= \{(\langle\rangle, X) \mid X \in \mathsf{p}Act\} \quad \text{(the denotation of } stop\text{)} \\
\biguplus_{\Lambda} P_\Lambda &= \bigcup_{\lambda \in \Lambda} P_\lambda \\
P \oplus Q &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in P \cap Q\} \cup \\
&\quad \{(s, X) \mid s \neq \langle\rangle,\ (s, X) \in P \cup Q\} \\
A; P_A &= \{(\langle\rangle, X) \mid X \in \mathsf{p}(Act - A)\} \cup \\
&\quad \{(as, X) \mid a \in A,\ (s, X) \in P_a\} \\
P[f] &= \{(f(s), f(X)) \mid (s, X) \in P\} \qquad (f \text{ injective}) \\
\|_{n=1}^{N} P_n{:}B_n &= \{(s, Y \cup \bigcup_{n=1}^{N} X_n) \mid s \in B^*,\ Y \in \mathsf{p}(Act - B), \\
&\quad\quad \forall n{:}\ X_n \subseteq B_n,\ (s{\restriction}B_n, X_n) \in P_n\} \qquad (B = \bigcup_{n=1}^{N} B_n)
\end{aligned}
$$

The definition of $P\backslash B$, which is based on that of the trace set $\delta_B P$, makes sense only for a finite B:

$$
\begin{aligned}
s \in \delta_B P \quad &\text{iff} \quad \{u \in B^* \mid (su, \varnothing) \in P\} \text{ are unbounded (in length)} \\
P\backslash B &= \{(s\backslash B, X) \mid (s, X \cup B) \in P\} \cup \\
&\quad \{((s\backslash B)t, X) \mid t \in Act^*,\ X \in \mathsf{p}Act,\ s \in \delta_B P\}
\end{aligned}
$$

To complete the definition of failure semantics, *Proc* is given the structure of a cpo and the above functions are shown to be continuous. This ensures that every $PL^{\oplus}_{\uplus}$-generated function $F\colon Proc^{\Gamma} \to Proc^{\Gamma}$ is continuous, so $\mathrm{fix}\, F$ can be taken to be the least fixpoint of F.

A different denotational semantics for $PL^{\oplus}_{\uplus}$, motivated by compatibility with fair abstraction (cf. Section 2.6.2), is proposed in Chapter 3.

2.5 $PL^{\oplus}_{\uplus}$ versus PL_{+}

Which of the approaches represented by PL_{+} and $PL^{\oplus}_{\uplus}$ is preferable is a highly debated issue, which is rather difficult to answer definitively. This explains why both approaches boast a long-standing tradition in the literature and have failed to prevail or converge so far, although their relationship has been investigated at length (see e.g. [Brookes, 1983a; Brookes, 1983b]).

Formally, both languages can be mapped onto a LTS, so both can be endowed with an observational equivalence like $\approx$. However, in our view, many equational laws for PL_{+} tend to take a more suggestive form than their counterparts for $PL^{\oplus}_{\uplus}$, thanks to the presence of the explicit internal action ι. On the other hand, exactly for the same reason, a compositional denotational semantics is more easily introduced for $PL^{\oplus}_{\uplus}$; indeed, giving such a semantics to PL_{+} is possible, but requires the observation set to contain also 'instability information', which makes the definition of process semantic functions more cumbersome.

The practical upshot of these differences is that PL_{+} may prove more useful as a basis for equational verification, whereas $PL^{\oplus}_{\uplus}$ lends itself more easily to the design of a **sat** calculus for reasoning about input-output relation and deadlock. Since the latter is our ultimate goal, the rest of this work is mainly based on $PL^{\oplus}_{\uplus}$. The only exception, and the reason for introducing PL_{+} and its equational calculus, is represented by the important Section 2.6.2; this establishes the incompatibility between fair abstraction and traditional failure semantics, thus motivating the extension proposed in Chapter 3. Below, introducing Section 2.6.2, we also put the case for preferring PL_{+} in it.

2.6 Problems with Failure Semantics

This section deals with two problems presented by the denotational failure semantics of Section 2.4.3. As already discussed informally in Section 1.6.2, they are the impossibility of hiding infinite sets of actions and applying fair abstraction to divergence. It will be seen that the latter problem is in fact common to any failure semantics.

2.6.1 Hiding Infinite Action Sets: A Solution

The hiding function of [Brookes, Hoare, & Roscoe, 1984], reproduced below for convenience from Section 2.4.3, does not allow an infinite action set B to be hidden.

$$\begin{aligned} s \in \delta_B P \;\text{ iff }\; & \{u \in B^* \mid (su, \varnothing) \in P\} \text{ is unbounded} \qquad (2\text{-}3) \\ P\backslash B \;=\; & \{(s\backslash B, X) \mid (s, X \cup B) \in P\} \cup \\ & \{((s\backslash B)t, X) \mid t \in Act^*,\ X \in \mathsf{p}Act,\ s \in \delta_B P\} \end{aligned}$$

The reason, clearly, is that in the latter equation the set $X \cup B$, being infinite, cannot be a refusal of P (by Definition 1.1 a refusal is a finite set). A possible remedy could be to redefine $P\backslash B$ thus:

$$\begin{aligned} P\backslash B \;=\; & \{(s\backslash B, X) \mid X \in \mathsf{p}Act,\ \forall Y \in \mathsf{p}(X \cup B) : (s, Y) \in P\} \cup \qquad (2\text{-}4) \\ & \{((s\backslash B)t, X) \mid t \in Act^*,\ X \in \mathsf{p}Act,\ s \in \delta_B P\} \end{aligned}$$

This exploits the same 'compactness' closure condition whereby infinite refusal sets were introduced in [Brookes & Roscoe, 1985]. Unfortunately, hiding, redefined as above and extended to infinite sets, is not associative. A counterexample is provided by P below:

$$\begin{aligned} B \;&=\; \{b_1, b_2, \ldots\} \quad \text{(an infinitely countable set)} \\ P \;&=\; \{(s, X) \mid s \in T,\ sX \cap T = \varnothing\} \\ T \;&=\; \bigcup_{n>0} \{t \mid t \leqslant b_n a c^n\} \end{aligned}$$

Indeed, $\delta_{B\cup\{c\}} P = \varnothing$, so the only non-empty trace of $P\backslash(B\cup\{c\})$ is $\langle a \rangle$. In contrast, $(b_n a c^n, X) \in P$ for all n and $X \in \mathsf{p}Act^+$, which implies $(ac^n, \varnothing) \in P\backslash B$ for all n, so $a \in \delta_{\{c\}} P\backslash B$, whence, for all $w \in Act^*$, $(aw, \varnothing) \in P\backslash B\backslash\{c\}$.

We shall show that this problem can be obviated, if also $\delta_B P$ is redefined thus:

$$s \in \delta_B P \;\text{ iff }\; \{u \in B^* \mid (su, \varnothing) \in P\} \text{ is infinite} \qquad (2\text{-}5)$$

It should be noted that, for B finite (by König's lemma) this definition coincides with the previous one (2-3), and that of hiding coincides with the original one of [Brookes, Hoare, & Roscoe, 1984].

It will be shown first, after a preliminary lemma, that the new hiding function defined with (2-4) and (2-5) is process-preserving.

Lemma 2.6 If P is a process and $(s, \varnothing) \in P$, then $(s\backslash B, \varnothing) \in P\backslash B$.

Proof. If $s \in \delta_B P$, the lemma follows immediately. If, instead, the set $U = \{u \in B^* \mid (su, \varnothing) \in P\}$ is finite, let w be one of its maximal traces; the lemma then follows from $(sw, Y) \in P$ for all finite $Y \subseteq B$. To see this, suppose by contradiction $(sw, Y \cup \{b\}) \notin P$ and $(sw, Y) \in P$ for a finite $Y \subseteq B$. Then, by trace-refusal consistency, $(swb, \varnothing) \in P$, which contradicts the maximality of w. □

Theorem 2.7 If P is a process, so is $P\backslash B$.

Proof. The only non-trivial proofs are those of prefix-closure and trace-refusal consistency.

Suppose $(st, \varnothing) \in P\backslash B$. If $(w, \varnothing) \in P$ and $w\backslash B = st$ choose $u \leqslant w$ such that $u\backslash B = s$; then $(u, \varnothing) \in P$ by prefix-closure of P and $(s, \varnothing) \in P\backslash B$ by the previous lemma. Otherwise, choose $w \in \delta_B P$ such that $w\backslash B \leqslant st$. There are two cases for w. If $w\backslash B \leqslant s$, then $(s, \varnothing) \in P\backslash B$ is obvious. If instead $s < w\backslash B \leqslant st$, choose $u \leqslant w$ such that $u\backslash B = s$; then $(u, \varnothing) \in P$ (by $(w, \varnothing) \in P$ and prefix-closure of P) and $(s, \varnothing) \in P\backslash B$ by the previous lemma.

To prove trace-refusal consistency, suppose $(s, X) \in P\backslash B$ and $(sa, \varnothing) \notin P\backslash B$. If $a \in B$, it is obvious that $(s, X \cup \{a\}) \in P\backslash B$. If $a \notin B$, there is no $t \in \delta_B P$ for which $t\backslash B \leqslant s$ or $(sa, \varnothing) \in P\backslash B$. Thus for some t, $t\backslash B = s$ and $(t, X \cup Y) \in P$ for all finite $Y \subseteq B$. Now $(ta, \varnothing) \notin P$ or, by the previous lemma, the assumption $(sa, \varnothing) \notin P\backslash B$ would be contradicted. So trace-refusal consistency of P implies $(t, X \cup Y \cup \{a\}) \in P$ for all finite $Y \subseteq B$, whence, as desired, $(s, X \cup \{a\}) \in P\backslash B$. □

The last proofs have closely followed the style of their counterparts in [Brookes, Hoare, & Roscoe, 1984]. However, the proof that hiding is associative is based on a thoroughly new lemma.

Lemma 2.8 Let B be a (possibly infinite) action set and S an infinite trace set. If $S\backslash B$ is a finite set, then there exist a trace x and an infinite subset $W \subseteq B^*$ such that the traces in xW are prefixes of S.

Proof. For at least a trace $\langle a_1, \ldots, a_M \rangle \in S\backslash B$, there must be an infinite subset $S' \subseteq S$ such that $S'\backslash B = \{\langle a_1 \ldots a_M \rangle\}$. If $M = 0$, just take $W = S'$. For $M > 0$, it is possible to define $u_n^s \in B^*$ (for $0 \leqslant n \leqslant M$) such that:

(1) if $s \in S'$, then $s = u_0^s\, a_1 \ldots u_{M-1}^s\, a_M\, u_M^s$.

For $0 \leqslant m \leqslant M$, defining

$$\begin{aligned} U_m &= \{u_m^s \mid s \in S'\} \\ U'_m &= \{u_0^s\, a_1 \ldots u_{m-1}^s\, a_m \mid s \in S'\} \\ U_{m,x} &= \{u_m^s \mid s \in S',\ u_0^s\, a_1 \ldots u_{m-1}^s\, a_m = x\} \end{aligned}$$

(and letting $u_0^s\, a_1 \ldots u_{m-1}^s\, a_m = \langle\rangle$ if $m = 0$) yields

(2) $$U_m = \bigcup_{x \in U'_m} U_{m,x}$$

Now let m be the minimum index for which U_m is infinite (this m must exist or by (1) S' would be finite). Then U'_m is finite, so by (2) we can choose $x \in U'_m$ that makes infinite the set $U_{m,x}$. This $U_{m,x}$ is the sought W, because if $w \in U_{m,x}$, then some $s \in S' \subseteq S$ satisfies $w = u_m^s$ and $x = u_0^s\, a_1 \ldots u_{m-1}^s\, a_m$, whence (using (1) for the last relation)

$$xw = u_0^s\, a_1 \ldots u_{m-1}^s\, a_m\, u_m^s \leqslant s \qquad \square$$

The latter result is a non-trivial extension of Lemma 2 of [Brookes, Hoare, & Roscoe, 1984] (where B is assumed to be finite). It will be employed below in the form of the following lemma.

Lemma 2.9 Let B be an action set, P a process and tU an infinite subset of its traces.

Then if $U \backslash B$ is finite, there exists a prefix x of some trace of U such that $tx \in \delta_B P$.

Proof. By the previous lemma, there exist a trace x and an infinite $W \subseteq B^*$ such that all the traces in xW are prefixes of traces in U. Moreover, all the traces in txW belong also to P, so $tx \in \delta_B P$. $\square$

The associativity proof is split into two halves.

Theorem 2.10 Let P be a process and B, C action sets. Then $P \backslash B \backslash C \supseteq P \backslash B \cup C$.

Proof. Assume $(s, X) \in P\backslash B \cup C$. By the definition of hiding there can be two cases.

If $t\backslash(B \cup C) = s$ and, for all finite $Z \subseteq C$ and $Y \subseteq B$, $(t, X \cup Y \cup Z) \in P$, then, for all finite $Z \subseteq C$, $(t\backslash B, X \cup Z) \in P\backslash B$ and $(t\backslash B\backslash C, X) \in P\backslash B\backslash C$, as desired.

If, on the other hand, $t \in \delta_{B\cup C}P$ and $(t\backslash B \cup C) \leqslant s$, then $(s, X) \in P\backslash B\backslash C$ may be obtained by showing:

$$t\backslash B \in \delta_C(P\backslash B) \tag{1}$$

For this purpose, observe that the set $U = \{u \in (B \cup C)^* \mid (tu, \varnothing) \in P\}$ must be infinite and prefix-closed and the traces in tU are also traces of P. Now, if $U\backslash B$ is infinite, then (using Lemma 2.6) $t\backslash B \in \delta_C(P\backslash B)$, and (1) holds. If instead $U\backslash B$ is finite, by Lemma 2.9, there exists a trace $x \in (B\cup C)^*$ such that $tx \in \delta_B P$. Hence, by equation (2-4), $((t\backslash B)(x\backslash B)u, \varnothing) \in P\backslash B$ for all $u \in C^*$, so (1) holds again. □

The second half of the proof is preceded by a technical result that will be needed again in later sections.

Lemma 2.11 Let Ξ be a non-empty set of sets, ordered with inclusion. Suppose every finite subset of Ξ has an upper bound in Ξ.[1]

Let $f: \Xi \to E$ be a function under which Ξ has a finite image. Then there exists $e \in E$ such that for all $Z \in \Xi$ there is $Z' \in \Xi$ such that $Z' \supseteq Z$ and $f(Z') = e$.

Proof. Let $f(\Xi) = \{e_1, \ldots, e_H\}$ $(H > 0)$. By contradiction, suppose:

$$\forall h \in \{1, \ldots, H\}\colon\ \exists Z_h \in \Xi\colon\ \forall Z' \in \Xi\colon\ Z' \supseteq Z_h \Rightarrow f(Z') \neq e_h$$

By hypothesis, there must exist $Z' \in \Xi$ such that, for all $h \in \{1, \ldots, H\}$, $Z' \supseteq Z_h$, whence $f(Z') \neq e_h$, Thus $f(Z') \notin f(\Xi)$: a contradiction. □

Theorem 2.12 Let P be a process, B, C action sets. Then $P\backslash B\backslash C \subseteq P\backslash B \cup C$.

Proof. Assume $(w, X) \in P\backslash B\backslash C$. It is necessary to find s such that either of the two following statements holds:

[1] In practice, Ξ is usually such that this upper bound is the lub/union.

(1) $s\backslash B\backslash C = w$ and, for all $Y \in \mathsf{p}B$ and $Z \in \mathsf{p}C$, $(s, X \cup Y \cup Z) \in P$; or:

(2) $s \in \delta_{B\cup C}P$ and $s\backslash B\backslash C \leqslant w$.

There are two cases why $(w, X) \in P\backslash B\backslash C$, treated as **A** and **B** below.

Case **A**. There exists t such that:

(3) $t\backslash C = w$ and, for all $Z \in \mathsf{p}C$, $(t, X \cup Z) \in P\backslash B$.

Again, this statement splits into two subcases.

If there exists at least one $s \in \delta_B P$ such that $s\backslash B \leqslant t$, then (2) holds.

Otherwise (using 3) assume:

(4) for all $Z \in \mathsf{p}C$, there is s_Z such that $s_Z\backslash B = t$ and, for all $Y \in \mathsf{p}B$, $(s_Z, X \cup Y \cup Z) \in P$ holds.

and let $S = \{s_Z \mid Z \in \mathsf{p}C\}$, so that $S \subseteq \tau P$ and $S\backslash B = \{t\}$. If S is infinite, by Lemma 2.9 there exists $s \in \delta_B P$ prefix of some s_Z, and (2) can be seen to hold. If S is finite, by Lemma 2.11 there exists $s \in S$ that, using (4), can be recognized to satisfy (1).

Case **B**. There exists t such that:

(5) $t\backslash C \leqslant w$ and $t \in \delta_C(P\backslash B)$, so that $U = \{u \in C^* \mid (tu, \varnothing) \in P\}$ is infinite.

Two subcases may be considered.

If there is $u \in U$ such that, for some $s \in \delta_B P$, $s\backslash B \leqslant tu$, then (2) holds.

Otherwise (using the definition of U in (5)) assume:

(6) for all $u \in U$, there is s_u such that $s_u\backslash B = tu$ and, for all $Y \in \mathsf{p}B$, $(s_u, Y) \in P$.

and let $S = \{s_u \mid u \in U\}$, so that $S\backslash B\backslash C = (tU)\backslash C = \{t\backslash C\}$. S is infinite (or $S\backslash B = tU$ would be finite too, contradicting (5)). Then by Lemma 2.9 there exists $s \in \delta_{B\cup C}P$ prefix of some s_u, for $u \in U$, and (2) can be seen to hold. □

With respect to the process partial order of [Brookes, Hoare, & Roscoe, 1984], the proposed hiding function is not continuous, but is easily seen to be monotonic.

Discussion

The new hiding function might also appear to suffer from a weakness, as it enlarges the set $\delta_B P$ of traces of P after which $P\backslash B$ behaves chaotically (being permitted to do or refuse anything by (2-4)), from that of equation (2-3) to that of (2-5): it is now possible for $P\backslash B$ to degenerate into chaos also if, at a certain stage, P can choose among infinitely many actions from B. However, in practice this is rather unlikely to occur at an internal channel c of a real system. Usually, subsystems do not offer each other at c infinitely many values upon which to synchronize: while one of them, which is therefore thought to use c as an *input* channel, may be prepared to accept any value from an infinite set, the other is normally willing to *output* just a finite number of values.[2] The resulting match will then give rise to only finitely many options at c, which avoids chaotic behaviour even with the new hiding.

Another problem with the new function is that, while it is monotonic and hence endowed with a least fixpoint (by Knaster-Tarski's theorem, see [Loeckx & Sieber, 1987]), it is not continuous. Thus, fixpoint induction cannot be employed, as in [Brookes, Hoare, & Roscoe, 1984], to reason about process constants that occur within the scope of hiding in their recursive defining equations. This, however, is rather unlikely to happen in practical system specifications, which instead, as discussed, often need to hide channels that may exchange an infinite number of data.

Finally, it should be observed that, in any case, the new hiding is not inferior to the old one, in that they coincide if the hidden set B is finite (as required by the old one). It seems therefore acceptable to conclude that the new hiding function widens the applicability of the language $\mathrm{PL}^{\oplus}_{\omega}$ with failure semantics. Furthermore, we conjecture that the noted tradeoff between continuity and infinite set hiding is related by a subtle interplay to the requirement of finiteness for refusal sets. We plan to investigate this issue in future work.

[2] An exception could be a random generator of infinitely many values, but this would only represent a modelling problem for our version of hiding if this infinity were crucial for correctness.

2.6.2 Failures and Fair Abstraction

We will now substantiate the claim, made in Section 1.6.2, that fair abstraction is incompatible with failure semantics, whether it is defined denotationally, as in Section 2.4.3 for $\mathrm{PL}^{\oplus}_{\uplus}$, or operationally through a LTS. More precisely, an inconsistency will be derived from a set of equational laws, of which one reflects fair abstraction, while the others must necessarily hold in failure semantics, and indeed in a lot more: possible futures and ready simulation semantics together with all coarser ones, which include simulation, ready trace, ready, and failure trace semantics (see [v. Glabbeek, 1993] for a definition and a survey). That so many semantics can be treated with a single argument is an obvious benefit of the equational approach.

In fact, similar incompatibility results were already given in [Bergstra, Klop, & Olderog, 1988]. However, we provide a simpler counterexample and, more importantly, a weaker and smaller set of incompatible equational laws; this permits the results to be extended from ready and failure semantics to the many others cited. Furthermore, the incompatibility established would appear to confute successfully the contrary view put forward in [Bergstra, Klop, & Olderog, 1987].

While it would be possible to recast the following treatment in the setting of $\mathrm{PL}^{\oplus}_{\uplus}$, this would make a comparison with the cited works rather cumbersome, and would prevent a direct use of their results. It can be added that the process properties that will be exploited, and the spirit of the argument altogether, are more clearly conveyed in terms of explicit internal actions, in an explicit operational-transitional framework.

We will therefore work with the language PL_{+} and, for simplicity, will assume that only finitary choice is employed. We postulate that this language is endowed with a congruence $\simeq$ satisfying the laws described below.

To begin with, we assume the laws $\mathbf{EXP}_{\setminus}$, **NEUT+**, **FIX** and $\mathbf{UNIQ}_{\simeq}$, obtained from $\mathbf{EXP}_{\setminus}$, **NEUT+**, **FIX** and $\mathbf{UNIQ}_{=}$ of Section 2.3.3 by replacing $\sim$ or $=$ by $\simeq$. A weak distributivity of choice over action prefix is also required:

$$\mathbf{DISTR}^{-} \qquad a \cdot \sum_{n=1}^{N} b_n \cdot p_n \;\simeq\; (a \cdot \sum_{n=1}^{N} b_n \cdot p_n) + (a \cdot \sum_{h=1}^{H} b_{n_h} \cdot p_{n_h}),$$

$$\text{provided } \bigcup_{n=1}^{N}\{b_n\} = \bigcup_{h=1}^{H}\{b_{n_h}\}.$$

Example applications of this law are (the second will be repeated in Proposition 2.13):

$$\begin{aligned} a\cdot(b\cdot p+b\cdot q+c\cdot r) &= a\cdot(b\cdot p+b\cdot q+c\cdot r)+a\cdot(b\cdot p+c\cdot r)\\ a\cdot(b\cdot p+b\cdot q+b\cdot r) &= a\cdot(b\cdot p+b\cdot q+b\cdot r)+a\cdot b\cdot q \end{aligned}$$

Finally, we need a law to capture fair abstraction of tight ι-loops. 'Koomen's fair abstraction rule' [Koomen, 1985], in the formulation of [Bergstra, Klop, & Olderog, 1988], is one such law:

KFAR$_1$ Let $p \simeq b\cdot p+q$. Then $p\backslash b \simeq \iota\cdot(q\backslash b)$.

We are now ready to state:

Proposition 2.13 Let $\simeq$ be a congruence over PL_+ and suppose it satisfies the laws **EXP$_\backslash$**, **NEUT+**, **FIX**, **UNIQ$_\simeq$**, **DISTR$^-$** and **KFAR$_1$**.[3] Then, rather unexpectedly:

$$\iota\cdot\iota\cdot a\cdot stop \simeq \iota\cdot(\iota\cdot a\cdot stop+\iota\cdot\iota\cdot stop) \qquad (2\text{-}6)$$

Proof. Let us define:

$$\kappa := b\cdot\kappa+b\cdot a\cdot stop \qquad \eta := b\cdot\eta+b\cdot\zeta+b\cdot a\cdot stop \qquad \zeta := b\cdot\zeta$$

Then the laws selected suffice for the following derivation:

$$\begin{aligned} \eta &\simeq b\cdot(b\cdot\eta+b\cdot\zeta+b\cdot a\cdot stop)+b\cdot b\cdot\zeta+b\cdot a\cdot stop && \textbf{FIX for } \eta,\eta,\zeta\\ &\simeq b\cdot(b\cdot\eta+b\cdot\zeta+b\cdot a\cdot stop)+b\cdot a\cdot stop && \text{by } \textbf{DISTR}^-\\ &\simeq b\cdot\eta+b\cdot a\cdot stop && \textbf{FIX for } \eta \end{aligned}$$

So **UNIQ$_\simeq$** implies:

(1) $\qquad \kappa \simeq \eta$

[3] In fact, the proof shall be given as though the operator + denoted multiple finitary, rather than binary, choice (more precisely, we should either replace + by $\sum$ or invoke also the associativity of +).

In [Bergstra, Klop, & Olderog, 1988], a stronger distributivity and their version of **iCANC** (here $\alpha\cdot\iota\cdot p \simeq \alpha\cdot p$) were also needed for a different counterexample.

The fair abstraction law, with the other laws in the second column below, justifies the congruences in the first column:

$$\kappa\backslash b \simeq \iota\cdot\iota\cdot a\cdot stop \qquad \mathbf{FIX}, \mathbf{KFAR}_1, \mathbf{EXP}_\backslash$$

$$\zeta\backslash b \simeq \iota\cdot stop \qquad \mathbf{FIX}, \mathbf{NEUT+}, \mathbf{KFAR}_1, \mathbf{EXP}_\backslash$$

$$\eta\backslash b \simeq \iota\cdot(\iota\cdot a\cdot stop + \iota\cdot\iota\cdot stop) \qquad \mathbf{FIX}, \mathbf{KFAR}_1, \mathbf{EXP}_\backslash, \zeta\backslash b \simeq \iota\cdot stop$$

Hence and from (1), since $\simeq$ is a congruence, relation (2-6) follows. □

Equation (2-6) is the inconsistency promised: it clearly cannot be valid in any meaningful observation equivalence $\simeq$. It is now important to analyze what happens when $\simeq$ is instantiated by one of the known observation congruences.

If $\simeq$ is weak bisimulation congruence $\simeq_b$, then $\mathbf{DISTR}^-$ simply does not hold, so no inconsistency arises.

Let instead $\simeq$ be an equivalence relation satisfying $\mathbf{DISTR}^-$ and $\mathbf{UNIQ}_\simeq$ and coarser than $\simeq_b$; then $\mathbf{EXP}_\backslash$, $\mathbf{NEUT+}$, $\mathbf{FIX}$, $\mathbf{KFAR}_1$ must also be valid (as they are for $\simeq_b$). Therefore, the only way to prevent the problematic Proposition 2.13 from holding is to admit that $\simeq$ is not a congruence, in that it is not preserved by the application of $\backslash b$, as assumed in deriving (2-6). This applies to ready simulation equivalence and all the above-mentioned weaker equivalences, down to failure equivalence, if they are defined in such a way to be coarser than $\simeq_b$.

Suppose, e.g., that $\simeq$ is defined as the failure equivalence:

$$p \simeq q \quad \text{iff} \quad \mathit{failures}(p) = \mathit{failures}(q) \text{ and } p \xrightarrow{\iota} \text{ iff } q \xrightarrow{\iota} \tag{2-7}$$

$$\mathit{failures}(p) = \{(s, X) \mid X \subseteq Act,\ \exists p' : p \stackrel{s}{\Longrightarrow} p' \ \&\ \forall x \in X : p' \not\stackrel{x}{\Longrightarrow}\} \tag{2-8}$$

(the second conjunct of (2-7) aims at ensuring substitutivity of summands). This failure equivalence can be proved to satisfy $\mathbf{UNIQ}_\simeq$ and is easily recognized to satisfy $\mathbf{DISTR}^-$ and be coarser than $\simeq_b$. Moreover it clearly does not satisfy equation (2-6), so it cannot be a congruence.

A way to regain a congruence is to give up fair abstraction and equate all process expressions that may perform an infinite sequence of ι-actions. E.g., if κ, η, ζ are defined as in Proposition 2.13, i.e.

$$\kappa := b\cdot\kappa + b\cdot a\cdot stop \qquad \eta := b\cdot\eta + b\cdot a\cdot stop + b\cdot\zeta \qquad \zeta := b\cdot\zeta$$

then:

$$\kappa\backslash b \simeq \eta\backslash b \simeq \zeta\backslash b \tag{2-9}$$

This treatment of divergence is said to be *catastrophic*, because it collapses all behaviours for which divergence cannot be excluded. It is clearly incompatible with the idea of fair abstraction, under which the expressions equated in (2-9) should be equivalent (cf. proof of Proposition 2.13) to:

$$\iota \cdot a \cdot stop \qquad \iota \cdot a \cdot stop + \iota \cdot stop \qquad \iota \cdot stop$$

respectively and therefore differ. Technically, the undesirable Proposition 2.13 cannot be proved because $\mathbf{KFAR}_1$ is now rejected; for the same reason, the catastrophic version of $\simeq$ cannot be coarser than bisimulation congruence $\simeq_b$.

Relation $\simeq$ can be (re)defined as a 'catastrophic' failure equivalence-congruence in at least two ways. The first is to keep (2-7), but redefine the *failures* function as in [Brookes, 1983a] (with a slight simplification justified if process constants are weakly guarded in all defining equations):

$$\begin{aligned} failures(p) \;=\; & \{(s,X) \mid X{\subseteq}Act,\ \exists p' \colon p \stackrel{s}{\Longrightarrow} p' \,\&\, \forall x{\in}X : p' \stackrel{x}{\not\Longrightarrow}\} \cup \\ & \{(st,X) \mid t{\in}Act^*, X{\subseteq}Act, \\ & \quad \exists p', p_1, p_2, \ldots \colon p \stackrel{s}{\Longrightarrow} p' \stackrel{\iota}{\longrightarrow} p_1 \stackrel{\iota}{\longrightarrow} p_2 \stackrel{\iota}{\longrightarrow} \ldots\} \end{aligned}$$

Another definition could be based on a compositional denotational semantic function $\mathcal{O}[\![\ ]\!]$ similar to that of Section 2.4.3 (but adapted for PL_+):

$$p \simeq q \quad \text{iff} \quad \mathcal{O}[\![p]\!] = \mathcal{O}[\![q]\!]$$

A Failure Semantics with Fair Abstraction?

Is there a way to integrate a failure-based semantics with fair abstraction? This would be an appealing achievement, for it would combine the advantages attributed to each approach in Sections 1.6.1 and 1.6.2.

A solution was proposed by Bergstra, Klop and Olderog in their paper *Failures without Chaos: A New Process Semantics for Fair Abstraction* [Bergstra, Klop, & Olderog, 1987]. Below, we shall argue that their results, albeit sound, are not

particularly useful in practice, for correctness verification. As discussed in Section 1.6.2, we envisage a thoroughly different solution, which will be the subject of Chapter 3.

The approach of [Bergstra, Klop, & Olderog, 1987] is based on replacing the **KFAR** rules with a weaker infinitary version:

KFAR$^-$ Let $p_n \simeq b_n \cdot p_{n+1} + q_n$, where $b_n \in B$, for all $n \geqslant 0$; assume $q_m \simeq b \cdot q$ for some $m \geqslant 0, b \in B$.

Then $p_0 \backslash B \simeq \iota \cdot \sum_{n \geqslant 0}(q_n \backslash B)$.

Unlike **KFAR$_1$**, **KFAR$^-$** does not allow the inconsistency represented by Proposition 2.13 to be derived. On the contrary, Bergstra, Klop and Olderog provide a failure congruence that satisfies **KFAR$^-$**, **UNIQ$_\simeq$**, (a stronger law than) **DISTR$^-$**, and (substituting $\simeq$ for $\sim$) **EXP$_\backslash$**, **FIX** and the laws for ι and $+$ from Section 2.3.3.

Their definition is as follows (we still denote this new congruence by $\simeq$ to avoid rewriting equational laws with yet another relation symbol):[4]

$$\begin{aligned} p \simeq q \quad \text{iff} \quad & \mathit{failures}_{\mathrm{BKO}}(p) = \mathit{failures}_{\mathrm{BKO}}(q),\ p \xrightarrow{\iota} \text{ iff } q \xrightarrow{\iota}, \\ & \text{and } \mathit{traces}(p) = \mathit{traces}(q) \\ \mathit{traces}(p) \quad = \quad & \{s \mid \exists p' : p \stackrel{s}{\Longrightarrow} p'\} \\ \mathit{failures}_{\mathrm{BKO}}(p) \quad = \quad & \{(s, X) \mid X \subseteq Act, \\ & \qquad \exists p' : p \stackrel{s}{\Longrightarrow} p',\ p' \not\xrightarrow{\iota},\ \forall x \in X : p' \not\stackrel{x}{\Longrightarrow}\} \end{aligned} \tag{2-10}$$

The need for $\mathit{traces}(\,)$ arises because, even if $s \in \mathit{traces}(p)$, it may happen that $(s, \varnothing) \notin \mathit{failures}_{\mathrm{BKO}}(p)$ (if $p \stackrel{s}{\Longrightarrow} p'$ implies $p' \xrightarrow{\iota}$).

The problem with this approach is, in our view, that the applicability of a rule like **KFAR$^-$** is rather limited: basically, it says that an ι-loop can be abstracted from, provided there is an ι-transition taking out of it. It is not difficult to think of practically relevant cases in which this is insufficient to prove correctness. Actually, in [Bergstra, Klop, & Olderog, 1987] an example communication system is presented with the opposite intent.

[4] The semantics of [Bergstra, Klop, & Olderog, 1987] is in fact slightly more complicated, in order to treat successful termination and sequential composition.

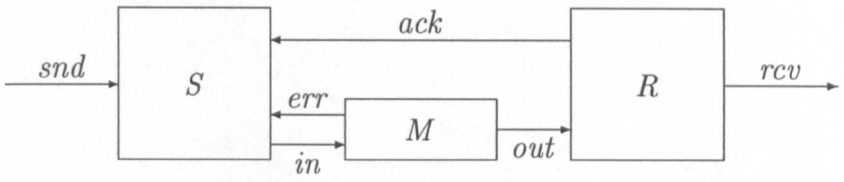


Figure 2.1: Bergstra, Klop and Olderog's example.

This example, depicted in Figure 2.1, is a toy communication system made up of a sender S and a receiver R that exchange messages, through a lossy medium M, and acknowledgements, directly, through a shared channel *ack*. The service offered by the system is a cycle in which a *snd* action is accepted by S, and subsequently a *rcv* action is issued by R. The protocol is based on the assumption that, if M accepts an action *in* from S, then it either notifies R with action *out* or reports an error action *err* back to S, which will then retry *in*. This assumption, which would seem to aim at avoiding ι-loops that $\mathbf{KFAR}^-$ cannot reduce, is rather unrealistic.

Indeed, consider the mechanism whereby M may realize at the transmitting end that delivery at the receiving end has failed: M can forward the data and start waiting for an acknowledgement (a new one, inside M); if ack transmission is totally reliable and timings are predictable, then a timeout expiry may tell the transmitting end for sure that data have not been delivered. But, apart from this unusual case, rather complex protocols would seem necessary, based on repeated back and forth communication within M; yet this is what S and R already carry out on top of M: considerable function duplication among adjacent layers would ensue, in contrast with the principles of sound protocol design [Tanenbaum, 1988]. In any case, the problem circumvented by the error report action would only have been moved to a lower level, namely the verification of M.

So let us consider instead a realistic modelling of the famous *alternating bit protocol* [Bartlett, Scantlebury, & Wilkinson, 1969], whereby the system of Figure 2.1 may provide a reliable service based on retransmission, without any need for the medium to possess the questionable error-reporting capability.

The service expected by the system is easily described by:

$$SE \;:=\; snd \cdot rcv \cdot SE$$

The service provided by the system can be specified as $SP_{000}(0,0,0)$, where $SP_{ijk}(l, m, n)$ is just a shorthand (not a constant) for the process expression:

$$(S_i(l):\{snd, in, ack\} \parallel M_j(m):\{in, out\} \parallel R_k(n):\{out, ack, rcv\}) \setminus \{in, out, ack\}$$

Sender, medium and receiver are respectively described by the indexed and parametric constants S, M and R.

The sender starts out as $S_0(0)$. Initially, $S_0(l)$ accepts a *snd* request from a user, tries to inform the receiver by passing bit l at *in* to the medium. and starts waiting for an acknowledgement. Waiting may be disrupted by an internal action, followed by a retry to send bit l at *in* (this models a timeout expiry, abstracting from timeout duration). If instead an ack carrying bit l arrives, all the above behaviour is repeated with bit l negated.

$$\begin{aligned} S_0(l) &:= snd \cdot S_1(l) \\ S_1(l) &:= in!l \cdot S_2(l) \\ S_2(l) &:= \iota \cdot S_1(l) + ack!l \cdot S_0(\neg l) \end{aligned}$$

The medium starts as $M_0(0)$ (the parameter of M_0 is in fact a dummy, introduced for uniformity). It behaves like a one-bit buffer that may internally decide to lose the yet undelivered bit m, as described by $M_1(m)$.

$$\begin{aligned} M_0(m) &:= in!0 \cdot M_1(0) + in!1 \cdot M_1(1) \\ M_1(m) &:= out!m \cdot M_0(0) + \iota \cdot M_0(0) \end{aligned}$$

The receiver starts as $R_0(0)$. Initially, $R_0(n)$ repeatedly acknowledges the previous reception of bit $\neg n$ and waits for bit n to come out from the medium, at *out*. If this happens, the user of R is informed with a *rcv*, and the above behaviour is repeated with bit n negated.

$$\begin{aligned} R_0(n) &:= ack!(\neg n) \cdot R_0(n) + out!n \cdot R_1(n) \\ R_1(n) &:= rcv \cdot R_0(\neg n) \end{aligned}$$

Proving by hand that SE is bisimulation congruent to $SP_{000}(0,0,0)$ is slightly tedious, because the latter expression corresponds to about thirty LTS states. A

proof has been more easily obtained with our mechanical tool described in [Pappalardo, 1987].

In contrast, $SE \simeq SP_{000}(0,0,0)$ does not hold if $\simeq$ is the failure equivalence of [Bergstra, Klop, & Olderog, 1987], with $failures_{\mathrm{BKO}}()$ defined as in (2-10). Clearly, the failure $(\langle snd \rangle, \{snd\})$ lies in $failures_{\mathrm{BKO}}(SE)$, but it will now be shown:

Proposition 2.14 $(\langle snd \rangle, \{snd\}) \notin failures_{\mathrm{BKO}}(SP_{000}(0,0,0))$, for $failures_{\mathrm{BKO}}()$ as in (2-10).

Proof. It is enough to show:

(1) $SP_{000}(0,0,0) \stackrel{snd}{\Longrightarrow} p$ and $p \stackrel{snd}{\not\Longrightarrow}$ imply $p \stackrel{\iota}{\longrightarrow}$

To see this, note first that $SP_{000}(0,0,0) \stackrel{s}{\Longrightarrow} p$ entails that p is $SP_{ijk}(l,m,n)$ for proper indices and parameters (this claim is easily proved by induction). Moreover, $p \stackrel{snd}{\not\Longrightarrow}$ implies that i is either 1 or 2. If $i = 2$ or $j = 1$, the proposition follows from $S_2(l) \stackrel{\iota}{\longrightarrow}$ or $M_1(m) \stackrel{\iota}{\longrightarrow}$, respectively. If $i = 1$ and $j = 0$, it follows from:

$$S_1(l){:}\{snd, in, ack\} \| M_0(m){:}\{in, out\} \stackrel{\iota}{\longrightarrow} S_2(l){:}\{snd, in, ack\} \| M_1(l){:}\{in, out\}. \quad \square$$

In a more realistic specification, acknowledgements would be dealt with just like messages and media capacity would be unbounded. However, the previous proposition would still hold because of the persistent ι-actions arising.

2.6.3 Extended Failures for Fair Abstraction

The previous discussion should have convinced the reader that no failure semantics is really compatible with fair abstraction. It is therefore necessary to settle for a less ambitious goal: to find a process semantics that (1) enables fair abstraction, and (2) is based on observations such that any failure of a process can be extracted from (at least) one of them. A good candidate observation is an *extended failure*, i.e. a member of the set:

$$xfailures(p) \;=\; \{(s,X) \mid X \subseteq Act^+,\ \exists p' : p \stackrel{s}{\Longrightarrow} p' \ \&\ \forall x \in X : p' \stackrel{x}{\not\Longrightarrow}\}$$

Thus, in an extended failure (s,X), the refusal X also records non-empty traces that can be refused by a process after performing X. Of course, an ordinary, 'action-refusing' failure is also an extended failure, which easily fulfils the requirement (2)

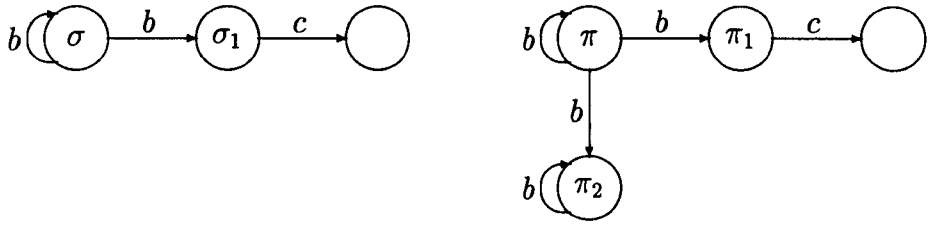


Figure 2.2: σ and π are failure- but not extended failure-equivalent.

identified above. Henceforth, we will mainly rely on the context to make clear whether a failure is an extended or an ordinary one.

The other requirement noted, viz. (1), that extended failure semantics should permit fair abstraction, is also satisfied because bisimulation congruence $\simeq_b$ is easily proved to be a finer relation than the new $\simeq$ adopted, defined this time as an extended failure equivalence:

$$p \simeq q \quad \text{iff} \quad \mathit{xfailures}(p) = \mathit{xfailures}(q) \text{ and } p \xrightarrow{\iota} \text{ iff } q \xrightarrow{\iota}$$

Thus, any fair abstraction identifications permitted by $\simeq_b$ (including those induced by the **KFAR** rules) is also valid under this $\simeq$. Further, for a LTS without ι, $\simeq$ is stricter than failure equivalence. Thus, since $\simeq$ lies between two well-established observation equivalences, it should also be intuitively acceptable. More precisely, in the 'equivalence lattice' of [v. Glabbeek, 1990], $\simeq$ would be inserted out of the ready-simulation sublattice, above ordinary failure equivalence, and below Milner's $\approx_2$ equivalence.

Of course, it must also be ensured that $\simeq$ does not suffer from inconsistencies like that represented by Proposition 2.13. Evidence that this should not be the case is provided by the remark that this $\simeq$ does not satisfy **DISTR**$^-$, so one of the hypotheses of Proposition 2.13 does not hold. Informally, it should be quite convincing to consider the LTS of Figure 2.2 and observe that, unlike the ordinary failure equivalence (2-7), the new $\simeq$ does distinguish the states σ and π both before and after b is hidden. Indeed,

$$(b, \{\langle b, c\rangle, \langle c\rangle\}) \in \mathit{xfailures}(\pi) \qquad \text{by } \pi \xRightarrow{b} \pi_2,\ \pi_2 \not\xRightarrow{bc},\ \pi_2 \not\xRightarrow{c}$$

$$(b, \{\langle b, c\rangle, \langle c\rangle\}) \notin \mathit{xfailures}(\sigma) \qquad \text{by } \sigma \xRightarrow{bc},\ \sigma_1 \xRightarrow{c}$$

More formally, it should be proved that $\simeq$ is a congruence. The critical case, hiding, is dealt with by proving that, for fixed B, $xfailures(p\backslash B)$ depends only on $xfailures(p)$. A proof can be given along the lines of Proposition 3.99. However, we shall not pursue this approach any further.

Extended failures will instead be employed, in Chapter 3, to give an observation set-based, compositional denotational semantics. The advantage of this approach is to afford a smooth treatment of fixpoints.

Chapter 3

Extended Failures

3.1 About This Chapter

The adoption of a denotational process semantics based on failures that record refused traces has been motivated at length, in Sections 1.6.2 and 2.6.2, with the goal of combining fair abstraction, the ease of failure-oriented specification, and fixpoint-related techniques.

In the following Section 3.2, a process domain of sets of such failures is presented, and provided with sufficient structure to afford a smooth definition of fixpoints and a fixpoint induction rule. These results cannot be obtained as customary [Brookes, Hoare, & Roscoe, 1984], by making the process domain a complete partial order (cpo), without also jeopardizing fair abstraction; the alternative approach devised here is to identify the class—large enough for applications—of *trace-based* process functions, and show that they possess a particular fixpoint, compatible with fair abstraction. It should also be remarked that the fixpoint rule provided is more manageable than that of [Brookes, Hoare, & Roscoe, 1984] because it suggests a possible induction basis which can be proved without further recourse to fixpoint induction. The search for practical fixpoint rules also brings us to discovering an interesting alternative process order, and the useful notion of *specification-oriented* process predicate.

In Section 3.3, the process language $\mathrm{PL}^{\oplus}_{\uplus}$ is given a compositional denotational semantics as described in Section 2.4.3, by choosing as semantic domain that of (ex-

tended failure) processes and defining, for each language operator, an appropriate process function.

In our view, a rather pleasing conclusion about the formalism of this chapter is that, while the proposed extension of conventional failure-based approaches appears to be natural and useful (cf. Chapter 4), it is by no means straightforward. Indeed, throughout the treatment, several novel concepts arise and some results turn out to be non-trivial to derive, like e.g. the relationship between processes and LTSs (cf. Proposition 3.11), fixpoint theory and fixpoint induction rules (revolving around the notions of trace-based process function and specification-oriented predicate), continuity of the action sequence operation, and associativity of hiding combined with parallel composition.

3.2 An Extended Failure Model

3.2.1 The Domain of Processes

Preliminaries

As amply discussed earlier, in the approach under study, a process is essentially identified with a set of failures recording also refused traces. However, we find it more convenient here to represent a process, equivalently, with two structures. The first is a non-empty, prefix-closed set of traces:

Definition 3.1 A *trace* is a sequence of actions, i.e. a member of Act^*.

A *tree* is a non-empty, prefix-closed set of traces. The set of trees is denoted by *Tree* and ranged over by $T, U, V, \ldots$. □

The other structure is a function mapping a trace onto a set of (extended) refusals.

Definition 3.2 A *refusal function* r maps a trace s to a set $r(s)$ satisfying the following constraints:

1. $\varnothing \subset r(s) \subseteq \mathsf{p}Act^+$;

2. refusal subset-closure: if $X \in r(s)$ and $Y \subseteq X$, then $Y \in r(s)$;

3. refusal suffix-closure: If $X \in r(s)$ and $t \in X$, then $X \cup \{tw\} \in r(s)$ for all $w \in Act^+$.

The set of refusal functions is denoted by *RefFun* and ranged over by r. A refusal function image is a set of trace sets called *refusals* and ranged over by $X, Y, Z \ldots$. □

Note that, since every value $r(s)$ of a refusal function r is a non-empty, subset-closed set, it must always contain $\varnothing$ (which is a member of $\mathsf{p}Act^+$), i.e. $\varnothing \in r(s)$ for all s.

Trees and refusal functions can be paired to obtain a semi-process:

Definition 3.3 A *semi-process* is a pair $(T, r) \in \mathit{Tree} \times \mathit{RefFun}$ such that r has domain T. The set of semi-processes is denoted $\bar{\mathcal{P}}\mathit{roc}$ and ranged over by $P, Q, R, \ldots$.

For $P = (T, r)$, $P \in \bar{\mathcal{P}}\mathit{roc}$, define τP (the traces of P) and ρP (the refusal function of P) as:

$$\tau P = T \qquad\qquad \rho P = r$$

The τ, ρ operators will in fact be used to access the elements of any pair (T, r) where T is a trace set and r maps a trace to a set of trace sets.

In order to spare parentheses, with a device typical of functional programming, the application of function ρP to trace s will be denoted $\rho P(s)$ rather than $(\rho P)(s)$. □

Some licence will be taken as to the domain of the refusal function in a semi-process:

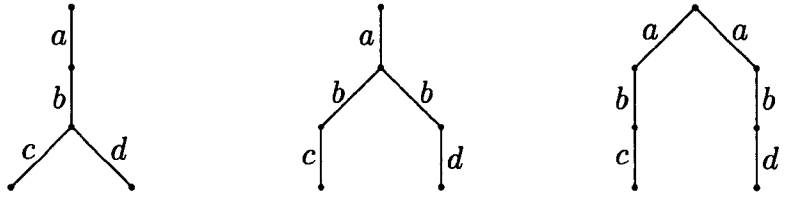
Remark 3.4 The condition that, for a semi-process P, τP and the domain of ρP should coincide is often implied.

Thus (T, r), where $\mathrm{dom}\, r \supseteq T$, should be viewed as a shorthand for the semi-process $(T, \mathrm{restr}_T\, r)$ ($\mathrm{restr}_T\, r$ denotes the *restriction* of function r to domain T).

Likewise, in defining a semi-process P through τP and $\rho P(s)$, the condition $s \in \tau P$ is normally omitted for brevity. □

Examples

In order to illustrate the previous concepts, consider three systems behaving as described by the following trees:



The semi-processes describing these behaviours will be denoted respectively by (T, r_1), (T, r_2), (T, r_3). All of them share the tree

$$T = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle, \langle a, b, d\rangle\}$$

As for refusal functions, $r_1(s)$ is defined to contain X iff the first system refuses every trace in X from at least a state reached after s:

$$\begin{aligned}
r_1(\langle\rangle) &= \{X \in \mathsf{p}Act^+ \mid X \cap T = \varnothing\} \\
r_1(\langle a\rangle) &= \{X \in \mathsf{p}Act^+ \mid X \cap \{\langle b\rangle, \langle b, c\rangle, \langle b, d\rangle\} = \varnothing\} \\
r_1(\langle a, b\rangle) &= \{X \in \mathsf{p}Act^+ \mid X \cap \{\langle c\rangle, \langle d\rangle\} = \varnothing\} \\
r_1(\langle a, b, c\rangle) &= \mathsf{p}Act^+ \\
r_1(\langle a, b, d\rangle) &= \mathsf{p}Act^+
\end{aligned}$$

Likewise, $r_2(s)$ is defined to contain X iff the second system refuses every trace in X from at least a state reached after s. So $r_2(s)$ agrees with $r_1(s)$ for $s \neq \langle a, b\rangle$ and:

$$r_2(\langle a, b\rangle) = \{X \in \mathsf{p}Act^+ \mid X \cap \{\langle c\rangle\} = \varnothing \vee X \cap \{\langle d\rangle\} = \varnothing\}$$

Finally, $r_3(s)$ is defined to contain X iff the third system refuses every trace in X from at least a state reached after s. So $r_3(s)$ agrees with $r_2(s)$ for $s \neq \langle a\rangle$ and:

$$r_3(\langle a\rangle) = \{X \in \mathsf{p}Act^+ \mid X \cap \{\langle b\rangle, \langle b, c\rangle\} = \varnothing \vee X \cap \{\langle b\rangle, \langle b, d\rangle\} = \varnothing\}$$

It is worth noting that the second and third behaviour are indistinguishable in the standard failure model; yet it is not difficult to devise simple experiments that can distinguish between them: the crucial difference suggested by our model is that, after doing an a, the second system can never refuse to perform $\langle b, c\rangle$ and $\langle b, d\rangle$, whereas the second will certainly refuse either $\langle b, c\rangle$ or $\langle b, d\rangle$.

As a final example semi-process, suppose $b \in Act$ and consider (T_0, r_0) with

$$\begin{aligned} T_0 &= \{\langle\rangle, \langle a\rangle\} \\ r_0(\langle\rangle) &= \{\varnothing\} \\ r_0(\langle a\rangle) &= \mathsf{p}Act^+ \end{aligned}$$

Now r_0 is indeed a refusal function, but seems to be inconsistent with T_0, if (T_0, r_0) is to represent the behaviour of a system. For the assumed $r_0(\langle\rangle)$ seems to indicate that the system can silently move to a state whence no trace is refused, but then we would expect $\langle b\rangle$ to be in T.

Processes

The first three example semi-processes above seem to represent correctly the respective intended behaviour, but the fourth example shows that not any tree T and refusal function r, once paired, do actually correspond to real process behaviour. For this purpose, T and r must satisfy a consistency requirement; this may be explained by recalling that, operationally, a LTS state σ refuses X after a trace s if, for some σ', $\sigma \stackrel{s}{\Longrightarrow} \sigma'$ and

$$\text{for all } x \in X,\ \sigma' \stackrel{x}{\not\Longrightarrow}. \tag{3-1}$$

Therefore, if σ refuses X but not $X \cup \{t\}$ after s, then $\sigma' \stackrel{t}{\Longrightarrow} \sigma''$ must hold. Two consequences follow. (1) $\sigma \stackrel{st}{\Longrightarrow} \sigma''$ (st is a trace of σ). (2) For all $tw \in X$, $\sigma'' \stackrel{w}{\not\Longrightarrow}$ must hold or $\sigma' \stackrel{tw}{\Longrightarrow}$ would follow, contradicting (3-1) above; in other words, σ refuses any trace like w after st. Thus:

Definition 3.5 A *process* is a semi-process $(T, r) \in \bar{}Proc$ such that T and r are *consistent* in the sense that, for all $s \in T$: if $X \in r(s)$ and $X \cup \{t\} \notin r(s)$, then: (1) $st \in T$, and (2) $X \div t \in r(st)$, where:

$$X \div t = \{w \mid tw \in X\}$$

$X \div t$ is read 'X after t'.

The set of processes is denoted $Proc$ and ranged over by $P, Q, R, \ldots$. □

Note that, in the consistency requirement, since $t \notin X$, $\langle\rangle \notin X \div t$, as necessary for $X \div t \in r(st)$. Moreover, the consistency requirement also (vacuously) holds for $t = \langle\rangle$ because $X \div \langle\rangle = X$.

The following consequence of trace-refusal consistency resembles more closely consistency for ordinary failures (cf. Definition 1.1):

Proposition 3.6 Let $P \in Proc$, and $s \in \tau P$, $X \in \mathfrak{p}Act^+$. Then $sX \cap \tau P = \varnothing$ implies $X \in \rho P(s)$.

Proof. By induction on the size of X. Let $|X| = n+1$, $X = X' \cup \{t\}$, $|X'| = n$. Then $X' \in \rho P(s)$ by induction hypothesis and $X' \cup \{t\} \in \rho P(s)$ or, by trace-refusal consistency, $st \in \tau P$, contradicting $sX \cap \tau P = \varnothing$. □

Sufficiency of Process Properties

Trace-refusal consistency has been informally seen to be a necessary condition for a failure set to be generable by a LTS. The same can be easily shown for the non-emptiness and closure process properties. It is natural to ask, besides, if these properties also suffice to characterize process behaviour, or if more are needed instead. A way to answer this question affirmatively is to show that every failure set enjoying the process properties can be generated by a suitable LTS.

For this purpose, a LTS having processes as states will be defined, beginning from a $\Longrightarrow$ relation over *Proc*.

Definition 3.7 For $P, Q \in Proc$, $P \overset{s}{\Longrightarrow} Q$ holds iff:

$$\forall t \in \tau Q,\ X \in \rho Q(t) \colon st \in \tau P \wedge X \in \rho P(st). \qquad \square$$

Constructing a $\longrightarrow$ relation for this LTS is not difficult and will be omitted; it will just be observed here that the construction depends on the following 'reflexive-transitive' closure property of $\Longrightarrow$:

Proposition 3.8 For all $P, Q \in Proc$ and $s, t \in Act^*$:

1. $P \overset{\langle\rangle}{\Longrightarrow} P$;
2. $P \overset{st}{\Longrightarrow} Q$ iff $P \overset{s}{\Longrightarrow} R$ and $R \overset{t}{\Longrightarrow} Q$ for some $R \in Proc$.

Proof. Reflexivity and the transitivity if-part are immediate to prove. For the only-if part, the R sought is the process $P \div s$ introduced below, within Proposition 3.9. □

Proposition 3.9 For $P \in Proc$ such that $s \in \tau P$, let

$$\tau(P \div s) = (\tau P) \div s \qquad\qquad \rho(P \div s)(t) = \rho P(st)$$

Then (1) $P \div s$ ('P after s') is a process, and (2) $P \stackrel{s}{\Longrightarrow} P \div s$.

Proof. Proving (2) is immediate. Below we consider (1).

$\tau(P \div s)$ is a tree: $\langle\rangle \in \tau(P \div s)$ by $s \in \tau P$, and $tu \in \tau(P \div s)$ implies $stu \in \tau P$, whence $st \in \tau P$ and $t \in \tau(P \div s)$.

Assume now $t \in \tau(P \div s)$ and $X \in \rho(P \div s)(t)$. Then $st \in \tau P$ and $X \in \rho P(st)$. It is easy to infer subset-, suffix-closure and trace-refusal consistency for $P \div s$ from the corresponding property of P. E.g. for the latter: if $X \cup \{w\} \notin \rho(P \div s)(t)$, then $X \cup \{w\} \notin \rho P(st)$, so trace-refusal consistency of P implies $stw \in \tau P$, whence $tw \in \tau(P \div s)$, and $X \div w \in \rho P(st)$, whence $X \div w \in \rho(P \div s)(t)$. □

We now get back to the main issue, i.e. showing that, in the LTS introduced, transition-based traces and refusals of P coincide with τP and ρP respectively. Formally:

Theorem 3.10 Let $P \in Proc$. Then $s \in \tau P$ and $X \in \rho P(s)$ iff, for some $Q \in Proc$, $P \stackrel{s}{\Longrightarrow} Q$ and $Q \stackrel{x}{\not\Longrightarrow}$ for all $x \in X$,

Proof. The if-part follows, by definition of $P \stackrel{s}{\Longrightarrow} Q$, from $X \in \rho Q(\langle\rangle)$. This, in turn, follows by Proposition 3.6 because $x \in X$ implies $x \notin \tau Q$ (or, by Proposition 3.9, $Q \stackrel{x}{\Longrightarrow} Q \div x$, against the hypothesis).

For the only-if part, note that $X \in \rho(P \div s)(\langle\rangle)$. By the following Proposition 3.11, this ensures that there exists Q such that $P \div s \stackrel{\langle\rangle}{\Longrightarrow} Q$ and $Q \stackrel{x}{\not\Longrightarrow}$ for all $x \in X$. Hence, by Propositions 3.9 and 3.8(2), $P \stackrel{s}{\Longrightarrow} Q$ and the theorem follows. □

The proof obligation left from Theorem 3.10 is rather hard:

Proposition 3.11 Let $P \in Proc$ and $X \in \rho P(\langle\rangle)$. Define:

$$\begin{aligned} \dot{X} &= \{xw \in Act^+ \mid x \in X\} \quad \text{(suffix-closure of } X\text{)} \\ \tau P_X &= \{s \mid s \notin \dot{X},\ \forall u, Z \in \mathsf{p}Act^+ : u \leqslant s \wedge uZ \subseteq \dot{X} \Rightarrow u \in \tau P \wedge Z \in \rho P(u)\} \\ \rho P_X(s) &= \{Y \in \mathsf{p}Act^+ \mid sY \cap \tau P_X = \varnothing\} \end{aligned}$$

Then $P_X \in Proc$, $P_X \stackrel{x}{\not\Longrightarrow}$ for all $x \in X$, and $P \stackrel{\langle\rangle}{\Longrightarrow} P_X$.

Proof. To see that $P_X \in Proc$, we invoke Proposition 3.15 and show here that τP_X is a tree. Indeed, $\langle\rangle \in \tau P_X$ holds because $\langle\rangle \notin \dot{X}$ and $Z \subseteq \dot{X}$, Z finite, implies $Z \in \rho P(\langle\rangle)$; the latter fact follows from $Z \cap X \in \rho P(\langle\rangle)$, by suffix-closure of $\rho P(\langle\rangle)$ and induction on the size of $Z - X$. As for prefix-closure, $st \in \tau P_X$ easily implies $s \in \tau P_X$ ($s \notin \dot{X}$, or $st \in \dot{X}$ would follow).

The second claim is easy. If, by contradiction, $x \in X$ and $P_X \stackrel{x}{\Longrightarrow}$ then, by Theorem 3.10 (if-part, already proved), $x \in \tau P_X$, so $x \notin \dot{X}$, a contradiction with $x \in \dot{X}$.

For the third claim ($P \stackrel{\langle\rangle}{\Longrightarrow} P_X$), assume $s \in \tau P_X$ and $Y \in \rho P_X(s)$, i.e.:

(1) $s \in \tau P_X$ and $sY \cap \tau P_X = \varnothing$.

Now, that $s \in \tau P$ is obvious from the definition of τP_X, so we must only show:

(2) $Y \in \rho P(s)$.

Let $Y_1 = \{y \in Y \mid sy \in \dot{X}\}$ and $Y_2 = \{y \in Y \mid sy \notin \dot{X}\}$. Suppose $y \in Y_2$. Then from (1) it follows $sy \notin \tau P_X$; so (since $sy \notin \dot{X}$) there are u and $Z \in \mathsf{p}Act^+$ such that $u \leqslant sy$, $uZ \subseteq \dot{X}$, and $u \in \tau P \wedge Z \in \rho P(u)$ is false. This would be impossible for $u \leqslant s$ (because $s \in \tau P_X$), so $u = sw$ for some w. Thus it has been proved that, for all y, there are $w_y \in Act^*$ and $Z_y \in \mathsf{p}Act^+$ such that:

(3) If $y \in Y_2$, then $w_y \leqslant y$, $sw_yZ_y \subseteq \dot{X}$, and $sw_y \notin \tau P$ or $Z_y \notin \rho P(sw_y)$

Let

$$W = \bigcup_{y \in Y_2} w_y Z_y$$

and recall $Y = Y_1 \cup Y_2$. The desired result (2) will now be derived by proving $Y_1 \cup W \cup Y_2 \in \rho P(s)$, by induction on the size of Y_2.

Basis. By (3), $s(Y_1 \cup W) \subseteq \dot{X}$. Hence, by $s \in \tau P_X$ (from (1)), and the way τP_X is defined, $Y_1 \cup W \in \rho P(s)$ follows.

Step. Let $|Y_2| = n+1$, $Y_2 = Y_2' \cup \{y\}$, $|Y_2'| = n$ and

(4) $Y_1 \cup W \cup Y_2' \in \rho P(s)$ (induction hypothesis).

We shall now prove:

(5) $sw_y \notin \tau P$ or $(Y_1 \cup W \cup Y_2') \div w_y \notin \rho P(sw_y)$

Statement (5) follows from (3) and $Z_y \subseteq (Y_1 \cup W \cup Y_2') \div w_y$; to see this, assume $z \in Z_y$, then $w_y z \in W \subseteq (Y_1 \cup W \cup Y_2')$, whence $z \in (Y_1 \cup W \cup Y_2') \div w_y$. Now, from (4) and (5), by consistency of τP and ρP:

$$(Y_1 \cup W \cup Y_2' \cup \{w_y\}) \in \rho P(s)$$

whence, by suffix-closure of $\rho P(s)$ and $w_y \leqslant y$ (by (3)):

$$(Y_1 \cup W \cup Y_2' \cup \{y\}) = (Y_1 \cup W \cup Y_2) \in \rho P(s) \qquad \square$$

3.2.2 Deterministic Processes

In this section we shall overload the ρ operator by applying it to a tree U; the result will be defined to be a refusal function ρU, relating s to the refusal set X iff in the tree U every trace $x \in X$ is impossible after s. Formally:

Definition 3.12 Let $U \in \mathit{Tree}$. The function ρU defined, for all trace $s \in \mathit{Act}^*$, by:

$$\rho U(s) = \{X \in \mathsf{p}\mathit{Act}^+ \mid sX \cap U = \varnothing\}$$

is easily shown to be a refusal function. $\square$

Note in particular that $\varnothing \in \rho U(s)$ for all s, because sX is defined to be $\varnothing$ for $X = \varnothing$.

Tree based refusal functions interact with set operations in a natural way:

Proposition 3.13 Let $T, V \in \mathit{Tree}$ and $\mathcal{V} \subseteq \mathit{Tree}$, $\mathcal{V} \neq \varnothing$. Then, for all trace s:

1. $\rho T(s) \supseteq \rho V(s)$ if $T \subseteq V$;

2. $\rho(\bigcup \mathcal{V})(s) = \bigcap\{\rho V(s) \mid V \in \mathcal{V}\}, \quad \rho(\bigcap \mathcal{V})(s) = \bigcup\{\rho V(s) \mid V \in \mathcal{V}\}.$

Proof. Let X be a finite subset of Act^+.

1. $X \in \rho V(s)$ means $sX \cap V = \varnothing$, whence also $sX \cap T = \varnothing$, i.e. $X \in \rho T(s)$.

2. $X \in \rho(\bigcup \mathcal{V})(s)$ holds iff $sX \cap \bigcup \mathcal{V} = \varnothing$, which holds iff $sX \cap V = \varnothing$ for all $V \in \mathcal{V}$, i.e. $X \in \rho V(s)$ for all $V \in \mathcal{V}$. The other equality under (2) is proved similarly. □

A deterministic process is one that refuses only impossible traces. Formally:

Definition 3.14 A process P is *deterministic* if $\rho P = \rho(\tau P)$. □

Thus, there is at most one deterministic process with a given trace set, and indeed there is exactly one:

Proposition 3.15 Let T be a tree. Then $(T, \rho T)$ is a deterministic process, denoted by $\mathrm{det}(T)$.

Proof. We need only show that T and ρT are consistent. Let $s \in T$ and $X \in \rho T(s)$, i.e. $sX \cap T = \varnothing$. If $s(X \cup \{t\}) \cap T \neq \varnothing$, then $st \in T$. Moreover $(st)(X \div t) = s(t(X \div t)) \subseteq sX$, so $(st)(X \div t) \cap T = \varnothing$, i.e. $X \div t \in \rho T(st)$. □

Conversely, a process is non-deterministic if it refuses more traces than those impossible.

A notable deterministic process is $STOP$, which has as trace set $\bullet$, the tree containing only the empty trace:

Definition 3.16 The *empty* tree $\bullet$ and the *deadlock* process $STOP$ are defined:

$$\bullet = \{\langle\rangle\} \qquad\qquad STOP = \mathrm{det}(\bullet)$$

Note that, for every trace s:

$$\rho \bullet (s) = \mathrm{p}Act^+$$

□

3.2.3 A Partial Order for Processes

In this chapter, a process P is considered to be 'less than' another process Q ($P \sqsubseteq Q$) if P may perform a trace s only if Q may too, and P is less non-deterministic than Q. This means that, after any trace common to P and Q, Q may refuse any trace set that P refuses, and possibly more. This ordering is the reverse of that defined in [Brookes, Hoare, & Roscoe, 1984] for ordinary failures.

Definition 3.17 For all $P, Q \in Proc$, the relation $P \sqsubseteq Q$ holds if:

1. $\tau P \subseteq \tau Q$
2. $\rho P(s) \subseteq \rho Q(s)$ for all $s \in \tau P$

$P \sqsubseteq Q$ is also written $Q \sqsupseteq P$. □

It is immediate to show that:

Proposition 3.18 $(Proc, \sqsubseteq)$ is a reflexive partial order. □

Proposition 3.19 For all $P \in Proc$, $P \sqsupseteq \det(\tau P)$.

Proof. Clearly, $\tau \det(\tau P) = \tau P$. For refusal functions, assume $s \in \tau P$, and $X \in \rho(\det(\tau P))(s)$. Then $sX \cap \tau P = \varnothing$, and, by Proposition 3.6, the claim follows. □

Let Dom be a non-empty set and $\leqslant$ a partial order relation over Dom. It is convenient to introduce the well-known notion of least upper bound for the generic partial order $(Dom, \leqslant)$.

Definition 3.20 Let $(Dom, \leqslant)$ be a partial order and $\mathcal{D}$ a non-empty subset of Dom. A member $e \in Dom$ is said to be an *upper bound* of $\mathcal{D}$ if $d \leqslant e$ for all $d \in \mathcal{D}$; furthermore, e is said to be a *least upper bound* (lub) of $\mathcal{D}$ if, for all upper bounds e' of $\mathcal{D}$, $e \leqslant e'$. It is easy to prove that $\mathcal{D}$ can have at most one lub, which is denoted $\bigsqcup \mathcal{D}$. □

The notion of chain is also best defined for the generic partial order $(Dom, \leqslant)$:

Definition 3.21 A *chain* is a non-empty subset $\mathcal{D}$ of Dom such that, if $d, e \in \mathcal{D}$, then either $d \sqsubseteq e$ or $e \sqsubseteq d$. □

The above concepts apply in particular to the partial order $(Proc, \sqsubseteq)$.

If $\mathcal{P}$ is a non-empty set of processes, $\biguplus\mathcal{P}$ is intended to represent the process that may nondeterministically choose to behave as any $P \in \mathcal{P}$ (for a justification, cf. the transitional semantics in Section 2.4 and Property 3.84 for the related operators $\uplus$, $\biguplus_\Lambda$).

Definition 3.22 Let $\mathcal{P} \subseteq Proc$, $\mathcal{P} \neq \varnothing$. Define $\biguplus\mathcal{P}$ as the pair:

$$\tau\big(\biguplus\mathcal{P}\big) = \bigcup\{\tau P \mid P \in \mathcal{P}\} \qquad \rho\big(\biguplus\mathcal{P}\big)(s) = \bigcup\{\rho P(s) \mid P \in \mathcal{P},\ s \in \tau P\} \quad \square$$

$\biguplus\mathcal{P}$ is indeed a process:

Proposition 3.23 If $\mathcal{P} \subseteq Proc$, $\mathcal{P} \neq \varnothing$, then $\biguplus\mathcal{P} \in Proc$.

Proof. $\tau\biguplus\mathcal{P}$ is easily proved a tree using just the fact that, for each $P \in \mathcal{P}$, τP is a tree.

To show that $\rho(\biguplus\mathcal{P})$ is a refusal function, let $s \in \tau\biguplus\mathcal{P}$. Then $\rho(\biguplus\mathcal{P})(s)$ is non-empty, being the union of a non-empty set of non-empty sets. Further, assume $X \in \rho(\biguplus\mathcal{P})(s)$; then there is $P \in \mathcal{P}$ such that $s \in \tau P$ and $X \in \rho P(s)$. If $Y \subseteq X$, then $Y \in \rho P(s)$ by the subset-closure property of $\rho P(s)$, whence also $Y \in \rho(\biguplus\mathcal{P})(s)$. If $t \in X$ and w is a trace, then $X \cup \{tw\} \in \rho P(s)$ by the suffix-closure property of $\rho P(s)$, whence also $X \cup \{tw\} \in \rho(\biguplus\mathcal{P})(s)$.

Finally, to see that traces and refusals of $\biguplus\mathcal{P}$ are consistent, suppose that $X \cup \{t\} \notin \rho(\biguplus\mathcal{P})(s)$. Then $X \cup \{t\} \notin \rho P(s)$, whence $st \in \tau P$ and $X \div t \in \rho P(st)$ (by consistency of τP, ρP), so that $st \in \tau\biguplus\mathcal{P}$ and $X \div t \in \rho\biguplus\mathcal{P}(st)$. $\square$

A fundamental, though easy to prove, result is:

Theorem 3.24 For every non-empty process set $\mathcal{P}$, the lub $\bigsqcup\mathcal{P}$ exists and is $\biguplus\mathcal{P}$. $\square$

Compared to conventional *complete* partial orders, $(Proc, \sqsubseteq)$ lacks a bottom $\bot$ (such that $\bot \sqsubseteq P$ for all P), but enjoys the property that every non-empty set, not just a chain, has a lub. Of course, the partial order introduced has a top $\top$ (such that $P \sqsubseteq \top$ for all P), defined by:

$$\tau\top = Act^* \qquad\qquad \rho\top(s) = \mathsf{p}Act^+$$

$\top$ is obviously a process.

3.2.4 Process Tuples

In the following, process tuples (over an arbitrary index set Λ) are widely employed. For this purpose, the notation of Section 2.2 needs to be completed thus:

Definition 3.25 Let $x_\Lambda \in D^\Lambda$, $y_\Lambda \in E^\Lambda$. Then $x_\Lambda \times y_\Lambda$ (*cross-product* of x_Λ and y_Λ) is defined as the tuple $\langle (x_\lambda, y_\lambda) \mid \lambda \in \Lambda \rangle$ of $(D \times E)^\Lambda$. □

Some process-related notions will now be extended to process tuples. To begin with, the operators τ, ρ are component-wise extended to tuples.

Definition 3.26 For $P_\Lambda \in Proc^\Lambda$, let

$$\tau P_\Lambda = \langle \tau P_\lambda \mid \lambda \in \Lambda \rangle \qquad \rho P_\Lambda = \langle \rho P_\lambda \mid \lambda \in \Lambda \rangle$$ □

Some easy-to-derive relations between τ, ρ and process tuples follow.

Proposition 3.27 Let $P_\Lambda \in Proc^\Lambda$, $T_\Lambda \in Tree^\Lambda$, $r_\Lambda \in RefFun^\Lambda$. Then:

1. $P_\Lambda = \tau P_\Lambda \times \rho P_\Lambda$.
2. $T_\Lambda \times r_\Lambda \in \overline{Proc}^\Lambda$ provided $(T_\lambda, r_\lambda) \in \overline{Proc}$ for every $\lambda \in \Lambda$;
3. $T_\Lambda \times r_\Lambda \in Proc^\Lambda$ provided $(T_\lambda, r_\lambda) \in Proc$ for every $\lambda \in \Lambda$;
4. $\tau(T_\Lambda \times r_\Lambda) = T_\Lambda$, $\rho(T_\Lambda \times r_\Lambda) = r_\Lambda$. □

Determinism and tree-defined refusal functions generalize thus:

Definition 3.28

1. Let $U_\Lambda \in Tree^\Lambda$. Then $\rho U_\Lambda \in RefFun^\Lambda$ is defined to be $\langle \rho U_\lambda \mid \lambda \in \Lambda \rangle$.
2. A process tuple $P_\Lambda \in Proc^\Lambda$ is deterministic if, for all $\lambda \in \Lambda$, P_λ is deterministic or, equivalently, if $\rho P_\Lambda = \rho(\tau P_\Lambda)$.
3. If $T_\Lambda \in Tree^\Lambda$, $\det(T_\Lambda)$ is defined to be the deterministic process tuple $T_\Lambda \times \rho T_\Lambda$ (which coincides with $\langle \det(T_\lambda) \mid \lambda \in \Lambda \rangle$).

Note that P_Λ is deterministic iff $P_\Lambda = \det(\tau P_\Lambda)$. □

Tuples having the empty tree $\bullet$ and the deadlock process $STOP$ as elements will be needed:

Definition 3.29 For all $\lambda \in \Lambda$, define the tuples $\bullet_\Lambda$ and $STOP_\Lambda$ by:

$$(\bullet_\Lambda)_\lambda = \bullet \qquad\qquad (STOP_\Lambda)_\lambda = STOP$$

Note that:

$$STOP_\Lambda = \det(\bullet_\Lambda)$$ □

Finally, the partial order relation is extended component-wise to process tuples:

Definition 3.30 Let $P_\Lambda, Q_\Lambda \in Proc^\Lambda$. Define $P_\Lambda \sqsubseteq Q_\Lambda$ if $P_\lambda \sqsubseteq Q_\lambda$ for all $\lambda \in \Lambda$. □

$(Proc^\Lambda, \sqsubseteq)$ is easily proved to be a partial order. An extension of an order-related property to process tuples is:

Proposition 3.31 For all $P_\Lambda \in Proc^\Lambda$, $P_\Lambda \sqsupseteq \det(\tau P_\Lambda)$.

Proof. It is sufficient to observe that, for $\lambda \in \Lambda$, using Proposition 3.19:

$$P_\lambda \sqsupseteq \det(\tau P_\lambda) = \det((\tau P_\Lambda)_\lambda) = (\det(\tau P_\Lambda))_\lambda$$ □

3.2.5 Sets of Process Tuples

It will also be necessary to consider sets of process tuples like $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$ (note that, despite the subscript, $\mathcal{P}_\Lambda$ is not a tuple).

The $(\,)_\lambda$ operation ($\lambda \in \Lambda$) on tuples over Λ is pointwise extended to sets of such tuples; thus, when applied to $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$, it produces a set of processes:

Definition 3.32 Let $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$. Then, for all $\lambda \in \Lambda$, $(\mathcal{P}_\Lambda)_\lambda$ is $\{(P_\Lambda)_\lambda \mid P_\Lambda \in \mathcal{P}_\Lambda\}$.

Note that, just as P_λ is shorthand for $(P_\Lambda)_\lambda$, $\mathcal{P}_\lambda$ will abbreviate $(\mathcal{P}_\Lambda)_\lambda$ □

The τ operator is also pointwise extended to sets of process tuples (and of processes).

Definition 3.33 For $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$, let $\tau\mathcal{P}_\Lambda = \{\tau P_\Lambda \mid P_\Lambda \in \mathcal{P}_\Lambda\}$. □

These extensions commute:

Proposition 3.34 For $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$, $(\tau\mathcal{P}_\Lambda)_\lambda = \tau((\mathcal{P}_\Lambda)_\lambda)$. □

Since $(Proc^\Lambda, \sqsubseteq)$ is a partial order, the notion of lub can be considered for a subset $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$, and turns out to be related to the lubs of each $\mathcal{P}_\lambda$:

Proposition 3.35 Every $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$ has a lub, and $\bigsqcup \mathcal{P}_\Lambda = \langle \bigsqcup \mathcal{P}_\lambda \mid \lambda \in \Lambda \rangle$

Proof. Since every $\mathcal{P}_\lambda$ $(\lambda \in \Lambda)$ has a lub (by Theorem 3.24), the proposition is a general property of partial order tuples ([Loeckx & Sieber, 1987], Theorem 4.3). □

This provides the motivation for extending the $\uplus$ operation to sets of process tuples:

Definition 3.36 Let $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$, $\mathcal{P}_\Lambda \neq \varnothing$. Define $\uplus \mathcal{P}_\Lambda$ as $\langle \uplus \mathcal{P}_\lambda \mid \lambda \in \Lambda \rangle$. □

Thus, the fundamental Theorem 3.24 (about the existence of lubs) may be extended from process sets to process tuple sets:

Theorem 3.37 For every non-empty process tuple set $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$, $\bigsqcup \mathcal{P}_\Lambda$ exists and coincides with $\uplus \mathcal{P}_\Lambda$.

Proof. By Proposition 3.35, Theorem 3.24 and Definition 3.36 of $\uplus \mathcal{P}_\Lambda$. □

Complements for Trees

It should be recalled that, like every set of sets closed under intersection and union, $(Tree, \subseteq)$ is a partial order with bottom $\bigcap Tree = \bullet$, and every set $\mathcal{T} \subseteq Tree$ has $\bigcup \mathcal{T}$ as a lub. As was done for process tuples, by general results, these order and lub notions can be component-wise extended to the set $Tree^\Lambda$ of tree tuples; with some overloading, they will also be denoted by $\sqsubseteq$ and $\bigsqcup$ respectively. $Tree^\Lambda$ has bottom $\bullet_\Lambda$.

When applied to a set of process tuples, the lub and τ operator commute:

Lemma 3.38 Let $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$. Then $\tau \bigsqcup \mathcal{P}_\Lambda = \bigsqcup \tau\mathcal{P}_\Lambda$.

Proof.

$$\begin{aligned}
\tau(\bigsqcup \mathcal{P}_\Lambda) &= \tau\langle \uplus \mathcal{P}_\lambda \mid \lambda \in \Lambda \rangle && \text{by Proposition 3.35} \\
&= \langle \bigcup \tau(\mathcal{P}_\lambda) \mid \lambda \in \Lambda \rangle \\
&= \langle \bigcup (\tau\mathcal{P}_\Lambda)_\lambda \mid \lambda \in \Lambda \rangle && \tau \text{ and } (\,)_\lambda \text{ commute, Proposition 3.34} \\
&= \bigsqcup \tau\mathcal{P}_\Lambda && \text{by counterpart of Prop. 3.35 for trees } \square
\end{aligned}$$

3.2.6 Properties of Process Functions

A function F: *Proc* $\rightarrow$ *Proc* may enjoy several desirable properties:

- *monotonicity*, if it preserves the order $\sqsubseteq$;
- *continuity*, if it preserves the lub of a chain;
- *distributivity*, if it preserves non-deterministic choice (i.e., in the chosen process order, the lub of a non-empty set).

These notions readily generalize to functions over process tuples. Formally:

Definition 3.39 A function F: $Proc^{\Lambda} \rightarrow Proc^{\Delta}$ is said to be:

- *monotonic* if, for all $P_{\Lambda}, Q_{\Lambda} \in Proc^{\Lambda}$, $P_{\Lambda} \sqsubseteq Q_{\Lambda}$ implies $F(P_{\Lambda}) \sqsubseteq F(P_{\Delta})$;
- *continuous* if, for all chain $\mathcal{P}_{\Lambda}$ in $Proc^{\Lambda}$, the lub $\bigsqcup F(\mathcal{P}_{\Lambda})$ exists and $\bigsqcup F(\mathcal{P}_{\Lambda}) = F(\bigsqcup \mathcal{P}_{\Lambda})$;
- *distributive* if, for all subset $\mathcal{P}_{\Lambda}$ of $Proc^{\Lambda}$, $\uplus F(\mathcal{P}_{\Lambda}) = F(\uplus \mathcal{P}_{\Lambda})$. □

In the general theory of partial orders, continuity implies monotonicity. Moreover, for our $\sqsubseteq$ partial order, since every process tuple set has a lub (by Theorem 3.37), distributivity is enough to ensure continuity. Thus:

Theorem 3.40 Let F: $Proc^{\Lambda} \rightarrow Proc^{\Delta}$ be a function. Then:

1. if F is continuous, then it is monotonic;
2. if F is distributive, then it is continuous (this fact is normally used just to prove that a function F: *Proc* $\rightarrow$ *Proc* is continuous).

Proof. For (1), see [Loeckx & Sieber, 1987], Theorem 4.14. (2) follows immediately from Theorem 3.37 and distributivity. □

Complex functions may be built out of simpler ones by the usual (function) composition and by *juxtaposition*, defined through tuple construction as follows:

Definition 3.41 Let $F_\delta : Proc^\Lambda \to Proc$ be a function for every $\delta \in \Delta$. Then the function $F_\Delta : Proc^\Lambda \to Proc^\Delta$ is defined, for $P_\Lambda \in Proc^\Lambda$, by: $F_\Delta(P_\Lambda) = \langle F_\delta(P_\Lambda) \mid \delta \in \Delta \rangle$.

Conversely, given a function $G : Proc^\Lambda \to Proc^\Delta$, the *component* functions $G_\delta : Proc^\Lambda \to Proc$ ($\delta \in \Delta$) are defined, for $P_\Lambda \in Proc^\Lambda$, by: $G_\delta(P_\Lambda) = (G(P_\Lambda))_\delta$. □

Luckily, any property among monotonicity, continuity and distributivity may be determined, for a complex function, from the corresponding property of its component functions. This is true independently of the particular partial order $\sqsubseteq$ chosen. This fact will be shown for function composition first.

Proposition 3.42 Let $F : Proc^\Theta \to Proc^\Delta$, $G : Proc^\Lambda \to Proc^\Theta$ be functions. Then the function $F \circ G : Proc^\Lambda \to Proc^\Delta$ is:

1. monotonic if F and G are monotonic;
2. continuous if F and G are continuous;
3. distributive if F and G are distributive.

Proof. For monotonicity, the proof is immediate. For continuity, it is a standard result of partial order theory ([Loeckx & Sieber, 1987], Theorem 4.21).

For distributivity, let $\varnothing \subset \mathcal{P}_\Lambda \subseteq Proc^\Lambda$. Applying distributivity of G, then F yields:

$$(F \circ G)(\biguplus \mathcal{P}_\Lambda) = F(G(\biguplus \mathcal{P}_\Lambda)) = F(\biguplus G(\mathcal{P}_\Lambda)) = \biguplus F(G(\mathcal{P}_\Lambda)) = \biguplus (F \circ G)(\mathcal{P}_\Lambda) \quad \square$$

A similar result applies to juxtaposition and component functions:

Proposition 3.43 $F : Proc^\Lambda \to Proc^\Delta$ is:

1. monotonic iff, for every $\delta \in \Delta$, F_δ is monotonic;
2. continuous iff, for every $\delta \in \Delta$, F_δ is continuous;
3. distributive iff, for every $\delta \in \Delta$, F_δ is distributive.

Proof. For the monotonicity part, let $P_\Lambda, Q_\Lambda \in Proc^\Lambda$, $P_\Lambda \sqsubseteq Q_\Lambda$. Then $F(P_\Lambda) \sqsubseteq F(Q_\Lambda)$ iff, for all $\delta \in \Delta$, $(F(P_\Lambda))_\delta \sqsubseteq (F(Q_\Lambda))_\delta$ iff, for all $\delta \in \Delta$, $F_\delta(P_\Lambda) \sqsubseteq F_\delta(Q_\Lambda)$. The continuity result is standard ([Loeckx & Sieber, 1987], Theorem 4.20).

For distributivity, let $\varnothing \subset \mathcal{P}_\Lambda \subseteq Proc^\Lambda$. By definition of $\biguplus$ on tuples:

$$\biguplus F(\mathcal{P}_\Lambda) = \langle \biguplus F_\delta(\mathcal{P}_\Lambda) \mid \delta \in \Delta \rangle$$

On the other hand, by definition of F_δ:

$$F(\biguplus \mathcal{P}_\Lambda) = \langle F_\delta(\biguplus \mathcal{P}_\Lambda) \mid \delta \in \Delta \rangle$$

Now F is distributive iff the two right hand sides above coincide, i.e. iff each F_δ ($\delta \in \Delta$) is distributive. □

Most basic process functions employed in the following are binary. The following result is then very handy, in that it allows such a function to be recognized as monotonic or continuous by simply checking that it is so with respect to each argument separately.

Theorem 3.44 Let $F\colon Proc^2 \to Proc$ be a function. For any $Q \in Proc$, define the functions $G_Q\colon Proc \to Proc$ and $H_Q\colon Proc \to Proc$ by letting, for $P \in Proc$:

$$G_Q(P) = F(P, Q) \qquad\qquad H_Q(P) = F(Q, P)$$

Then F is continuous (resp. monotonic) iff, for every process Q, G_Q and H_Q are continuous (resp. monotonic) .

Proof. For monotonicity, the only-if part is obvious; for the if-part, let $P_1 \sqsubseteq Q_1$ and $P_2 \sqsubseteq Q_2$; then:

$$F(P_1, P_2) = H_{P_1}(P_2) \sqsubseteq H_{P_1}(Q_2) = G_{Q_2}(P_1) \sqsubseteq G_{Q_2}(Q_1) = F(Q_1, Q_2)$$

For continuity, see [Loeckx & Sieber, 1987], 4.2-5. □

As a matter of fact, the result that continuity in a finite number of arguments is implied by continuity in each argument holds for any partial order. Moreover, for our $\sqsubseteq$, it is possible to infer continuity and monotonicity in an argument from distributivity in that argument (Theorem 3.40).

3.2.7 Fixpoints

This section tackles the problem of determining a fixpoint of a continuous function F: $Proc^\Lambda \to Proc^\Lambda$, i.e. a process tuple $P_\Lambda \in Proc^\Lambda$ such that:

$$P_\Lambda = F(P_\Lambda)$$

Trace-based Functions

In fact, a fixpoint will be sought for a continuous function F that is also *trace-based*, in the sense that the trace set of its image is only determined by the trace set of its argument.

Definition 3.45 A function $F : Proc^\Lambda \to Proc^\Delta$ is *trace-based* if there exists a function F_τ: $Tree^\Lambda \to Tree^\Delta$ such that, for all $P_\Lambda \in Proc^\Lambda$:

$$\tau F(P_\Lambda) = F_\tau(\tau P_\Lambda) \qquad \square$$

In practice, defining a fixpoint only for trace-based functions is not a serious limitation. For all the basic process functions to be introduced in Section 3.3 are trace-based, and it is immediate to check that function composition and juxtaposition preserve this property.

The pointwise extension to a set of a trace-based function is still trace-based:

Lemma 3.46 Let $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$, F: $Proc^\Lambda \to Proc^\Delta$ trace-based. Then

$$\tau F(\mathcal{P}_\Lambda) = F_\tau(\tau \mathcal{P}_\Lambda)$$

Proof.

$$\begin{aligned}
\tau F(\mathcal{P}_\Lambda) &= \tau\{F(P_\Lambda) \mid P_\Lambda \in \mathcal{P}_\Lambda\} \\
&= \{\tau F(P_\Lambda) \mid P_\Lambda \in \mathcal{P}_\Lambda\} \\
&= \{F_\tau(\tau P_\Lambda) \mid P_\Lambda \in \mathcal{P}_\Lambda\} \qquad (F \text{ trace-based}) \\
&= F_\tau(\{\tau P_\Lambda \mid P_\Lambda \in \mathcal{P}_\Lambda\}) \\
&= F_\tau(\tau \mathcal{P}_\Lambda)
\end{aligned}$$

$\square$

Taking the trace version F_τ of a trace-based process function F preserves monotonicity and continuity.

Proposition 3.47 Let a function $F : Proc^{\Lambda} \to Proc^{\Delta}$ be trace-based so that $F_\tau : Tree^{\Lambda} \to Tree^{\Lambda}$. Then:

1. F_τ is continuous if F is continuous;
2. F_τ is monotonic if F is monotonic.

Proof. The reasoning for monotonicity is easy, along the lines of that below for continuity.

Let $\mathcal{T}_\Lambda$ be a chain in $Tree^{\Lambda}$. Introduce the refusal function $r \in RefFun$ such that for all trace s, $r(s) = \mathrm{p}Act^{+}$, and the refusal function tuple r_Λ such that $(r_\Lambda)_\lambda = r$. Now define (recalling Remark 3.4):

$$\mathcal{P}_\Lambda = \{T_\Lambda \times r_\Lambda \mid T_\Lambda \in \mathcal{T}_\Lambda\}$$

By Proposition 3.27, $\mathcal{P}_\Lambda$ is a subset of $Proc^{\Lambda}$. Moreover, it is a chain: let $T_\Lambda \times r_\Lambda, U_\Lambda \times r_\Lambda$ with $T_\Lambda, U_\Lambda \in \mathcal{T}_\Lambda$ be members of $\mathcal{P}_\Lambda$; without loss of generality, assume $T_\Lambda \sqsubseteq U_\Lambda$ ($\mathcal{T}_\Lambda$ is a chain); then $(T_\lambda, r) \sqsubseteq (U_\lambda, r)$ for all $\lambda \in \Lambda$ and $T_\Lambda \times r_\Lambda \sqsubseteq U_\Lambda \times r_\Lambda$. Since $F : Proc^{\Lambda} \to Proc^{\Delta}$ is continuous, $F(\mathcal{P}_\Lambda)$ has a lub and $\bigsqcup F(\mathcal{P}_\Lambda) = F(\bigsqcup \mathcal{P}_\Lambda)$. Thus:

$$\tau\big(\bigsqcup F(\mathcal{P}_\Lambda)\big) = \tau F(\bigsqcup \mathcal{P}_\Lambda)$$

Now, the left-hand side rewrites (by Lemma 3.38) to $\bigsqcup \tau(F(\mathcal{P}_\Lambda))$ and (by Lemma 3.46) to $\bigsqcup F_\tau(\tau\mathcal{P}_\Lambda)$, i.e. $\bigsqcup F_\tau(\mathcal{T}_\Lambda)$. The right hand side rewrites (by F_τ trace-based) to $F_\tau(\tau \bigsqcup \mathcal{P}_\Lambda)$ and (by Lemma 3.38) to $F_\tau(\bigsqcup \tau\mathcal{P}_\Lambda)$, i.e. $F_\tau(\bigsqcup \mathcal{T}_\Lambda)$. □

Fixpoints of Trace-based Continuous Functions

For a continuous, trace-based (endo-)function F it is always possible to find $\bot_F$, a member of F's domain which is never greater than its F-image.

Definition 3.48 Let $F : Proc^{\Lambda} \to Proc^{\Lambda}$ be a continuous, trace-based function. Define

$$\bot_F = \det\big(\bigsqcup\{F_\tau{}^n(\bullet_\Lambda) \mid n \geqslant 0\}\big)$$

Note that, since $\{F_\tau{}^n(\bullet_\Lambda) \mid n \geqslant 0\}$ always has a lub (like any set of tree tuples), $\bot_F$ is well-defined. Moreover, it is a chain ($\bullet_\Lambda$ is the bottom of $Tree^{\Lambda}$ and F_τ is monotonic by Proposition 3.47). □

Lemma 3.49 Let F: $Proc^{\Lambda} \to Proc^{\Lambda}$ be a continuous, trace-based function. Then $\tau\bot_F$ is the least fixpoint of F_τ.

Proof. A standard result of fixpoint theory, applied to the partial order $(Tree^{\Lambda}, \sqsubseteq)$ and the continuous function F_τ: $Tree^{\Lambda} \to Tree^{\Lambda}$. □

The fundamental result of this section is:

Theorem 3.50 Let F : $Proc^{\Lambda} \to Proc^{\Lambda}$ be a continuous, trace-based function. Then:

1. $\tau F^n(\bot_F) = \tau\bot_F$ for $n \geqslant 0$;
2. $\{F^n(\bot_F) \mid n \geqslant 0\}$ is a chain and its lub is a fixpoint for F.

Proof. The first statement is proved by induction: the basis is trivial; for the step:

$$\tau F^{n+1}(\bot_F) = \tau F(F^n(\bot_F)) = F_\tau(\tau F^n(\bot_F)) = F_\tau(\tau\bot_F) = \tau\bot_F$$

which orderly exploits: F being trace-based, the induction hypothesis, and $\tau\bot_F$ being a fixpoint of F_τ.

For part (2), we first prove by induction that $F^n(\bot_F) \sqsubseteq F^{n+1}(\bot_F)$ holds for $n \geqslant 0$. The basis is derived using Proposition 3.31, part (1) and the definition of $\bot_F$:

$$F(\bot_F) \sqsupseteq \det(\tau F(\bot_F)) = \det(\tau\bot_F) = \bot_F$$

The step uses monotonicity of F and the induction hypothesis:

$$F^{n+1}(\bot_F) = F(F^n(\bot_F)) \sqsubseteq F(F^{n+1}(\bot_F)) = F^{n+2}(\bot_F)$$

Thus $\{F^n(\bot_F) \mid n \geqslant 0\}$ is a chain. Now, by continuity of F:

$$\begin{aligned} F(\bigsqcup\{F^n(\bot_F) \mid n \geqslant 0\}) &= \bigsqcup\{F^n(\bot_F) \mid n > 0\} \\ &= \bigsqcup\{F^n(\bot_F) \mid n \geqslant 0\} \end{aligned}$$

Thus, $\bigsqcup\{F^n(\bot_F) \mid n \geqslant 0\}$ is a fixpoint of F. □

That just identified will be taken to be 'the' fixpoint of a function:

Definition 3.51 Let $F: Proc^{\Lambda} \to Proc^{\Lambda}$ be a continuous, trace-based function. Then define the process tuple $\mathrm{fix}\, F \in Proc^{\Lambda}$ as:

$$\mathrm{fix}\, F = \bigsqcup\{F^n(\bot_F) \mid n \geqslant 0\}$$

By Theorem 3.50, $\mathrm{fix}\, F$ is indeed a fixpoint of F. □

In general, it could be shown, along the lines of [Brookes, Hoare, & Roscoe, 1984], that if a function is $\mathrm{PL}^{\oplus}_{\uplus}$-generated (see Definition 2.4), it may have several fixpoints, unless all process constants are guarded in the generating process expressions. However, the fixpoint $\mathrm{fix}\, F$ is minimal in the following sense:

Proposition 3.52 For a continuous, trace-based function $F: Proc^{\Lambda} \to Proc^{\Lambda}$, $\mathrm{fix}\, F$ is the least fixpoint with trace set $\tau\bot_F$.

Proof. Let Q be a fixpoint for F and $\tau Q = \tau\bot_F$. Then, using Proposition 3.31:

$$Q \sqsupseteq \det(\tau Q) = \det(\tau\bot_F) = \bot_F$$

Hence, by induction, $Q \sqsupseteq F^n(\bot_F)$. Thus, Q is an upper bound of the same chain that has $\mathrm{fix}\, F$ as lub. □

It will be recalled from Section 2.4.3 that the compositional denotational semantics of the process language $\mathrm{PL}^{\oplus}_{\uplus}$ presupposes that every $\mathrm{PL}^{\oplus}_{\uplus}$-generated process function possesses a fixpoint. For our semantics, this is ensured by the following:

Remark 3.53 Every continuous, trace-based process function F has a fixpoint $\mathrm{fix}\, F$, as defined above. This still holds if continuity is replaced by monotonicity, as shown in the following section.

All the basic process functions defined in Section 3.3 are trace-based, and continuous or, at least, monotonic.

Furthermore, $\mathrm{PL}^{\oplus}_{\uplus}$-generated functions are also trace-based, and continuous or monotonic; this follows, by structural induction, from the previous fact and because each of these three properties is preserved by function composition and juxtaposition (see Section 3.2.6). □

Fixpoints of Trace-based Monotonic Functions

The process tuple $\bot_F$ may be introduced also for a monotonic function F.

Definition 3.54 Let $F : \mathit{Proc}^\Lambda \to \mathit{Proc}^\Lambda$ be a monotonic, trace-based function. Define

$$\bot_F = \det(\operatorname{fix} F_\tau)$$

where F_τ is monotonic by Proposition 3.47 and $\operatorname{fix} F_\tau$ is its least fixpoint (which exists by Knaster-Tarski's theorem). □

As for a continuous F, $\bot_F$ is never greater than its F-image:

Proposition 3.55 Let $F: \mathit{Proc}^\Lambda \to \mathit{Proc}^\Lambda$ be a monotonic, trace-based function. Then:

1. $\tau F^n(\bot_F) = \tau\bot_F$ for $n \geqslant 0$;
2. $\{F^n(\bot_F) \mid n \geqslant 0\}$ is a chain.

Proof. See Theorem 3.50. □

A non-trivial adaptation of Knaster-Tarski's theorem shows that the 'fix' functional of Definition 3.51 can be extended to monotonic, trace-based functions.

Theorem 3.56 Let $F: \mathit{Proc}^\Lambda \to \mathit{Proc}^\Lambda$ be a monotonic, trace-based function, and

$$\Pi_F = \{\mathcal{P} \subseteq \mathit{Proc}^\Lambda \mid F(\mathcal{P}) \subseteq \mathcal{P},\ \bot_F \in \mathcal{P},\ \forall \text{nonempty } \mathcal{S} \subseteq \mathcal{P} : \bigsqcup \mathcal{S} \in \mathcal{P}\}$$

Then $\bigsqcup \bigcap \Pi_F$ is a fixpoint of F, and will be denoted by $\operatorname{fix} F$.

Proof. Π_F comprises all process sets that are closed under F and are 'complete' with respect to subsets (rather than sub-chains).

First, note $\Pi_F \neq \varnothing$, because $\mathit{Proc}^\Lambda \in \Pi_F$. Also, $\bigcap \Pi_F \in \Pi_F$ is easy to establish. It will now be proved that:

$$\mathcal{P}_F = \{P_\Lambda \in \mathit{Proc}^\Lambda \mid P_\Lambda \sqsubseteq F(P_\Lambda)\} \quad \in \quad \Pi_F$$

That $\mathcal{P}_F$ is closed under F is an immediate consequence of F being monotonic. Clearly, $\bot_F \in \mathcal{P}_F$. We must also show that, if $\mathcal{S}$ is a non-empty subset of $\mathcal{P}_F$, then

$\bigsqcup \mathcal{S} \in \mathcal{P}_F$, i.e. $\bigsqcup \mathcal{S} \sqsubseteq F(\bigsqcup \mathcal{S})$; for this it is enough to show that $F(\bigsqcup \mathcal{S})$ is an upper bound of $\mathcal{S}$:

$P_\Lambda \in \mathcal{S} \subseteq \mathcal{P}_F$	assumption	(1)
$P_\Lambda \sqsubseteq \bigsqcup \mathcal{S}$	from (1)	(2)
$P_\Lambda \sqsubseteq F(P_\Lambda) \sqsubseteq F(\bigsqcup \mathcal{S})$	from (1) and, by F monotonic, (2)	

Finally:

$\bigsqcup \bigcap \Pi_F \in \bigcap \Pi_F$	by $\bigcap \Pi_F \in \Pi_F$	(3)
$\bigsqcup \bigcap \Pi_F \in \mathcal{P}_F$	from (3), by $\mathcal{P}_F \in \Pi_F$	(4)
$\bigsqcup \bigcap \Pi_F \sqsubseteq F(\bigsqcup \bigcap \Pi_F)$	from (4), using definition of $\mathcal{P}_F$	(5)
$F(\bigsqcup \bigcap \Pi_F) \in \bigcap \Pi_F$	from (3), by $\bigcap \Pi_F \in \Pi_F$ ($\bigcap \Pi_F$ is F-closed)	(6)

From (5) and (6) it follows that $\bigsqcup \bigcap \Pi_F$ is a fixpoint. □

Theorem 3.57 If $F : \mathit{Proc}^\Lambda \to \mathit{Proc}^\Lambda$ is continuous and trace-based, $\mathrm{fix}\, F$ of Theorem 3.56 coincides with $\mathrm{fix}\, F$ of Definition 3.51.

Proof. Let $\bot_F$ be as in Definition 3.54 and

$$\mathcal{F} = \{F^n(\bot_F) \mid n \geqslant 0\} \qquad \overline{\mathcal{F}} = \mathcal{F} \cup \{\bigsqcup \mathcal{F}\}$$

$\overline{\mathcal{F}}$ belongs to the set Π_F defined in Theorem 3.56: indeed $F(F^n(\bot_F)) \in \overline{\mathcal{F}}$ is obvious, $F(\bigsqcup \mathcal{F}) \in \overline{\mathcal{F}}$ holds because $\bigsqcup \mathcal{F}$ is a fixpoint of F (Theorem 3.50), $\bot_F \in \overline{\mathcal{F}}$ is obvious; so we need only show that, if $\varnothing \subset \mathcal{S} \subseteq \overline{\mathcal{F}}$, then $\bigsqcup \mathcal{S} \in \overline{\mathcal{F}}$. This is easy if $\bigsqcup \mathcal{F} \in \mathcal{S}$ or $\mathcal{S}$ is finite; if, instead, for all $k \geqslant 0$ there is $n_k \geqslant k$ such that $F^{n_k}(\bot_F) \in \mathcal{S}$, then $\bigsqcup \mathcal{S}$ is an upper bound of $\mathcal{F}$ and hence must coincide with $\bigsqcup \mathcal{F}$. Moreover, if $\mathcal{P} \in \Pi_F$, it is easy to see that $\overline{\mathcal{F}} \subseteq \mathcal{P}$. It follows that $\bigcap \Pi_F = \overline{\mathcal{F}}$, whence:

$$\bigsqcup \bigcap \Pi_F = \bigsqcup \overline{\mathcal{F}} = \bigsqcup \mathcal{F}$$ □

3.2.8 A Fixpoint Induction Rule

A tool is now needed to reason about fixpoints of functions, proving that they satisfy the desired properties, expressed as a predicate ψ over process tuples. For

this purpose, a fixpoint induction rule similar to Scott's (see [Loeckx & Sieber, 1987]) can be given. The predicate ψ is *admissible* (for application of the induction rule) if, whenever it is satisfied by every member of a chain, it is also satisfied by the lub of the chain.

Definition 3.58 A predicate $\psi : \mathit{Proc}^\Lambda \to \mathit{Bool}$ is *admissible* if, for every chain $\mathcal{P}_\Lambda \subseteq \mathit{Proc}^\Lambda$ such that $\psi(P)$ holds for all $P \in \mathcal{P}_\Lambda$, also $\psi(\biguplus \mathcal{P}_\Lambda)$ holds. □

The induction rule can now be stated.

Theorem 3.59 Let F: $\mathit{Proc}^\Lambda \to \mathit{Proc}^\Lambda$ be a continuous, trace-based function, and ψ: $\mathit{Proc}^\Lambda \to \mathit{Bool}$ an admissible predicate such that:

1. $\psi(\bot_F)$ holds (basis), and
2. for all $P_\Lambda \in \mathit{Proc}^\Lambda$, if $\psi(P_\Lambda)$ holds, then $\psi(F(P_\Lambda))$ holds (ψ is F-inductive).

Then $\psi(\mathrm{fix}\, F)$ holds.

Proof. By induction it is easy to prove that $\psi(F^n(\bot_F))$ is true for $n \geqslant 0$. Since ψ is admissible, $\psi(\biguplus\{F^n(\bot_F) \mid n \geqslant 0\})$. So by the fixpoint theorem, $\psi(\mathrm{fix}\, F)$ is true. □

It would now be possible to show that the class of admissible predicates is adequate for practical purposes. This will not be done here because this induction rule is of limited applicability. The difficulty in its use lies in the requirement that $\psi(\bot_F)$ should hold; for, as a result of the inductive definition of $\bot_F$, the proof of $\psi(\bot_F)$ is likely to need some form of induction itself. It should be noted that a similar difficulty arises in using the induction rule of [Brookes, Hoare, & Roscoe, 1984], which requires $\psi(Q_\Lambda)$ to hold for at least some process tuple Q_Λ—a fact that may require an induction in order to be proved. A more manageable induction rule may be obtained by restricting the classes of functions and predicates.

Trace-Based Semi-Process Functions

The generic process function F that will now be considered must have in fact semi-process tuples as domain and range. However, F is also required to be process-preserving, so that its restriction to the set of process tuples has a set of process

tuples as range. Viewed as a function over process tuples, F must be continuous and trace-based so that $\mathrm{fix}\, F$ is defined. Viewed in its entirety, F must also be trace-based, though in a sense that is wider than that introduced for process tuple functions.

Explaining this sense requires extending the notion of partial order to semi-process tuples. Actually, partial order and also lub definitions (3.17), (3.20) for processes are already immediately applicable to semi-processes, for they make no reference to consistency between traces and refusals. Therefore, in the following the relation $\sqsubseteq$ and the operator $\bigsqcup$ will be indifferently applied also to semi-processes and semi-process tuples. Likewise, the definition of monotonicity is extended without change to functions over semi-process tuples.

A function F over semi-process tuples is trace-based if it 'commutes' not only with the τ operator, but also, in a limited way, with the ρ operator on tree tuples (Definitions 3.12, 3.28):

Definition 3.60 A function $F\colon \overline{\mathcal{P}}roc^{\Lambda} \to \overline{\mathcal{P}}roc^{\Delta}$ is said to be *trace-based* if there exists a function $F_\tau\colon Tree^{\Lambda} \to Tree^{\Delta}$ such that, for all $P_\Lambda \in \overline{\mathcal{P}}roc^{\Lambda}$, $T_\Lambda, U_\Lambda \in Tree^{\Lambda}$, $T_\Lambda \sqsubseteq U_\Lambda$:

$$\tau F(P_\Lambda) = F_\tau(\tau P_\Lambda) \qquad\qquad F(T_\Lambda \times \rho U_\Lambda) \sqsupseteq F_\tau(T_\Lambda) \times \rho F_\tau(U_\Lambda)$$

If only the first condition holds, F is said to be *weakly* trace-based. □

Note that, if $F : \overline{\mathcal{P}}roc^{\Lambda} \to \overline{\mathcal{P}}roc^{\Delta}$ is weakly trace-based and process-preserving, then its restriction to $Proc^{\Lambda}$ is trace-based according to Definition 3.45.

The question now arises whether the function class introduced is large enough and how membership can be determined for it. An answer is provided by the following:

Remark 3.61 Except hiding, all the basic process functions defined in Section 3.3 are *well-behaved*, in the sense that they enjoy these properties:

1. they map a semi-process tuple to a semi-process;
2. they are trace-based and monotonic (monotonicity is needed for Proposition 3.62);

3. they are process-preserving;

4. their restriction to process tuples is continuous.

Hiding-free $PL^{\oplus}_{\Psi}$-generated functions (see Definition 2.4) enjoy the above properties; this follows, by structural induction, from the previous fact and because each of the above properties is preserved by function composition and juxtaposition. □

That composition and juxtaposition preserve properties (1–4) above is either obvious or has already been established (see Section 3.2.6), except for composition and property (2):

Proposition 3.62 Let $F: \bar{\mathcal{P}}roc^{\Theta} \to \bar{\mathcal{P}}roc^{\Delta}$, $G: \bar{\mathcal{P}}roc^{\Lambda} \to \bar{\mathcal{P}}roc^{\Theta}$ be functions. Then the function $F \circ G: \bar{\mathcal{P}}roc^{\Lambda} \to \bar{\mathcal{P}}roc^{\Delta}$ is trace-based and monotonic if F and G are trace-based and monotonic.

Proof. The proof is immediate for monotonicity and the first condition for $F \circ G$ to be trace-based. For the second condition, let $T_\Lambda, U_\Lambda \in \mathcal{T}ree^{\Lambda}$, $T_\Lambda \sqsubseteq U_\Lambda$. Then:

$$\begin{aligned}
G_\tau(U_\Lambda) &\sqsupseteq G_\tau(T_\Lambda) && \text{by } G \text{ monotonic}\\
G(T_\Lambda \times \rho U_\Lambda) &\sqsupseteq G_\tau(T_\Lambda) \times \rho G_\tau(U_\Lambda) && \text{by } G \text{ trace-based}\\
F(G(T_\Lambda \times \rho U_\Lambda)) &\sqsupseteq F(G_\tau(T_\Lambda) \times \rho G_\tau(U_\Lambda)) && \text{by } F \text{ monotonic}\\
&\sqsupseteq F_\tau(G_\tau(T_\Lambda)) \times \rho F_\tau(G_\tau(U_\Lambda)) && \text{by } F \text{ trace-based}
\end{aligned}$$

The last two inequalities complete the proof. □

For a *refusal-based* process function F, the ρ-component of the image is only determined by the ρ-component of the argument:

Definition 3.63 A function $F: \bar{\mathcal{P}}roc^{\Lambda} \to \bar{\mathcal{P}}roc$ is said to be *refusal-based* if there exists a function $F_\rho: \mathcal{R}efFun^{\Lambda} \to \mathcal{R}efFun$ such that for all $T_\Lambda \times r_\Lambda \in \bar{\mathcal{P}}roc^{\Lambda}$, over the trace domain $\tau F(T_\Lambda \times r_\Lambda)$:

$$\rho F(T_\Lambda \times r_\Lambda) = F_\rho(r_\Lambda)$$ □

This allows the statement of a useful sufficient condition for a process function to be trace-based:

Proposition 3.64 Let $F\colon \overline{Proc}^{\Lambda} \to \overline{Proc}$ be monotonic, weakly trace-based and refusal-based. Then F is trace-based.

Proof. Let $T_\Lambda, U_\Lambda \in Tree^{\Lambda}$, $T_\Lambda \sqsubseteq U_\Lambda$.

By the hypotheses and Proposition 3.19:

$$F(U_\Lambda \times \rho U_\Lambda) = (F_\tau(U_\Lambda), F_\rho(\rho U_\Lambda)) \sqsupseteq \det(F_\tau(U_\Lambda)) = (F_\tau(U_\Lambda), \rho F_\tau(U_\Lambda))$$

Thus, for all $s \in F_\tau(U_\Lambda)$:

$$F_\rho(\rho U_\Lambda)(s) \supseteq \rho F_\tau(U_\Lambda)(s)$$

In particular, this condition holds for $s \in F_\tau(T_\Lambda)$ ($T_\Lambda \sqsubseteq U_\Lambda$ and F_τ is monotonic by Proposition 3.47). Using this fact and the hypotheses:

$$F(T_\Lambda \times \rho U_\Lambda) = (F_\tau(T_\Lambda), F_\rho(\rho U_\Lambda)) \sqsupseteq (F_\tau(T_\Lambda), \rho F_\tau(U_\Lambda)) \qquad \square$$

Finally, it is convenient to show that $\biguplus$, viewed as an operator, enjoys a property analogous to being trace-based:

Proposition 3.65 Let $T_n, V_n \in Tree$ for $n \geqslant 0$. Then

$$\biguplus\{(T_n, \rho V_n) \mid n \geqslant 0\} \sqsupseteq (\bigcup\{T_n \mid n \geqslant 0\}, \rho\bigcup\{V_n \mid n \geqslant 0\})$$

Proof. Let Q denote the left-hand side above. Then $\tau Q = \bigcup\{T_n \mid n \geqslant 0\}$ and, for all $s \in \tau Q$:

$$\rho Q(s) \;=\; \bigcup\{\rho V_n(s) \mid n \geqslant 0, s \in T_n\}$$

Let $s \in T_m$ for some $m \geqslant 0$; then, using also Proposition 3.13:

$$\rho Q(s) \supseteq \rho V_m(s) \supseteq \bigcap\{\rho V_n(s) \mid n \geqslant 0\} = \rho(\bigcup\{V_n \mid n \geqslant 0\})(s) \qquad \square$$

This can be generalized to tuples:

Proposition 3.66 Let $T_n, V_n \in Tree^{\Lambda}$ for $n \geqslant 0$. Then

$$\biguplus\{T_n \times \rho V_n \mid n \geqslant 0\} \sqsupseteq \bigsqcup\{T_n \mid n \geqslant 0\} \times \rho\bigsqcup\{V_n \mid n \geqslant 0\}$$

Proof. Define the sets:

$$\mathcal{P} = \{T_n \times \rho V_n \mid n \geqslant 0\} \qquad \mathcal{T} = \{T_n \mid n \geqslant 0\} \qquad \mathcal{V} = \{V_n \mid n \geqslant 0\}$$

Then the proposition follows if, for all $\lambda \in \Lambda$, $(\biguplus \mathcal{P})_\lambda \sqsupseteq ((\bigsqcup \mathcal{T})_\lambda, (\rho \bigsqcup \mathcal{V})_\lambda)$, i.e. if for all $\lambda \in \Lambda$:

$$\biguplus \mathcal{P}_\lambda \sqsupseteq (\bigcup \mathcal{T}_\lambda, \rho(\bigcup \mathcal{V}_\lambda)) \tag{1}$$

Indeed, for $\lambda \in \Lambda$, letting $T_{\lambda,n} = (T_n)_\lambda$, $V_{\lambda,n} = (V_n)_\lambda$ yields:

$$\mathcal{P}_\lambda = \{(T_{\lambda,n}, \rho V_{\lambda,n}) \mid n \geqslant 0\} \qquad \mathcal{T}_\lambda = \{T_{\lambda,n} \mid n \geqslant 0\} \qquad \mathcal{V}_\lambda = \{V_{\lambda,n} \mid n \geqslant 0\}$$

and (1) is ensured by applying to these sets, for each $\lambda \in \Lambda$, Proposition 3.65, which yields:

$$\biguplus\{(T_{\lambda,n}, \rho V_{\lambda,n}) \mid n \geqslant 0\} \sqsupseteq (\bigcup\{T_{\lambda,n} \mid n \geqslant 0\}, \rho \bigcup\{V_{\lambda,n} \mid n \geqslant 0\}) \qquad \square$$

Another Partial Order

It is convenient at this stage to introduce briefly another partial order $\boxleftarrow$, to capture the fact that a (semi-)process is 'lazier' (has less traces and more refusals) than another one.

Definition 3.67 For all $P, Q \in \bar{\mathcal{P}}roc$, the relation $P \boxleftarrow Q$ holds if:

1. $\tau P \subseteq \tau Q$,
2. $\rho P(s) \supseteq \rho Q(s)$ for all $s \in \tau P$.

$P \boxleftarrow Q$ is also written $Q \boxrightarrow P$. $\square$

This relation is clearly a partial order over $\bar{\mathcal{P}}roc$. A $\boxleftarrow$-chain is simply a chain for the order $\boxleftarrow$.

It is easy to define an operator $\bowtie$ that, applied to a semi-process set, represents its lub for the partial order $\boxleftarrow$.

Definition 3.68 Let $\mathcal{P} \subseteq \bar{\mathcal{P}}roc$, $\mathcal{P} \neq \varnothing$. Define $\bowtie\mathcal{P}$ by the pair:

$$\tau(\bowtie\mathcal{P}) = \bigcup\{\tau P \mid P \in \mathcal{P}\} \qquad \rho(\bowtie\mathcal{P})(s) = \bigcap\{\rho P(s) \mid P \in \mathcal{P},\ s \in \tau P\} \qquad \square$$

$\bowtie\mathcal{P}$ is indeed a semi-process:

Proposition 3.69 If $\mathcal{P} \subseteq \overline{\mathcal{P}roc}$, $\mathcal{P} \neq \varnothing$, then $\bowtie\mathcal{P} \in \overline{\mathcal{P}roc}$.

Proof. $\tau\bowtie\mathcal{P}$ is easily proved a tree using just the fact that, for each $P \in \mathcal{P}$, τP is a tree.

To show that $\rho\bowtie\mathcal{P}$ is a refusal function, assume $s \in \tau\bowtie\mathcal{P}$. Then $\rho\bowtie\mathcal{P}(s)$ is non-empty (intersection of a non-empty set of sets each containing $\varnothing$). Further, let $X \in \rho\bowtie\mathcal{P}$; then, for all $P \in \mathcal{P}$, $s \in \tau P$ (there must be at least one such P) implies $X \in \rho P(s)$. If $Y \subseteq X$, $Y \in \rho P(s)$ by the subset-closure property of $\rho P(s)$, whence also $Y \in \rho\bowtie\mathcal{P}(s)$. If $t \in X$ and w is a trace, then $X \cup \{tw\} \in \rho P(s)$ by the suffix-closure property of $\rho P(s)$, whence also $X \cup \{tw\} \in \rho\bowtie\mathcal{P}(s)$. □

It is worth noting that the partial order $(\overline{\mathcal{P}roc}, \mathbin{\boxtimes})$ has *STOP* as a bottom.

The generalization of $\mathbin{\boxtimes}$ to semi-process tuples is defined component-wise, exactly as for the $\sqsubseteq$ order. The $\bowtie$ operator can also be extended component-wise to a subset $\mathcal{P}_\Lambda \subseteq \overline{\mathcal{P}roc}^\Lambda$ and turns out to be the lub of $\mathcal{P}_\Lambda$ for the order $\mathbin{\boxtimes}$.

Two connections between $\sqsubseteq$ and $\mathbin{\boxtimes}$ are immediate:

Proposition 3.70

1. For all $\mathcal{P}_\Lambda \subseteq \overline{\mathcal{P}roc}^\Lambda$, $\bowtie\mathcal{P}_\Lambda \sqsubseteq \uplus\mathcal{P}_\Lambda$.
2. For all $P_\Lambda, Q_\Lambda \in \overline{\mathcal{P}roc}^\Lambda$ such that $\tau P_\Lambda = \tau Q_\Lambda$: $P_\Lambda \mathbin{\boxtimes} Q_\Lambda$ iff $P_\Lambda \sqsupseteq Q_\Lambda$. □

It is also useful to show:

Proposition 3.71 Let $T_n, V_n \in \mathit{Tree}$ for $n \geqslant 0$. Then

$$\bowtie\{(T_n, \rho V_n) \mid n \geqslant 0\} \sqsupseteq (\bigcup\{T_n \mid n \geqslant 0\}, \rho\bigcup\{V_n \mid n \geqslant 0\})$$

Proof. Let Q denote the left-hand side above. Then $\tau Q = \bigcup\{T_n \mid n \geqslant 0\}$ and, for $s \in \tau Q$:

$$\rho Q(s) = \bigcap\{\rho V_n(s) \mid n \geqslant 0, s \in T_n\} \supseteq \bigcap\{\rho V_n(s) \mid n \geqslant 0\} = \rho(\bigcup\{V_n \mid n \geqslant 0\})(s)$$

where the last equality is justified by Proposition 3.13. □

This can be generalized to tuples:

Proposition 3.72 Let $T_n, V_n \in \mathit{Tree}^\Lambda$ for $n \geqslant 0$. Then

$$\bigotimes\{T_n \times \rho V_n \mid n \geqslant 0\} \sqsupseteq \bigsqcup\{T_n \mid n \geqslant 0\} \times \rho \bigsqcup\{V_n \mid n \geqslant 0\}$$

Proof. Define the sets:

$$\mathcal{P} = \{T_n \times \rho V_n \mid n \geqslant 0\} \qquad \mathcal{T} = \{T_n \mid n \geqslant 0\} \qquad \mathcal{V} = \{V_n \mid n \geqslant 0\}$$

Then the proposition follows if, for all $\lambda \in \Lambda$, $(\bigotimes \mathcal{P})_\lambda \sqsupseteq ((\bigsqcup \mathcal{T})_\lambda, (\rho \bigsqcup \mathcal{V})_\lambda)$, i.e. if for all $\lambda \in \Lambda$:

$$\bigotimes \mathcal{P}_\lambda \sqsupseteq (\bigcup \mathcal{T}_\lambda, \rho(\bigcup \mathcal{V}_\lambda)) \tag{1}$$

Indeed, for $\lambda \in \Lambda$, letting $T_{\lambda,n} = (T_n)_\lambda$, $V_{\lambda,n} = (V_n)_\lambda$ yields:

$$\mathcal{P}_\lambda = \{(T_{\lambda,n}, \rho V_{\lambda,n}) \mid n \geqslant 0\} \qquad \mathcal{T}_\lambda = \{T_{\lambda,n} \mid n \geqslant 0\} \qquad \mathcal{V}_\lambda = \{V_{\lambda,n} \mid n \geqslant 0\}$$

and (1) is ensured by applying, for each $\lambda \in \Lambda$, the previous proposition, which yields:

$$\bigotimes\{(T_{\lambda,n}, \rho V_{\lambda,n}) \mid n \geqslant 0\} \sqsupseteq (\bigcup\{T_{\lambda,n} \mid n \geqslant 0\}, \rho \bigcup\{V_{\lambda,n} \mid n \geqslant 0\}) \qquad \square$$

Specification-Oriented Predicates

The predicate ψ occurring in the induction rules introduced later on must be *specification-oriented*, in the following sense (different from that of [Olderog & Hoare, 1986]):

1. if ψ holds for a nondeterministic choice over a set, then it also holds for every member of the set (this is quite natural for a specification); equivalently, it can be required that validity of ψ is preserved by making its argument more deterministic; this is also expressed by saying that ψ is $\sqsubseteq$-closed;

2. ψ should be *admissible* both for process chains and semi-process $\boxtimes$-chains.

Formally:

Definition 3.73 A predicate $\psi\colon \mathcal{P}\mathit{roc}^\Lambda \to \mathit{Bool}$ is *specification-oriented* if:

1. if $P_\Lambda, Q_\Lambda \in \overline{\mathcal{P}roc}^\Lambda$, $P_\Lambda \sqsubseteq Q_\Lambda$, then $\psi(Q_\Lambda)$ implies $\psi(P_\Lambda)$;

2. if $\mathcal{P}_\Lambda$ is a chain over $Proc^\Lambda$ and $\psi(P)$ holds for all $P \in \mathcal{P}_\Lambda$, then $\psi(\biguplus \mathcal{P}_\Lambda)$ holds;

3. if $\mathcal{P}_\Lambda$ is a $\overline{\boxtimes}$-chain over $\overline{\mathcal{P}roc}^\Lambda$ and $\psi(P)$ holds for all $P \in \mathcal{P}_\Lambda$, then $\psi(\bigotimes \mathcal{P}_\Lambda)$ holds too. □

It is now important to show that interesting specification-oriented predicates do exist. The basic specification-oriented predicate is dealt with in:

Proposition 3.74 For any $Q \in \overline{\mathcal{P}roc}$, the predicate $\psi_Q : \overline{\mathcal{P}roc} \to \mathcal{B}ool$ defined by $\psi_Q(P) = P \sqsubseteq Q$ is specification-oriented.

Proof. The proof that ψ is $\sqsubseteq$-closed is immediate. Admissibility is a consequence of another easy fact: given any subset $\mathcal{P}$ of $Proc$ such that $P \sqsubseteq Q$ for all $P \in \mathcal{P}$, it follows that $\biguplus \mathcal{P} \sqsubseteq Q$ (because Q is an upper bound of $\mathcal{P}$ and $\biguplus \mathcal{P}$ is the lub). Finally, $\overline{\boxtimes}$-admissibility follows from the latter fact and $\bigotimes \mathcal{P} \sqsubseteq \biguplus \mathcal{P}$. □

Another specification-oriented predicate is $\psi_S(P)$ which holds if the predicate S is satisfied by every trace and refusal of P. This predicate serves to give a meaning to the formulae of the **sat** language introduced in Chapter 4.

Proposition 3.75 Let $S : Act^* \times \mathsf{p}Act^+ \to \mathcal{B}ool$ be a predicate over failures. The predicate $\psi_S : \overline{\mathcal{P}roc} \to \mathcal{B}ool$ defined by:

$$\psi_S(P) = \text{true} \quad \text{iff} \quad \text{for all } t \in \tau P \text{ and } X \in \rho P(t),\ S(t, X) = \text{true}$$

is specification-oriented.

Proof. Consider the pair

$$\begin{aligned} \tau Q &= \{t \mid \exists X \in \mathsf{p}Act^+ : S(t, X) = \text{true}\} \\ \rho Q(s) &= \{X \in \mathsf{p}Act^+ \mid S(s, X) = \text{true}\} \end{aligned}$$

Though Q may not be a semi-process, $\psi_S(P)$ is true iff $P \sqsubseteq Q$, which may be shown to be specification-oriented with the same reasoning as in Proposition 3.74. □

With some licence, in the following the operators $\wedge, \vee, \Rightarrow$ are applied to predicates to denote the boolean connectives 'and', 'or' and 'implication' respectively.

A specification-oriented predicate over a process tuple can be obtained from the conjunction of specification-oriented predicates over each process in the tuple.

Proposition 3.76 For all $\lambda \in \Lambda$, let $\psi_\lambda \colon \overline{\mathcal{P}roc} \to \mathit{Bool}$ be a specification-oriented predicate. Then $\psi : \overline{\mathcal{P}roc}^\Lambda \to \mathit{Bool}$ defined by $\psi(P_\Lambda) = \bigwedge\{\psi_\lambda(P_\lambda) \mid \lambda \in \Lambda\}$ is specification-oriented.

Proof. Proving that ψ is $\sqsubseteq$-closed is immediate.

For admissibility, let $\mathcal{P}_\Lambda$ be a chain over Proc^Λ, $\psi(P_\Lambda) = \text{true}$ for all $P_\Lambda \in \mathcal{P}_\Lambda$. By definition of ψ, it follows that, for $\lambda \in \Lambda$ and $P \in \mathcal{P}_\lambda$, $\psi_\lambda(P)$ holds. Since $\mathcal{P}_\lambda$ is a process chain for every $\lambda \in \Lambda$, the admissibility of ψ_λ implies that $\psi_\lambda(\biguplus \mathcal{P}_\lambda)$ holds. Thus, by definitions of ψ and $\biguplus$ for a process tuple, $\psi(\biguplus \mathcal{P}_\Lambda)$ holds too.

The $\overline{\sqsubseteq}$-admissibility proof is similar. □

Finally, it is worth noting that the conjunction and disjunction of specification-oriented predicates is specification oriented.

Proposition 3.77 The following predicates over $\overline{\mathcal{P}roc}^\Lambda$ are specification oriented:

1. $\bigwedge_{i \in I} \psi_i(P_\Lambda)$ if, for every $i \in I$, $\psi_i \colon \overline{\mathcal{P}roc}^\Lambda \to \mathit{Bool}$ is specification oriented;
2. $\psi_1(P_\Lambda) \vee \psi_2(P_\Lambda)$ if, for $i = 1, 2$, $\psi_i \colon \overline{\mathcal{P}roc}^\Lambda \to \mathit{Bool}$ is specification oriented.

Proof. Showing that in both cases the relevant predicate is $\sqsubseteq$-closed is easy.

That admissibility is preserved by conjunction and finite disjunction is a standard result [Loeckx & Sieber, 1987]. □

Other Fixpoint Induction Rules

We are now ready to formulate an induction rule that is easier to use than Theorem 3.59, in that the basis ($\psi(\bot_F)$) is replaced by a condition whose proof does not require induction.

Theorem 3.78 Let $F : \overline{\mathcal{P}roc}^\Lambda \to \overline{\mathcal{P}roc}^\Lambda$ be trace-based and process-preserving; suppose its restriction to process tuples is continuous. Let $\psi \colon \overline{\mathcal{P}roc}^\Lambda \to \mathit{Bool}$ be a specification-oriented predicate. If:

1. $\psi(\bullet_\Lambda \times \rho F_\tau^{\,k}(\bullet_\Lambda))$ holds for some $k \geqslant 0$ (basis);
2. for all $P_\Lambda \in Proc^\Lambda$, if $\psi(P_\Lambda)$ holds, then $\psi(F(P_\Lambda))$ holds (ψ is F-inductive).

Then $\psi(\mathrm{fix}\, F)$ holds.

Proof. Because of Theorem 3.59, it is only necessary to prove that $\psi(\bot_F)$ holds. First, note that $\{F_\tau^{\,n}(\bullet_\Lambda) \mid n \geqslant 0\}$ is a chain of trace tuples (cf. Definition 3.48). Induction on n is now used to show that, for all $n \geqslant 0$,

$$\psi(F_\tau^{\,n}(\bullet_\Lambda) \times \rho F_\tau^{\,k+n}(\bullet_\Lambda)). \tag{1}$$

The basis is true by hypothesis. For the step, assume as induction hypothesis that (1) holds; then, since ψ is F-inductive, it follows that $\psi(F(F_\tau^{\,n}(\bullet_\Lambda) \times \rho F_\tau^{\,k+n}(\bullet_\Lambda)))$. Since F is trace-based and ψ is $\sqsubseteq$-closed, ψ must also hold for the argument $F_\tau^{\,n+1}(\bullet_\Lambda) \times \rho F_\tau^{\,k+n+1}(\bullet_\Lambda)$.

The set

$$\{F_\tau^{\,n}(\bullet_\Lambda) \times \rho F_\tau^{\,k+n}(\bullet_\Lambda) \mid n \geqslant 0\}$$

is a $\boxtimes$-chain, because $\{F_\tau^{\,n}(\bullet_\Lambda) \mid n \geqslant 0\}$ is a chain of trace tuples, and by Proposition 3.13. Thus:

$\psi(\bigotimes\{F_\tau^{\,n}(\bullet_\Lambda) \times \rho F_\tau^{\,k+n}(\bullet_\Lambda) \mid n \geqslant 0\})$ from (1), by ψ $\boxtimes$-admissible

$\psi(\bigsqcup\{F_\tau^{\,n}(\bullet_\Lambda) \mid n \geqslant 0\} \times \rho \bigsqcup\{F_\tau^{\,k+n}(\bullet_\Lambda) \mid n \geqslant 0\})$ by Prop. 3.72, ψ $\sqsubseteq$-closed

In order to show that the argument of ψ in the latter condition is actually $\bot_F$ it suffices to observe that

$$\bigsqcup\{F_\tau^{\,n}(\bullet_\Lambda) \mid n \geqslant 0\} = \bigsqcup\{F_\tau^{\,k+n}(\bullet_\Lambda) \mid n \geqslant 0\}$$

because $\{F_\tau^{\,n}(\bullet_\Lambda) \mid n \geqslant 0\}$ is a chain. □

The basis of the previous induction rule may be replaced by one which is slightly stronger, but handier in practice because only F, not F_τ, appears in it:

Theorem 3.79 Let $F : \overline{Proc}^\Lambda \to \overline{Proc}^\Lambda$ be trace-based and process-preserving; suppose its restriction to process tuples is continuous. Let $\psi : \overline{Proc}^\Lambda \to Bool$ be a specification-oriented predicate. If:

1. $\psi(\langle(\tau STOP, \rho(F^{h(\lambda)}(STOP_\Lambda))_\lambda) \mid \lambda \in \Lambda\rangle)$ holds for some function $h\colon \Lambda \to Nat$ bounded by some k (basis);

2. for all $P_\Lambda \in Proc^\Lambda$, if $\psi(P_\Lambda)$ holds, then $\psi(F(P_\Lambda))$ holds (ψ is F-inductive).

Then $\psi(\mathrm{fix}\, F)$ holds.

Proof. Since F is trace-based and monotonic, it can be shown by induction, for all $h \geqslant 0$:

$$F^h(STOP_\Lambda) = F^h(\bullet_\Lambda \times \rho\bullet_\Lambda) \sqsupseteq {F_\tau}^h(\bullet_\Lambda) \times \rho{F_\tau}^h(\bullet_\Lambda)$$

whence, for all $\lambda \in \Lambda$:

$$(\bullet, \rho(F^{h(\lambda)}(STOP_\Lambda))_\lambda) \;\sqsupseteq\; (\bullet, \rho({F_\tau}^{h(\lambda)}(\bullet_\Lambda))_\lambda).$$

Hence, since $h(\lambda) \leqslant k$ and $\{{F_\tau}^n(\bullet_\Lambda) \mid n \geqslant 0\}$ is a chain of trace tuples, recalling Proposition 3.13 it follows:

$$\langle(\tau STOP, \rho(F^{h(\lambda)}(STOP_\Lambda))_\lambda) \mid \lambda \in \Lambda\rangle \;\sqsupseteq\; \bullet_\Lambda \times \rho{F_\tau}^k(\bullet_\Lambda).$$

Hence, by the basis and since ψ is $\sqsubseteq$-closed:

$$\psi(\bullet_\Lambda \times \rho{F_\tau}^k(\bullet_\Lambda))$$

from which the theorem follows by Theorem 3.78. □

Remark 3.80 The latter induction rule may be even more useful when the predicate $\psi(P_\Lambda)$ is of the form $\bigwedge_{\lambda\in\Lambda} \psi_{S_\lambda}(P_\lambda)$, i.e.:

$$\bigwedge_{\lambda\in\Lambda} \bigwedge_{t\in\tau P_\lambda} \bigwedge_{X\in\rho P_\lambda(t)} S_\lambda(t, X)$$

For then the basis need not mention semi-processes:

$$\begin{aligned}
&\textstyle\bigwedge_{\lambda\in\Lambda} \bigwedge_{t\in\bullet} \bigwedge_{X\in\rho(F^{h(\lambda)}(STOP_\Lambda))_\lambda(t)} S_\lambda(t, X) \;=\\
&\textstyle\bigwedge_{\lambda\in\Lambda} \bigwedge_{t\in\tau(F^{h(\lambda)}(STOP_\Lambda))_\lambda(t)} \bigwedge_{X\in\rho(F^{h(\lambda)}(STOP_\Lambda))_\lambda(t)} t = \langle\rangle \Rightarrow S_\lambda(t, X) \;=\\
&\textstyle\bigwedge_{\lambda\in\Lambda} \psi_{(t=\langle\rangle\Rightarrow S_\lambda(t,X))}((F^{h(\lambda)}(STOP_\Lambda))_\lambda)
\end{aligned}$$

□

The next induction rule has the least demanding basis, but only applies when the function F_τ has a unique fixpoint. This is not always the case, as for, e.g., $F_\tau(T) = T \cup aT$. Actually, F must also be bi-continuous, i.e. continuous also with respect to a partial order and lub which are dual of $\sqsubseteq$ and $\bigsqcup$ (in that they are obtained by replacing $\subseteq$ with $\supseteq$ and $\cup$ with $\cap$ everywhere). However, most interesting process function are indeed bi-continuous (see [Brookes, Hoare, & Roscoe, 1984] for a treatment of this dual kind of continuity).

The rule is best preceded by a lemma. In both, let $\bigcap_\Lambda$ stand for component-wise intersection: $\bigcap_\Lambda \mathcal{X}_\Lambda = \langle \bigcap \mathcal{X}_\lambda \mid \lambda \in \Lambda \rangle$ and define $*$ and $*_\Lambda$ by:

$$* = Act^* \qquad\qquad (*_\Lambda)_\lambda = *$$

Note that $\rho*(s) = \{\varnothing\}$ for all s.

Lemma 3.81 Let $F : \bar{\mathcal{P}}roc^\Lambda \to \bar{\mathcal{P}}roc^\Lambda$ be trace-based and process-preserving; suppose F_τ is bi-continuous and has a unique fixpoint. Then

$$\mathcal{P} \;=\; \{F_\tau{}^n(\bullet_\Lambda) \times \rho F_\tau{}^n(*_\Lambda) \mid n \geqslant 0\}$$

is a chain and $\bigsqcup \mathcal{P} \sqsupseteq \bot_F$.

Proof. By the monotonicity of F_τ (Proposition 3.47), which entails (by induction) for all n:

$$F_\tau{}^n(\bullet_\Lambda) \sqsubseteq F_\tau{}^{n+1}(\bullet_\Lambda) \qquad\qquad F_\tau{}^n(*_\Lambda) \sqsupseteq F_\tau{}^{n+1}(*_\Lambda)$$

and Proposition 3.13, the set $\mathcal{P}$ is a chain.

It is possible to establish:

$$(1) \qquad \bigsqcup \mathcal{P} \;\sqsupseteq\; \bigsqcup \{F_\tau{}^n(\bullet_\Lambda) \mid n \geqslant 0\} \times \rho \bigcap_\Lambda \{F_\tau{}^n(*_\Lambda) \mid n \geqslant 0\}$$

This relationship will be proved for each λ-component. Let $T_n = F_\tau{}^n(\bullet_\Lambda)$, $V_n = F_\tau{}^n(*_\Lambda)$. Then for every $\lambda \in \Lambda$:

$$\begin{aligned}(\tau \bigsqcup \mathcal{P})_\lambda \;&=\; \tau \uplus \mathcal{P}_\lambda \;=\; \bigcup\{\tau P \mid P \in \mathcal{P}_\lambda\} \;=\; \bigcup\{\tau P_\lambda \mid P_\Lambda \in \mathcal{P}\} \\ &=\; \bigcup\{(T_n)_\lambda \mid n \geqslant 0\} \;=\; (\bigsqcup\{T_n \mid n \geqslant 0\})_\lambda\end{aligned}$$

as desired. Moreover, let s be in the latter trace set, and in particular let m be the minimum index such that $s \in (T_m)_\lambda$. Then:

$$\begin{aligned}
\rho(\bigsqcup \mathcal{P})_\lambda(s) &= \rho(\biguplus \mathcal{P}_\lambda)(s) \\
&\supseteq \bigcup\{\rho P(s) \mid P \in \mathcal{P}_\lambda,\ s \in \tau P\} \\
&= \bigcup\{\rho P_\lambda(s) \mid P_\Lambda \in \mathcal{P},\ s \in \tau P_\lambda\} \\
&= \bigcup\{\rho(V_n)_\lambda(s) \mid n \geqslant 0, s \in (T_n)_\lambda\} \\
&= \bigcup\{\rho(V_n)_\lambda(s) \mid n \geqslant m\} && \text{by } T_k \sqsubseteq T_{k+1},\ k \geqslant 0 \\
&= \rho(\bigcap\{(V_n)_\lambda \mid n \geqslant m\})(s) && \text{by Proposition 3.13} \\
&= \rho(\bigcap\{(V_n)_\lambda \mid n \geqslant 0\})(s) && \text{by } V_k \sqsupseteq V_{k+1},\ k \geqslant 0 \\
&= \rho(\bigcap_\Lambda\{V_n \mid n \geqslant 0\})_\lambda(s)
\end{aligned}$$

Since F_τ is bi-continuous, $\bigcap_\Lambda\{{F_\tau}^n(*_\Lambda) \mid n \geqslant 0\}$ is, by duality, a fixpoint of F_τ and must coincide, by hypothesis, with $\bigsqcup\{{F_\tau}^n(\bullet_\Lambda) \mid n \geqslant 0\}$. Thus the right-hand side of (1) is indeed $\bot_F$. □

Theorem 3.82 Let $F : \overline{\mathcal{P}}roc^\Lambda \to \overline{\mathcal{P}}roc^\Lambda$ be trace-based and process-preserving; suppose its restriction to process tuples is bi-continuous and F_τ has a unique fixpoint. Assume the predicate $\psi : \overline{\mathcal{P}}roc^\Lambda \to \mathit{Bool}$ is $\sqsubseteq$-closed and admissible for chains over $\overline{\mathcal{P}}roc^\Lambda$. If:

1. $\psi(\bullet_\Lambda \times \rho *_\Lambda)$ holds (basis);
2. for all $P_\Lambda \in \mathit{Proc}^\Lambda$, if $\psi(P_\Lambda)$ holds, then $\psi(F(P_\Lambda))$ holds (ψ is F-inductive).

Then $\psi(\mathrm{fix}\, F)$ holds.

Proof. Since F is trace-based and monotonic, it can be shown from the basis, by induction, that for all $n \geqslant 0$:

$$\psi({F_\tau}^n(\bullet_\Lambda) \times \rho {F_\tau}^n(*_\Lambda)) \tag{1}$$

By the previous lemma $\mathcal{P} = \{{F_\tau}^n(\bullet_\Lambda) \times \rho {F_\tau}^n(*_\Lambda) \mid n \geqslant 0\}$ is a chain. Thus, from (1) and since ψ is $\sqsubseteq$-admissible, $\psi(\bigsqcup \mathcal{P})$ follows, whence, again by the previous lemma and since ψ is $\sqsubseteq$-closed, $\psi(\bot_F)$ follows. This suffices by Theorem 3.59. □

3.3 Basic Process Functions

According to the flexible scheme introduced in Section 2.4.3, two steps are necessary in order to give the language $\mathrm{PL}^{\oplus}_{\uplus}$ a denotational semantics with *Proc* as semantic domain. The first is to make each language operator denote a process function (or operation) with domain and range in accordance with the operator arity. The second is to show that $\mathrm{PL}^{\oplus}_{\uplus}$-generated process functions possess a fixpoint; by Remark 3.53, this is ensured if the basic process functions employed in the construction of $\mathrm{PL}^{\oplus}_{\uplus}$-generated ones are trace-based, and continuous or monotonic. Basic process functions comprise, besides those denoted by operators, extraction of a component from a tuple (cf. equation (2-2), Definition 2.3), and are treated in the remainder of this section.

In the light of the fixpoint rules presented and of Remark 3.61, it will also be examined whether basic functions, which can all be viewed also as functions over semi-processes, are well-behaved (continuous, monotonic,[1] trace-based, process-preserving and refusal-based, both for processes and—where applicable—semi-processes).

3.3.1 Component Extraction

For each $\lambda \in \Lambda$, the function $(\,)_\lambda : \mathit{Proc}^\Lambda \to \mathit{Proc}$ is defined by $(P_\Lambda)_\lambda = P_\lambda$, for $P_\Lambda \in \mathit{Proc}^\Lambda$. It is straightforward to see that these functions must be well-behaved for every $\lambda \in \Lambda$.

3.3.2 Stop

The process expression *stop* is associated with the well-behaved constant function having as value *STOP*. This process can be characterized, in accordance with its Definition 3.16, by the relations:

$$\tau STOP = \{\langle\rangle\} \qquad\qquad \rho STOP(\langle\rangle) = \mathsf{p}Act$$

[1] While continuity is only needed for process functions, its proof will always be applicable also to semi-process extensions, which yields monotonicity for these extensions too, as demanded by Remark 3.61.

3.3.3 Operational Justification

The definition of the remaining process functions can be justified in terms of the transitional definition of the corresponding operators. Ideally, consistency should be required between the compositional denotational semantics $\mathcal{O}[\![\ ]\!]$ induced by the basic functions, and the transitional semantics based on the operator transition rules. Thus, one would hope that, for all process expression p in $\mathrm{PL}^{\oplus}_{\uplus}$:

$$\mathcal{O}[\![p]\!] = (\tau p, \rho p) \tag{3-2}$$

where τ, ρ are operationally defined (overloading them once more) as follows:

Definition 3.83 For all process expression p:

$$\begin{aligned} \tau p &= \{s \in Act^* \mid p \stackrel{s}{\Longrightarrow}\} \\ \rho p(s) &= \{X \in \mathsf{p}Act^+ \mid \exists q\colon p \stackrel{s}{\Longrightarrow} q \;\&\; \forall x \in X\colon q \stackrel{x}{\not\Longrightarrow}\} \quad (\text{for } s \in \tau P) \end{aligned}$$ □

Thus, informally, τp is defined as the set of traces that p may perform according to the derivation relation $\Longrightarrow$, and $X \in \rho p(s)$ is defined to hold if, after performing s, p may reach a state whence it is unable to continue with any trace in X.

In fact, for the technical reasons examined in Section 3.3.9, the consistency equation (3-2) can only hold if the relation $\Longrightarrow$ is image-finite; even so a proof is rather hard and better omitted, as done in [Brookes, 1983a] for the ordinary failures case (see [Barrett, 1991] for the proof of a similar result). We will settle instead for a weaker justification: if op is an n-ary operator, it will be shown that the relation defining $op(P_1, \ldots, P_n)$ in terms of its operands $P_1, \ldots, P_n$ also holds between $(\tau\, op(p_1, \ldots, p_n),\ \rho\, op(p_1, \ldots, p_n))$ and $(\tau p_1, \rho p_1), \ldots, (\tau p_n, \rho p_n)$, for all process expressions $p_1, \ldots, p_n$.

3.3.4 Non-Deterministic Choice

Binary non-deterministic choice is a specialization of nondeterministic choice over a process set (Definition 3.22):

$$P \uplus Q = \biguplus\{P, Q\}$$

Previous results about the $\biguplus$ operator ensure that binary $\uplus$ is well-defined. It is immediate to prove that $\uplus$ is distributive in each argument, hence (by Theorems 3.44 and 3.40), continuous and monotonic. Furthermore, it is also process-preserving, weakly trace-based and trace-based (owing to Proposition 3.65). In short, the operation $\uplus$ is well-behaved.

It is not difficult to extend the previous results to a function $\biguplus_\Lambda : Proc^\Lambda \to Proc$.[2]

The operational justification for this operation is represented by the following result.

Proposition 3.84 For all process expressions p and q:

$$\tau(p \uplus q) = \tau p \cup \tau q$$

$$\rho(p \uplus q)(s) = \begin{cases} \rho p(s) \cup \rho q(s) & \text{if } s \in \tau p \cap \tau q \\ \rho p(s) & \text{if } s \in \tau p - \tau q \\ \rho q(s) & \text{if } s \in \tau q - \tau p \end{cases}$$

Proof. By a property of $\stackrel{s}{\Longrightarrow}$, $p \uplus q \stackrel{s}{\Longrightarrow} r$ iff $p \stackrel{s}{\Longrightarrow} r$ or $q \stackrel{s}{\Longrightarrow} r$. Thus the first equality above is obviously true.

For the second equality, suppose $s \in \tau p \cup \tau q$.

Then $X \in \rho(p \uplus q)(s)$ holds iff, for some r, $p \uplus q \stackrel{s}{\Longrightarrow} r$ and $r \stackrel{x}{\not\Longrightarrow}$ for all $x \in X$. Without loss of generality, suppose $p \stackrel{s}{\Longrightarrow} r$: then $X \in \rho p(s)$, hence X is in the right hand side of the second equality.

Conversely, if X is in the right hand side, assume without loss of generality that $s \in \tau p$ and $X \in \rho p(s)$. Then for some r, $p \stackrel{s}{\Longrightarrow} r$ and $r \stackrel{x}{\not\Longrightarrow}$ for all $x \in X$. Since also $p \uplus q \stackrel{s}{\Longrightarrow} r$, it follows that $X \in \rho(p \uplus q)(s)$. □

3.3.5 Deterministic Choice

Deterministic choice is the binary process operation defined by:

$$\tau(P \oplus Q) = \tau P \cup \tau Q$$

[2] It is worth noting that, in contrast to the $\biguplus$ operation over sets of processes or process tuples, $\biguplus_\Lambda$ applies to process tuples of $Proc^\Lambda$ and (by pointwise extension) to sets of such tuples. Thus, for $\mathcal{P}_\Lambda \subseteq Proc^\Lambda$, there is a difference between $\biguplus_\Lambda \mathcal{P}_\Lambda = \{\biguplus_\Lambda P_\Lambda \mid P_\Lambda \in \mathcal{P}_\Lambda\}$ and $\biguplus \mathcal{P}_\Lambda = \langle \biguplus \mathcal{P}_\lambda \mid \lambda \in \Lambda \rangle$ (cf. Definition 3.36): the former is a process set, the latter a process tuple.

$$\begin{aligned}
\rho(P \oplus Q)(\langle\rangle) &= \rho P(\langle\rangle) \cap \rho Q(\langle\rangle) \\
\rho(P \oplus Q)(s) &= \begin{cases} \rho P(s) \cup \rho Q(s) & \text{if } s \in \tau P \cap \tau Q \\ \rho P(s) & \text{if } s \in \tau P - \tau Q \\ \rho Q(s) & \text{if } s \in \tau Q - \tau P \end{cases} \quad \text{for } s \neq \langle\rangle
\end{aligned}$$

To see that this operation is well-defined and process-preserving, we can adapt the analogous result about non-deterministic choice:

Proposition 3.85 If P, Q are (semi-)processes, so is $P \oplus Q$.

Proof. The proofs that $\tau(P \oplus Q)$ is a tree and, for $s \neq \langle\rangle$, $\rho(P \oplus Q)$ is a (consistent) refusal function are the same as for the $\uplus$ operation.

To complete the proof, assume $X \in \rho(P \oplus Q)(\langle\rangle)$. Then $X \in \rho P(\langle\rangle)$ and $X \in \rho Q(\langle\rangle)$ hold. If $Y \subseteq X$, then $Y \in \rho P(\langle\rangle)$ and $Y \in \rho Q(\langle\rangle)$, whence $Y \in \rho(P \oplus Q)(\langle\rangle)$. Likewise, if $t \in X$, for all w, $X \cup \{tw\} \in \rho P(\langle\rangle)$ and $X \cup \{tw\} \in \rho Q(\langle\rangle)$, whence $X \cup \{tw\} \in \rho(P \oplus Q)(\langle\rangle)$. Thus we have proved the subset and suffix closure properties of $P \oplus Q$.

To prove consistency, let $X \cup \{t\} \notin \rho(P \oplus Q)(\langle\rangle)$. Without loss of generality, let then $X \cup \{t\} \notin \rho P(\langle\rangle)$. Consistency of $\tau P, \rho P$ implies $t \in \tau P$ and $X \div t \in \rho P(t)$, so $t \in \tau(P \oplus Q)$ and $X \div t \in \rho(P \oplus Q)(t)$. □

The operational justification for this operation is represented by the following result.

Proposition 3.86 For all process expressions p and q:

$$\begin{aligned}
\tau(p \oplus q) &= \tau p \cup \tau q \\
\rho(p \oplus q)(\langle\rangle) &= \rho p(\langle\rangle) \cap \rho q(\langle\rangle) \\
\rho(p \oplus q)(s) &= \begin{cases} \rho p(s) \cup \rho q(s) & \text{if } s \in \tau p \cap \tau q \\ \rho p(s) & \text{if } s \in \tau p - \tau q \\ \rho q(s) & \text{if } s \in \tau q - \tau p \end{cases} \quad \text{for } s \neq \langle\rangle
\end{aligned}$$

Proof. The proof of the corresponding result for the $\uplus$ operation can be readily adapted for the first and third equality.

For the second equality, $X \in \rho(p \oplus q)(\langle\rangle)$ iff, for some r, $r \stackrel{x}{\not\Longrightarrow}$ for all $x \in X$ and $p \oplus q \stackrel{\langle\rangle}{\Longrightarrow} r$. By the properties of $\Longrightarrow$, this can be the case iff $r \equiv p' \oplus q'$, $p \stackrel{\langle\rangle}{\Longrightarrow} p'$,

$q \stackrel{\langle\rangle}{\Longrightarrow} q'$ and, for all $x \in X$, $p' \stackrel{x}{\not\Longrightarrow}$ and $q' \stackrel{x}{\not\Longrightarrow}$. This is equivalent to $X \in \rho p(\langle\rangle)$ and $X \in \rho q(\langle\rangle)$. □

Deterministic choice is distributive in each argument, hence (by Theorems 3.44 and 3.40) continuous and monotonic:

Proposition 3.87 For all process set $\mathcal{P}$, $(\uplus \mathcal{P}) \oplus Q = \uplus\{P \oplus Q \mid P \in \mathcal{P}\}$.

Proof. It is easy to see that L, the left-hand side of the above equality, and R, the right-hand side, have the same trace sets. We now prove equality of refusal functions.

Suppose $X \in \rho L(\langle\rangle)$. Then $X \in \rho\mathcal{P}(\langle\rangle)$ and $X \in \rho Q(\langle\rangle)$. So, there exists $P \in \mathcal{P}$ such that $X \in \rho P(\langle\rangle)$. This implies $X \in \rho(P \oplus Q)(\langle\rangle)$, hence $X \in \rho R(\langle\rangle)$. This argument can be easily reversed to show $\rho R(\langle\rangle) \subseteq \rho L(\langle\rangle)$.

Let now $s \neq \langle\rangle$ and $s \in \tau L = \tau R$. $X \in \rho L(s)$ holds iff $X \in \rho Q(s)$ and $s \in \tau Q$ or, for some $P \in \mathcal{P}$, $X \in \rho P(s)$ and $s \in \tau P$. All together this is equivalent to stating that for some $P \in \mathcal{P}$, $X \in \rho(P \oplus Q)(s)$ and $s \in \tau(P \oplus Q)$. Finally, the latter requirement means that $X \in \rho R(s)$. □

To complete showing that this operation is well-behaved, we prove:

Proposition 3.88 The function $\oplus\colon \overline{\mathcal{P}}roc^2 \to \overline{\mathcal{P}}roc$ is trace-based.

Proof. Clearly, $\oplus$ is weakly trace-based. We only need to check that for all trees T_1, V_1, T_2, V_2 such that $T_1 \subseteq V_1$ and $T_2 \subseteq V_2$:

$$(T_1, \rho V_1) \oplus (T_2, \rho V_2) \sqsupseteq (T_1 \cup T_2, \rho(V_1 \cup V_2))$$

For this purpose, suppose $s \in T_1 \cup T_2$ and $X \in \rho(V_1 \cup V_2)(s)$. Then by Proposition 3.13, $X \in \rho V_1(s)$ and $X \in \rho V_2(s)$. Thus, whether $s = \langle\rangle$ or not, $X \in \rho((T_1, \rho V_1) \oplus (T_2, \rho V_2))(s)$. □

3.3.6 Multiple Action Sequence

For any (possibly infinite) non-empty subset $A \subseteq Act$, multiple action sequence $A;$ is a function that maps every tuple $P_A \in \overline{\mathcal{P}}roc^A$ onto A; P_A defined by:

$$\tau(A; P_A) \;=\; \{\langle\rangle\} \cup \bigcup_{a \in A} a(\tau P_a)$$

$$\begin{aligned}\rho(A;P_A)(\langle\rangle) &= \{X \in \mathsf{p}Act^+ \mid \forall a \in A : X \div a \in \rho P_a(\langle\rangle)\}\\ \rho(A;P_A)(as) &= \rho P_a(s), \text{ for } a \in A\end{aligned}$$

It should be noted that if $a \in X \cap A$ then $\langle\rangle \in X \div a$, which ensures that $X \div a \notin \rho P_a(\langle\rangle)$ and, as intuition demands, $X \notin \rho(A;P_A)(\langle\rangle)$. Moreover, if no $x \in X$ starts with any $a \in A$ (i.e. if $X \div a = \varnothing$ for all $a \in A$), then certainly $X \in \rho(A;P_A)(\langle\rangle)$.

Multiple action sequence is well-defined and process-preserving.

Proposition 3.89 Let $A \subseteq Act$, $P_A \in \overline{\ }Proc^A$ (resp. $P_A \in Proc^A$). Then $A;P_A \in \overline{\ }Proc$ (resp. $P_A \in Proc$).

Proof. The arguments for $\tau(A;P_A)$ and, with $s \neq \langle\rangle$, $\rho(A;P_A)(s)$ are straightforward.

We consider therefore $\rho(A;P_A)(\langle\rangle)$, and assume X is in this set. By definition of $A;P_A$, $X \div a \in \rho P_a(\langle\rangle)$ for $a \in A$.

If $X' \subseteq X$, then, for $a \in A$, $X' \div a \subseteq X \div a$, hence $X' \div a \in \rho P_a(\langle\rangle)$ by the subset-closure property of ρP_a. It follows that $X' \in \rho(A;P_A)(\langle\rangle)$ (subset-closure of $\rho(A;P_A)$).

Let $t = bs \in X$; then $(X \cup \{tw\}) \div a = X \div a$ for all $a \neq b$. Moreover $s \in X \div b$ and, if $b \in A$, the suffix-closure property of ρP_b implies that $(X \div b) \cup \{sw\} \in \rho P_b(\langle\rangle)$ for any w. Hence, using $(X \div b) \cup \{sw\} = (X \cup \{tw\}) \div b$, the suffix-closure of $\rho(A;P_A)$ follows.

Finally, for trace-refusal consistency, suppose $P_A \in Proc^A$ and

(1) $\quad X \cup \{bu\} \notin \rho(A;P_A)(\langle\rangle)$ $(b \in Act)$.

Since $a \neq b$ implies $(X \cup \{bu\}) \div a = X \div a$, then $b \in A$ must hold or (1) would be contradicted; moreover, for the same reason, it must be the case that $((X \cup \{bu\}) \div b) \notin \rho P_b(\langle\rangle)$. Since $(X \cup \{bu\}) \div b = (X \div b) \cup \{u\}$, consistency of $\tau P_b, \rho P_b$ entails $u \in \tau P_b$ and $(X \div b) \div u \in \rho P_b(u)$. Hence, easily, $bu \in \tau(A;P_A)$ and, using $(X \div b) \div u = X \div bu$, $X \div bu \in \rho(A;P_A)(bu)$. □

The operational motivation for multiple action sequence is provided by:

Proposition 3.90 Let $\varnothing \subset A \subseteq Act$. For every tuple $p_A = \langle p_a \mid a \in A \rangle$ of process expressions:

$$\begin{aligned} \tau(A; p_A) &= \{\langle\rangle\} \cup \bigcup_{a \in A} a(\tau p_a) \\ \rho(A; p_A)(\langle\rangle) &= \{X \in \mathsf{p}Act^+ \mid \forall a \in A : X \div a \in \rho p_a(\langle\rangle)\} \\ \rho(A; p_A)(as) &= \rho p_a(s), \text{ for } a \in A \end{aligned}$$

Proof. We prove only the less straightforward second equality.

Let $X \in \rho(A; p_A)(\langle\rangle)$. Choose then a process expression q such that $A; p_A \stackrel{\langle\rangle}{\Longrightarrow} q$ and, for all $x \in X$, $q \stackrel{x}{\not\Longrightarrow}$; by the properties of $\Longrightarrow$, there exists a process tuple q_A such that $q = A; q_A$ and $p_a \stackrel{\langle\rangle}{\Longrightarrow} q_a$, for all $a \in A$. For every $a \in A$ and $z \in X \div a$, $q_a \stackrel{z}{\not\Longrightarrow}$, otherwise $q \stackrel{az}{\Longrightarrow}$, a contradiction with $az \in X$. Thus $X \div a \in \rho p_a(\langle\rangle)$.

Let now X be a member of the right hand side of the second equality in the proposition statement. For all $a \in A$, choose q_a such that $p_a \stackrel{\langle\rangle}{\Longrightarrow} q_a$ and, for all $z \in (X \div a)$, $q_a \stackrel{z}{\not\Longrightarrow}$. Hence, $A; p_A \stackrel{\langle\rangle}{\Longrightarrow} A; q_A$. To conclude the proof it suffices to show that, for all $bz \in X$, $A; q_A \stackrel{bz}{\not\Longrightarrow}$. This is obvious for $b \notin A$; for $b \in A$, $A; q_A \stackrel{bz}{\Longrightarrow}$ would imply (by the properties of $\Longrightarrow$) $q_b \stackrel{z}{\Longrightarrow}$: a contradiction, because $z \in X \div a$ (since $bz \in X$). □

For the process function $A;: Proc^A \to Proc$ proving continuity is rather hard:

Proposition 3.91 Multiple action sequence is continuous.

Proof. Let $\mathcal{P}_A$ be a chain in $Proc^A$. We must show:

$$A; \bigsqcup \mathcal{P}_A = \bigsqcup \{A; P_A \mid P_A \in \mathcal{P}_A\} \tag{1}$$

Let L and R denote the left and right hand sides of (1). The difficult case is proving $\rho L(\langle\rangle) = \rho R(\langle\rangle)$.

For this purpose, assume $X \in \rho L(\langle\rangle)$. Thus:

$$\text{for all } a \in A,\ X \div a \in \rho(\bigsqcup \mathcal{P}_A)_a(\langle\rangle) = \bigcup \{\rho(P_A)_a(\langle\rangle) \mid P_A \in \mathcal{P}_A\} \tag{2}$$

Let $B = \{a \in A \mid X \div a \neq \varnothing\}$. B must be finite or X would be infinite. For every $b \in B$, by (2) we can choose $Q_A{}^{(b)} \in \mathcal{P}_A$ such that:

(3) $X \div b \in \rho({Q_A}^{(b)})_b(\langle\rangle)$.

Letting $Q_A = \bigsqcup\{{Q_A}^{(b)} \mid b \in B\}$ entails ${Q_A}^{(b)} \sqsubseteq Q_A$ for all $b \in B$, hence:

(4) for all $b \in B$: $({Q_A}^{(b)})_b \sqsubseteq Q_b$.

Since $\mathcal{P}_A$ is a chain, Q_A is the lub of a finite chain in $\mathcal{P}_A$ and must itself be in $\mathcal{P}_A$. Moreover Q_A must satisfy $X \div a \in \rho Q_a(\langle\rangle)$ for all $a \in A$; this follows if a is a $b \in B$ from (3) and (4), for $a \notin B$ from $X \div a = \varnothing$ (definition of B). Thus $X \in \rho(A; Q_A)(\langle\rangle)$ and, by definition of $\bigsqcup$ (going back to (1)), $X \in \rho R(\langle\rangle)$.

Conversely, assume $X \in \rho R(\langle\rangle)$. Then, for some $Q_A \in \mathcal{P}_A$, $X \in \rho(A; Q_A)(\langle\rangle)$ i.e. $X \div a \in \rho Q_a(\langle\rangle)$ for all $a \in A$. Hence for all $a \in A$:

$$X \div a \in \bigcup\{\rho P_a(\langle\rangle) \mid P_A \in \mathcal{P}_A\} = \rho(\bigsqcup \mathcal{P}_A)_a(\langle\rangle)$$

which entails $X \in \rho(A; (\bigsqcup \mathcal{P}_A))(\langle\rangle)$. □

Finally, note that multiple action sequence is weakly trace-based and refusal-based, hence trace-based.

3.3.7 Renaming

Let $f : Act \to Act$ be an injective function, which is extended to traces and trace sets in the obvious way. The renaming process function is defined by:

$$\begin{aligned} \tau P[f] &= f(\tau P) \\ \rho(P[f])(f(s)) &= f(\rho P(s)) \quad (s \in \tau P) \end{aligned}$$

For brevity this operator has been made less general than that of [Brookes, Hoare, & Roscoe, 1984], but it is sufficient for most practical applications. Since $f : Act \to Act$ is injective, it is straightforward to check that the process function f is well-defined, consistent with the derivation relation $\Longrightarrow$, distributive (hence continuous and monotonic) and generally well-behaved.

3.3.8 Parallel Composition

Let $B_1, \ldots B_N$ be non-empty action sets and $P_1, \ldots, P_N$ be processes ($N \geqslant 0$). The parallel composition $\|_{n=1}^{N} P_n{:}B_n$ of $P_1, \ldots, P_N$ over $B_1, \ldots, B_N$ is defined by:

$$\begin{aligned}
\tau(\|_{n=1}^{N} P_n{:}B_n) &= \|_{n=1}^{N} \tau P_n{:}B_n \\
\rho(\|_{n=1}^{N} P_n{:}B_n)(s) &= \{X \in \mathsf{p}Act^+ \mid \exists X_1 \in \rho P_1(s\restriction B_1), \ldots, X_N \in \rho P_N(s\restriction B_N) : \\
&\qquad X \cap (\cup_{n=1}^{N} B_n)^* \subseteq \overline{\|}_{n=1}^{N} X_n{:}B_n\}
\end{aligned}$$

where the operators $\|$ and $\overline{\|}$ are defined, for trace sets $V_1, \ldots, V_N$:

$$\begin{aligned}
\|_{n=1}^{N} V_n{:}B_n &= \{u \in (\bigcup_{n=1}^{N} B_n)^* \mid \forall 1 \leqslant n \leqslant N : u\restriction B_n \in V_n\} \\
\overline{\|}_{n=1}^{N} V_n{:}B_n &= \{u \in (\bigcup_{n=1}^{N} B_n)^* \mid \exists 1 \leqslant n \leqslant N : u\restriction B_n \in V_n\}
\end{aligned}$$

The function $\|_{n=1}^{N} P_n{:}B_n$ will also be written as $\|(P_1{:}B_1, \ldots, P_N{:}B_N)$. The same applies to $\overline{\|}$.

First, we prove that parallel composition is well-defined.

Proposition 3.92 For all non-empty action sets $B_1, \ldots, B_N$, if $P_1, \ldots P_N$ are (semi-)processes, so is $\|_{n=1}^{N} P_n{:}B_n$.

Proof. Let $B = \cup_{n=1}^{N} B_n$. It is straightforward to realize that $\tau \|_{n=1}^{N} P_n{:}B_n$ is a tree. To see that $\rho \|_{n=1}^{N} P_n{:}B_n$ is a refusal function, let $s \in \tau \|_{n=1}^{N} P_n{:}B_n$ and $X \in \rho(\|_{n=1}^{N} P_n{:}B_n)(s)$. Thus, $s \in B^*$ and, for $1 \leqslant m \leqslant N$, $s\restriction B_n \in \tau P_n$. Moreover we can choose $X_1 \in \rho P_1(s\restriction B_1), \ldots, X_N \in \rho P_N(s\restriction B_N)$ such that $X \cap B^* \subseteq \overline{\|}_{n=1}^{N} X_n{:}B_n$. For subset-closure, suppose $Y \subseteq X$. Then $Y \cap B^* \subseteq \overline{\|}_{n=1}^{N} X_n{:}B_n$, whence it follows $Y \in \rho(\|_{n=1}^{N} P_n{:}B_n)(s)$.

For suffix-closure, let $t \in X$ and w be a trace. If $tw \notin B^*$, then $(X \cup \{tw\}) \cap B^* = X \cap B^*$, whence $(X \cup \{tw\}) \in \rho(\|_{n=1}^{N} P_n{:}B_n)(s)$. Otherwise, let $tw \in B^*$, so that $t \in X \cap B^* \subseteq \overline{\|}_{n=1}^{N} X_n{:}B_n$, whence, for some m, $t\restriction B_m \in X_m$. Letting $X'_m{=}X_m \cup \{tw\restriction B_m\}$ and $X'_n = X_n$ for $n \neq m$, it follows $(X \cup \{tw\}) \cap B^* \subseteq \overline{\|}_{n=1}^{N} X'_n{:}B_n$. Finally, by the suffix-closure property of ρP_m, $X'_m \in \rho P_m(s\restriction B_m)$, whence $X \cup \{tw\} \in \rho(\|_{n=1}^{N} P_n{:}B_n)(s)$, as desired.

Finally, for consistency of trace and refusals, let $X \cup \{t\} \notin \rho(\|_{n=1}^{N} P_n{:}B_n)(s)$. To avoid contradicting this assumption, two conditions must hold: (i) $t \in B^*$ (or $(X \cup \{t\}) \cap B^* = X \cap B^*$), and (ii) for all n, $X_n \cup \{t\restriction B_n\} \notin \rho P_n(s\restriction B_n)$. Thus, by consistency of $\tau P_n, \rho P_n$:

(1) for $n \in \{1,\ldots,N\}$, $(st)\restriction B_n \in \tau P_n$;

(2) for $n \in \{1,\ldots,N\}$, $X_n \div (t\restriction B_n) \in \rho P_n(st\restriction B_n)$.

Hence consistency of $\tau P, \rho P$ follows: $st \in \tau P$ from (1); $X \div t \in \rho(\|_{n=1}^{N} P_n{:}B_n)(st)$ from the conjunction of (2) and

(3) $(X \div t) \cap B^* \subseteq \overline{\|}_{n=1}^{N}(X_n \div (t\restriction B_n)) : B_n$

which is inferred thus: $w \in (X \div t) \cap B^*$ implies $tw \in X \cap B^*$, so, for some $m \in \{1,\ldots,N\}$, $tw\restriction B_m \in X_m$, i.e. $w\restriction B_m \in X_m \div (t\restriction B_m)$. □

To verify that the definition of parallel composition is operationally satisfactory, we prove:

Proposition 3.93 Let $B_1,\ldots B_N$ be non-empty action sets and $p_1,\ldots,p_N$ be process expressions ($N \geqslant 0$). Then

$$\begin{aligned}
\tau(\mathop{\|}_{n=1}^{N} p_n{:}B_n) &= \mathop{\|}_{n=1}^{N} \tau p_n{:}B_n \\
\rho(\mathop{\|}_{n=1}^{N} p_n{:}B_n)(s) &= \{X \in \mathsf{p}Act^+ \mid \exists X_1 \in \rho p_1(s\restriction B_1),\ldots,X_N \in \rho p_N(s\restriction B_N) : \\
&\qquad X \cap (\textstyle\bigcup_{n=1}^{N} B_n)^* \subseteq \overline{\|}_{n=1}^{N} X_n{:}B_n\}
\end{aligned}$$

Proof. The first equality is straightforward. For the second, let $B = \bigcup_{n=1}^{N} B_n$, $s \in \tau(\|_{n=1}^{N} p_n{:}B_n)$.

Let $X \in \rho(\|_{n=1}^{N} p_n{:}B_n)(s)$ and choose q such that $\|_{n=1}^{N} p_n{:}B_n \stackrel{s}{\Longrightarrow} q$ and $q \stackrel{x}{\not\Longrightarrow}$ for all $x \in X$. By the properties of $\stackrel{s}{\Longrightarrow}$, there exist $q_1,\ldots,q_N$ such that $q = \|_{n=1}^{N} q_n{:}B_n$ and $p_n \stackrel{s\restriction B_n}{\Longrightarrow} q_n$ for all n. For $1 \leqslant n \leqslant N$, let

$$X_n = \{x\restriction B_n \mid x \in X, q_n \stackrel{x\restriction B_n}{\not\Longrightarrow}\}$$

Note that $X_n \in \rho p_n(s\restriction B_n)$. Let $x \in X \cap B^*$: since $q \stackrel{x}{\not\Longrightarrow}$, it follows that $q_m \stackrel{x\restriction B_m}{\not\Longrightarrow}$ for some m such that $1 \leqslant m \leqslant N$. Thus $x\restriction B_m \in X_m$, $x \in \overline{\|}_{n=1}^{N} X_n{:}B_n$ and we

may conclude that X belongs to the right hand side of the second equality in the proposition statement.

To prove the converse, suppose now that, for $1 \leqslant n \leqslant N$, $X_n \in \rho p_n(s \restriction B_n)$. Choose then q_n such that $p_n \stackrel{s \restriction B_n}{\Longrightarrow} q_n$ and

(1) $$q_n \stackrel{z}{\not\Longrightarrow} \text{ for all } z \in X_n$$

By definition of $\Longrightarrow$, $\|_{n=1}^{N} p_n{:}B_n \stackrel{s}{\Longrightarrow} \|_{n=1}^{N} q_n{:}B_n$. Let X be finite and $X \cap B^* \subseteq \overline{\|}_{n=1}^{N} X_n{:}B_n$. If $x \in X \cap B^*$, choose then m such that $x \restriction B_m \in X_m$; it follows $\|_{n=1}^{N} q_n{:}B_n \stackrel{x}{\not\Longrightarrow}$ or else $q_m \stackrel{x \restriction B_m}{\Longrightarrow}$, contradicting (1). If instead $x \in X - B^*$, $\|_{n=1}^{N} q_n{:}B_n \stackrel{x}{\not\Longrightarrow}$ is an immediate consequence of the way $\Longrightarrow$ is defined for $\|$. In both cases we have that $X \in \rho(\|_{n=1}^{N} p_n{:}B_n)(s)$. □

It is easy to recognize that parallel composition is refusal-based and trace-based.

Commutativity and Associativity

It is obvious that parallel composition is commutative, in the sense that the assignment of the indices $1, \ldots, N$ to the composed processes is immaterial. This operation is also associative, in the sense represented by the following:

Proposition 3.94 Let $B_1, \ldots, B_N, B_{N+1}, \ldots, B_{N+M}$ be non-empty action sets and $P_1, \ldots, P_N, P_{N+1}, \ldots, P_{N+M}$ be processes $(N, M \geqslant 0)$. Then

$$\|_{k=1}^{N+M} P_k{:}B_k \;=\; \|(\,(\|_{n=1}^{N} P_n{:}B_n){:}(\bigcup_{n=1}^{N} B_n),\; P_{N+1}{:}B_{N+1},\; \ldots,\; P_{N+M}{:}B_{N+M}\,)$$

Proof. Let L and R denote the left and right hand side of the above equality. That $\tau L = \tau R$ is a consequence of the associativity of the $\|$ operator on trace sets. We now prove that, for $s \in \tau L = \tau R$, $\rho L(s) = \rho R(s)$. Below, it is convenient to let $B = \bigcup_{n=1}^{N} B_n$.

Suppose $X \in \rho L(s)$. Then there exist $X_1, \ldots, X_N, X_{N+1}, \ldots, X_{N+M}$ such that $X_k \in \rho P_k(s \restriction B_k)$ for all k and

$$\begin{aligned} X \cap (\bigcup_{k=1}^{N+M} B_k)^* \;&\subseteq\; \overline{\|}_{k=1}^{N+M} X_k{:}B_k \\ &=\; \overline{\|}(\,(\overline{\|}_{n=1}^{N} X_n{:}B_n){:}B,\; X_{N+1}{:}B_{N+1},\; \ldots,\; X_{N+M}{:}B_{N+M}) \end{aligned}$$

where the equality holds by associativity of $\overline{\|}$ and entails $X \in \rho R(s)$ because, by definition of parallel composition:

$$\overline{\|}_{n=1}^{N} X_n{:}B_n \;\in\; \rho(\|_{n=1}^{N} P_n{:}B_n)(s\restriction B)$$

To complete the proof, suppose $X \in \rho R(s)$. Then there exist

$$(1) \qquad Y \in \rho(\|_{n=1}^{N} P_n{:}B_n)(s\restriction B) \qquad\qquad X_k \in \rho P_k(s\restriction B_k) \quad (N{+}1 \leqslant k \leqslant N{+}M)$$

such that

$$(2) \qquad X \cap (\bigcup_{k=1}^{N+M} B_k)^* \;\subseteq\; \overline{\|}(Y{:}B, X_{N+1}{:}B_{N+1}, \ldots, X_{N+M}{:}B_{N+M})$$

By definition of parallel composition and (1), there exist $X_1, \ldots, X_N$ such that $X_k \in \rho P_k(s\restriction B_k)$ for all k and

$$(3) \qquad Y \cap B^* \;\subseteq\; \overline{\|}_{n=1}^{N} X_n{:}B_n$$

To infer the desired $X \in \rho L(s)$ it suffices to show that

$$(4) \qquad X \cap (\bigcup_{k=1}^{N+M} B_k)^* \;\subseteq\; \overline{\|}_{k=1}^{N+M} X_k{:}B_k$$

For this purpose, assume $x \in X$ and $x \in (\bigcup_{k=1}^{N+M} B_k)^*$. Then, by (2) either $x\restriction B_k \in X_k$ for some $k \in \{N{+}1, \ldots, N{+}M\}$ or $x\restriction B \in Y$; in the latter case (3) implies that $x\restriction B \in \overline{\|}_{n=1}^{N} X_n{:}B_n$, whence $x\restriction B_k \in X_k$ for some $k \in \{1, \ldots, N\}$. This shows $x \in \overline{\|}_{k=1}^{N+M} X_k{:}B_k$, hence (4). □

Binary Parallel Composition

By repeated application of the associativity Property 3.94 with $N = 2$, a parallel composition of arbitrary arity may be expressed with its binary version. This allows the remaining process properties to be studied for binary parallel composition only. As customary, the binary operator $\|$ is written infix.

This operation is distributive in either argument, hence continuous and monotonic:

Proposition 3.95 Parallel composition is distributive.

Proof. Let B, C be non-empty sets of actions. Let $Q \in \mathit{Proc}$ and $\mathcal{P} \subseteq \mathit{Proc}$. It is necessary to prove:

$$(\biguplus \mathcal{P}):B \parallel Q:C = \biguplus\{P:B \parallel Q:C \mid P \in \mathcal{P}\}$$

Let L and R denote as usual the left and right hand sides of the above equality. The proof that $\tau L = \tau R$ is easy. We show that, for $s \in \tau L = \tau R$, $\rho L(s) = \rho R(s)$.

If $X \in \rho L(s)$, then $X \cap (B \cup C)^* \subseteq Y:B \mid\mid Z:C$ for some $Z \in \rho Q(s\restriction C)$ and $Y \in \rho(\biguplus \mathcal{P})(s\restriction B)$. Thus there exists $P \in \mathcal{P}$ such that $s\restriction B \in \tau P$ and $Y \in \rho P(s\restriction B)$. It follows that $s \in \tau(P:B \parallel Q:C)$ and $X \in \rho(P:B \parallel Q:C)(s)$, hence $X \in \rho R(s)$.

This argument is easily reversed to prove the converse inclusion. □

We conclude with an important condition that allows the action set of a parallel composition to be enlarged without affecting the result:

Lemma 3.96 For all $P, Q \in \mathit{Proc}$, $P:B \parallel Q:C = P:B' \parallel Q:C$ if $\tau P \subseteq B^*$, $B \subseteq B'$ and $(B' - B) \cap C = \varnothing$.

Proof. We let L and R denote the left and right side of the desired equality, and begin with a useful, if obvious, fact:

(1) if $s\restriction B' \in B^*$ or $s \in (B \cup C)^*$, then $s\restriction B = s\restriction B'$.

Now if $s \in \tau L$, then $s \in (B \cup C)^*$, so (1) yields $s \in \tau R$. Conversely, if $s \in \tau R$, then $s\restriction B' \in \tau P \subseteq B^*$ and again (1) yields $s \in \tau L$.

We must now prove that, for $s \in \tau L = \tau R$, $\rho L(s) = \rho R(s)$. The proof is split into two parts.

First, assume $X \in \rho R(s)$. To prove $X \in \rho L(s)$ it is enough to derive, for $Y, Z \in \mathsf{p}\mathit{Act}^+$:

(2) If $X \cap (B' \cup C)^* \subseteq Y:B' \mid\mid Z:C$, then $X \cap (B \cup C)^* \subseteq Y:B \mid\mid Z:C$.

To show (2), let $x \in X \cap (B \cup C)^*$; then, assuming the premise of (2), $x \in Y:B' \mid\mid Z:C$. Thus, $x\restriction C \in Z$ or $x\restriction B' \in Y$ hold; in the latter case (by $x \in (B \cup C)^*$ and (1)) $x\restriction B \in Y$; in both cases, then, $x \in Y:B \mid\mid Z:C$, which proves the conclusion of (2).

Before showing the converse, observe that:

(3) if $w \in \tau P$ and $Y \in \rho P(w)$, then $Y \cup Y' \in \rho P(w)$, for all $Y' \in \mathsf{p}Act^+ - B^*$.

This holds by consistency of $\tau P, \rho P$ and $wY' \cap \tau P = \varnothing$, inferred from $\tau P \subseteq B^*$.

We now assume $X \in \rho L(s)$. To prove $X \in \rho R(s)$, we use (3) and show that, for $Y, Z \in Act^+$:

(4) if $X \cap (B \cup C)^* \subseteq Y{:}B \,\overline{\|}\, Z{:}C$, then, for some $Y' \in \mathsf{p}Act^+ - B^*$,
$X \cap (B' \cup C)^* \subseteq (Y \cup Y'){:}B' \,\overline{\|}\, Z{:}C$.

This last obligation is proved by induction on the size of X. The basis is easy. For the step, let $|X| = n+1$, $X = \{x\} \cup X_0$, $|X_0| = n$. Assume the premise of (4) for X, so that also $X_0 \cap (B \cup C)^* \subseteq Y{:}B \,\overline{\|}\, Z{:}C$. By induction hypothesis, there is $Y_0 \in \mathsf{p}Act^+ - B^*$ for which $X_0 \cap (B' \cup C)^* \subseteq (Y \cup Y_0){:}B' \,\overline{\|}\, Z{:}C$. We look for $Y_1 \in \mathsf{p}Act^+ - B^*$ such that

$$(X_0 \cup \{x\}) \cap (B' \cup C)^* \subseteq (Y \cup Y_1){:}B' \,\overline{\|}\, Z{:}C$$

Y_1 is determined by case analysis on x. The interesting case is $x \in (B' \cup C)^*$, $x\restriction B' \notin Y \cup Y_0$, $x\restriction C \notin Z$ (otherwise just take $Y_1 = Y_0$). There are two subcases: (1) $x \in (B \cup C)^*$, then $x\restriction B \in Y$ by the assumed premise of (4), and we take $Y_1 = Y_0$; (2) $x \notin (B \cup C)^*$, then $x\restriction B' \notin B^*$, and we take $Y_1 = Y_0 \cup \{x\restriction B'\}$. □

3.3.9 Hiding

Let B be a (possibly infinite) set of actions. The hiding operation is defined to map any $P \in \bar{\mathcal{P}}roc$ onto $P\backslash B$ defined by:

$$\begin{aligned} \tau(P\backslash B) &= (\tau P)\backslash B \\ \rho(P\backslash B)(s) &= \{X \in \mathsf{p}Act^+ \mid \exists t \in \tau P : t\backslash B = s \;\&\; \forall Y \in \mathsf{p}(X/B) : Y \in \rho P(t)\} \\ X/B &= \{u \in Act^* \mid u\backslash B \in X\} \quad (X \in \mathsf{p}Act^+) \end{aligned}$$

Hiding is well defined:

Proposition 3.97 If P is a (semi-)process, so is $P\backslash B$ for an action set B.

Proof. It is easy to check that $\tau(P\backslash B)$ is a tree. To prove that $\rho(P\backslash B)$ is a refusal function, assume $s \in \tau(P\backslash B)$ and $X \in \rho(P\backslash B)(s)$. Choose then t such that:

(1) $t \in \tau P$, $t\backslash B = s$, and, for $Y \in \mathrm{p}Act^+$, $Y\backslash B \subseteq X$ implies $Y \in \rho P(t)$.

Subset-closure is easy to show. For suffix-closure, we assume $x \in X$ and prove that $X \cup \{xw\} \in \rho(P\backslash B)(s)$ for any trace w. For this, we let $Y \in \mathrm{p}Act^+$ and $Y\backslash B \subseteq X \cup \{xw\}$, and try to derive $Y \in \rho P(t)$. Let $Y' = \{y \in Y \mid y\backslash B \in X\}$. Clearly $Y'\backslash B \subseteq X$, so $Y' \in \rho P(t)$ from (1). If $y \in Y - Y'$, then $y\backslash B = xw$, so for some y', z: $y = y'z$, $y'\backslash B = x$. Since $y' \in Y'$, $Y' \cup \{y'z\} \in \rho P(t)$ (by suffix-closure of ρP) i.e. $Y' \cup \{y\} \in \rho P(t)$. As the number of traces like y in $Y - Y'$ is finite, the same argument may be applied finitely many times (through induction) to infer $Y \in \rho P(t)$.

Finally, we show consistency between the traces and refusals of $P\backslash B$. Assume $X \cup \{w\} \notin \rho(P\backslash B)(s)$. Then, by (1), there are $Y_X, Y_w \in \mathrm{p}Act^+$ such that: $Y_X\backslash B \subseteq X$, $Y_w\backslash B = \{w\}$ and $Y_X \cup Y_w \notin \rho P(t)$. Let

$$\Xi = \{Y \in \mathrm{p}Act^+ \mid Y \supseteq Y_X, Y\backslash B \subseteq X\}$$

Suppose $Y \in \Xi$. Then $Y \in \rho P(t)$ (by (1)) and $Y \cup Y_w \notin \rho P(t)$ (by subset-closure). Hence, by consistency and subset-closure of $\tau P, \rho P$, there exists u_Y such that:

(2) $u_Y \in Y_w$, $tu_Y \in \tau P$ and $Y \div u_Y \in \rho P(tu_Y)$

By (2), as Y varies in Ξ, u_Y can only take on the finitely many values of Y_w. Then Lemma 2.11 ensures that there exists u such that:

(3) $u \in Y_w$, and for all $Y \in \Xi$ there is $Y' \in \Xi$ such that $Y' \supseteq Y$ and $u_{Y'} = u$.

From (3) and (2) $tu \in \tau P$ follows, whence (from (1) by $u \in Y_w$) $sw \in \tau(P\backslash B)$. To conclude the proof, it is enough to show that, if $Z \in \mathrm{p}Act^+$ and $Z\backslash B \subseteq X \div w$, then $Z \in \rho P(tu)$. Let then $Y = uZ \cup Y_X$. From (3), by the definitions of Y_w, Y_X, we infer $Y \in \Xi$, so there is Y' as in (3) and, from (2) and (3), it follows $Y' \div u \in \rho P(tu)$. Hence $Z \in \rho P(tu)$ follows by $Z \subseteq Y' \div u$, which is proved thus: $z \in Z$ implies $uz \in Y$; by (3) $Y \subseteq Y'$ holds; hence $z \in Y' \div u$. □

The usual operational justification does not hold, in general, for hiding. As a counterexample, consider the process constants:

$$\kappa := b; \kappa_1 \qquad\qquad \kappa_n := c^n; a; stop \oplus b; \kappa_{n+1} \quad (n > 0)$$

(where the notation c^n is self-explanatory) and let q be $\kappa\backslash b$. Then $\{a\} \notin \rho(q\backslash c)(\langle\rangle)$, because $q\backslash c \stackrel{\langle\rangle}{\Longrightarrow} q'\backslash c$ implies $q'\backslash c \stackrel{a}{\Longrightarrow}$. However:

$$\{a\} \in \{X \in \mathsf{p}Act^+ \mid \exists t \in \tau q\colon t\backslash c = \langle\rangle \;\&\; \forall Y \in \mathsf{p}Act^+\colon Y\backslash c \subseteq X \Rightarrow Y \in \rho q(t)\}$$

To show this, we assume $Y \in \mathsf{p}Act^+$, $Y\backslash c \subseteq \{a\}$ and derive $Y \in \rho(q)(\langle\rangle)$. Elements of Y must be of the form $c^k a c^h$ for some $k, h \geqslant 0$. Let $n-1$ be the maximum such k; then $\kappa \stackrel{b^n}{\Longrightarrow} \kappa_n$, so $q{=}(\kappa\backslash b) \stackrel{\langle\rangle}{\Longrightarrow} (\kappa_n\backslash b)$ and $(\kappa_n\backslash b) \stackrel{c^k a}{\not\Longrightarrow}$ for $k < n$.

The problem with q above is that $\stackrel{\langle\rangle}{\Longrightarrow}$ is not image-finite (i.e. $q \stackrel{\langle\rangle}{\Longrightarrow} q'$ for infinitely many q'). Note that this constraint may be acceptable in many practical cases; in particular it does not rule out, in the construction of q, the use of multiple action sequence $(A;)$ with A infinite. Image finiteness allows a useful lemma to be proved.

Lemma 3.98 Let t be a trace, $Z \subseteq Act^+$ and p a process expression such that $p \stackrel{t}{\Longrightarrow}$ is image-finite. Suppose that for all $Y \in \mathsf{p}Z$ there is p_Y such that $p \stackrel{t}{\Longrightarrow} p_Y$ and, for all $y \in Y$, $p_Y \stackrel{y}{\not\Longrightarrow}$.

Then there exists p_Z such that $p \stackrel{t}{\Longrightarrow} p_Z$ and, for all $z \in Z$, $p_Z \stackrel{z}{\not\Longrightarrow}$.

Proof. Let $\{q \mid p \stackrel{t}{\Longrightarrow} q\} = \{p_1, \ldots, p_k\}$ (by the hypothesis, $k > 0$). By contradiction, suppose that for all $h \in \{1, \ldots, k\}$ there is $z_h \in Z$ such that $p_h \stackrel{z_h}{\Longrightarrow}$. Then, taking $Y = \{z_1, \ldots, z_k\}$ contradicts the hypothesis. □

We can now prove an operational justification for hiding:

Proposition 3.99 Let B be a set of actions and p a process expression such that $p \stackrel{t}{\Longrightarrow}$ is image-finite for all t. Then:

$$\begin{aligned} \tau(p\backslash B) &= (\tau p)\backslash B \\ \rho(p\backslash B)(s) &= \{X \in \mathsf{p}Act^+ \mid \exists t \in \tau p\colon t\backslash B = s \;\&\; \forall Y \in \mathsf{p}(X/B)\colon Y \in \rho p(t)\} \end{aligned}$$

Proof. The first equality is easy. For the second, let $s \in \tau(p\backslash B)$.

If $X \in \rho(p\backslash B)(s)$, then $p\backslash B \stackrel{s}{\Longrightarrow} q'$ for some q' and $q' \stackrel{x}{\not\Longrightarrow}$ for all $x \in X$. By the properties of $\Longrightarrow$, for some t, p': $s = t\backslash B$, $q' = p'\backslash B$ and $p \stackrel{t}{\Longrightarrow} p'$, so $t \in \tau p$. Let now $Y\backslash B \subseteq X$, Y finite. For all $y \in Y$, $p' \stackrel{y}{\not\Longrightarrow}$ or else $q' \stackrel{y\backslash B}{\Longrightarrow}$, a contradiction because $y\backslash B \in X$. Thus $Y \in \rho p(t)$.

Suppose now that X belongs to the right hand side of the second equality required for the proposition. Thus there exists $t \in \tau p$ such that $t\backslash B = s$ and, for all $Y \in \mathsf{p}(X/B)$, $Y \in \rho p(t)$; so for all $y \in Y$ there exists p_Y such that $p \stackrel{t}{\Longrightarrow} p_Y$ and $p_Y \stackrel{y}{\not\Longrightarrow}$. The previous lemma may now be invoked to infer that there exists q such that $p \stackrel{t}{\Longrightarrow} q$ and $q \stackrel{z}{\not\Longrightarrow}$ for all $z \in X/B$. Thus $p\backslash B \stackrel{s}{\Longrightarrow} q\backslash B$ and, if $x \in X$, $q\backslash B \stackrel{x}{\not\Longrightarrow}$ (or $q \stackrel{z}{\Longrightarrow}$ for some $z \in X/B$). This amounts to $X \in \rho(p\backslash B)(s)$, as required. □

The hiding operator is not continuous, but is weakly trace-based and monotonic. While this is sufficient for the language definition (cf. Remark 3.53), it does not permit application of the various fixpoint induction rules developed. It must be said, however, that use of hiding in a recursively defined process constant hardly ever occurs in practice. It is also worth noting that the loss of continuity is the price paid to allow the hidden action set to be infinite. Thus, the reasons adduced in Section 2.6.1 for ordinary failures hiding might be repeated here to argue that this infinitary hiding operator is crucial for practical applications. Here, however, the additional issue of associativity has still to be considered.

Associativity of Hiding

The hiding operation introduced is not associative. The relevant counterexample is clearly related to that devised for the operational justification (with slight abuse, process P below is introduced inductively, rather than as a fixpoint):

$$P = b; P_1 \qquad\qquad P_n = c^n; a; STOP \oplus b; P_{n+1} \quad (n > 0)$$

It is not difficult to check that

$$\{a\} \in \rho(P\backslash b\backslash c)(\langle\rangle), \quad \text{but } \{a\} \notin \rho(P\backslash c\backslash b)(\langle\rangle) = \rho(P\backslash\{c,b\})(\langle\rangle)$$

the argument is surprisingly similar to that employed for the operational justification counterexample. In fact, all the problems with hiding (the imperfect operational justification and the lack of associativity and continuity) seem facets of the same limitation: the finiteness of refusal sets; but this is a topic for further study.

Nevertheless, use of the present form of hiding can be justified in various ways. First, it can be shown that associativity does hold for $P\backslash\ldots\backslash\ldots$, provided none of the hiding operations involved makes the $\Longrightarrow$ relation non-image-finite in the LTS corresponding to process P; the class of such processes is large enough for many applications. Another justification is provided by the fact that, for all B, C, $P\backslash B \cup C \sqsubseteq P\backslash B\backslash C$ and $\tau(P\backslash B \cup C) = \tau(P\backslash B\backslash C)$; thus, any verification that consists in proving every extended failure of $P\backslash B\backslash C$ enjoys a certain property will also apply to $P\backslash B\cup C$. Finally, the main argument for this notion of hiding is that it enjoys a weak form of associativity, which applies to a combination of processes by parallel composition followed by hiding of shared actions: it will be shown that combining thus $P_1, \ldots, P_N$ (in this particular order) is the same as combining P_1 with P_2 and the resulting process with the remaining ones (still in the same order). This property supports the study of hierarchical systems.

In the formalization, it is convenient to introduce an operator #, with a syntax similar to that of ||, to model parallel composition followed by hiding.

Definition 3.100 Let the action sets $B_1, \ldots, B_N$ $(N>0)$ be non-empty and triple-disjoint ($B_i \cap B_j \cap B_k = \varnothing$, for $i \neq j \neq k$). Let $\sum_{n=1}^{M} B_n$ denote the disjunct sum of the sets $B_1, \ldots B_M$ $(M \geqslant 1)$. Define:

$$\mathop{\#}_{n=1}^{N} P_n{:}B_n \;=\; (\mathop{\|}_{n=1}^{N} P_n{:}B_n)\backslash B_1 \cap B_2 \backslash \ldots \backslash (\textstyle\sum_{n=1}^{N-1} B_n) \cap B_N \qquad \square$$

Theorem 3.101 For $N > 2$, let $P_1, \ldots, P_N$ be processes and $B_1, \ldots, B_N$ non-empty and triple-disjoint action sets. Then:

$$\mathop{\#}_{n=1}^{N} P_n{:}B_n \;=\; \#(\,(P_1{:}B_1 \,\#\, P_2{:}B_2){:}(B_1{+}B_2),\; P_3{:}B_3,\; \ldots,\; P_N{:}B_N\,)$$

Proof. Note that since action sets are triple-disjoint:

(1) $$B_1 \cap B_2 \cap \bigcup_{n=3}^{N} B_n = \varnothing$$

The main argument is:

$$\#_{n=1}^{N} P_n{:}B_n \;=\; (\|_{n=1}^{N} P_n{:}B_n)\backslash B_1 \cap B_2 \backslash \ldots \backslash (\textstyle\sum_{n=1}^{N-1} B_n) \cap B_N$$

$$= \| \,(\, (P_1{:}B_1 \,\|\, P_2{:}B_2) : (B_1 \cup B_2),\; P_3 : B_3,\; \ldots,\; P_N : B_N$$
$$\,) \,\backslash B_1 \cap B_2 \backslash \ldots \backslash (\textstyle\sum_{n=1}^{N-1} B_n) \cap B_N \quad \text{(by associativity of } \|\text{, Prop. 3.94)}$$

$$= \| \,(\, ((P_1{:}B_1 \,\|\, P_2{:}B_2)\backslash B_1 \cap B_2) : (B_1 \cup B_2),\; P_3 : B_3,\; \ldots,\; P_N : B_N$$
$$\,) \,\backslash (B_1 + B_2) \cap B_3 \backslash \ldots \backslash (\textstyle\sum_{n=1}^{N-1} B_n) \cap B_N$$

(by Prop. 3.103, using (1))

$$= \| \,(\, (P_1{:}B_1 \,\#\, P_2{:}B_2) : (B_1{+}B_2),\; P_3 : B_3,\; \ldots,\; P_N : B_N$$
$$\,) \,\backslash (B_1 + B_2) \cap B_3 \backslash \ldots \backslash (\textstyle\sum_{n=1}^{N-1} B_n) \cap B_N$$

(by definition of #, and (1), Lemma 3.96 to reduce $B_1 \cup B_2$ to $B_1{+}B_2$)

$$= \#(\, (P_1{:}B_1 \,\#\, P_2{:}B_2) : (B_1{+}B_2),\; P_3{:}B_3,\; \ldots,\; P_N{:}B_N \,)$$

Note that the above derivation hides some extra applications of associativity of $\|$ to indices $3, \ldots, N$ in $\|(\ldots, P_3{:}B_3, \ldots, P_N{:}B_N)$. □

Before turning to Proposition 3.103, which encompasses the essence of weak associativity of #, it is convenient to introduce a particular way of merging two traces s and w that agree if we project s on B and conceal the actions of w at E; the result will be denoted by $s \;{}_B{\rhd}_E\; w$, introducing a new merge operator ${}_B{\rhd}_E$:

Lemma 3.102 Let $E \subseteq B \subseteq Act$. For traces s, w such that $w\backslash E = s\restriction B$, the trace $s \;{}_B{\rhd}_E\; w$ is recursively defined by:

$$s \;{}_B{\rhd}_E\; w \;=\; \begin{cases} s & \text{if } w\backslash E = w \\ s'e\,(s'' \;{}_B{\rhd}_E\; w'') & \text{if } w = w'ew'', w'\backslash E = w', e \in E \\ & \text{and } s = s's'', s' = \max\{u \mid u \leqslant s, s'\restriction B = w'\} \end{cases}$$

(Note that above $w''\backslash E = s''\restriction B$, so $s'' \;{}_B{\rhd}_E\; w''$ is well-defined.) Then:

$$s\backslash E = s \qquad (s \;{}_B{\rhd}_E\; w)\backslash E = s \qquad (s \;{}_B{\rhd}_E\; w)\restriction B = w$$

Proof. The first equality follows from $s\restriction E = s\restriction B\restriction E = w\backslash E\restriction E = \langle\rangle$. The last two are proved using induction on the number of elements of E occurring in w. □

It is now possible to prove that hiding can be brought within a parallel composition and applied to one of the composed processes, provided it does not affect actions shared by the other composed process.

Proposition 3.103 Let $E \subseteq B \subseteq Act$, $C \subseteq Act$, $E \cap C = \varnothing$. For all processes P, Q:

$$(P{:}B \parallel Q{:}C)\backslash E \;=\; (P\backslash E){:}B \parallel Q{:}C$$

where the left and right hand side are also denoted by L and R respectively.

Proof. Showing that $\tau L \subseteq \tau R$ is straightforward. The converse inclusion is proved by assuming $s \in \tau R$. Then $s \in (B \cup C)^*$, $s\restriction B \in \tau(P\backslash E)$ and $s\restriction C \in \tau Q$. Choose $w \in \tau P$ such that $w\backslash E = s\restriction B$, and let $z = s \;{}_B\rhd_E\; w$. Then (using Lemma 3.102), $z\restriction B{=}w \in \tau P$ and $z\restriction C{=}(z\backslash E\restriction C){=}s\restriction C \in \tau Q$. It follows that $z \in \tau(P{:}B \parallel Q{:}C)$, whence $z\backslash E{=}s \in \tau L$.

The refusal functions are proved to agree in the two following lemmata. □

Lemma 3.104 Under the assumptions of Proposition 3.103, $\rho R(s) \subseteq \rho L(s)$ for every $s \in \tau L = \tau R$.

Proof. If $X \in \rho R(s)$, by definition of $\parallel$, it is possible to choose $X_B \in \rho(P\backslash E)(s\restriction B)$ and $X_C \in \rho Q(s\restriction C)$ such that:

$$(1) \qquad X \cap (B \cup C)^* \;\subseteq\; X_B{:}B \;\overline{\parallel}\; X_C{:}C$$

By definition of $\backslash$, choose $t \in \tau P$ such that $t\backslash E = s\restriction B$ and, for $Y \in \mathrm{p}Act^+$, if $Y\backslash E \subseteq X_B$ then $Y \in \rho P(t)$. We now show that:

(2) if $Z \in \mathrm{p}Act^+$, $Z\backslash E \subseteq X$, then $\hat{Z} = \{z\restriction B \mid z \in Z \cap (B \cup C)^*, z\restriction C \notin X_C\} \in \rho P(t)$.

For this purpose, we prove $\hat{Z}\backslash E \subseteq X_B$. Let $w \in \hat{Z}\backslash E$; then, for some $u \in \hat{Z}$, $u\backslash E = w$ and, for some $z \in Z \cap (B \cup C)^*$, $z\restriction B = u$ and $z\restriction C \notin X_C$. Clearly $z\backslash E \in (B \cup C)^*$ and, by the assumption $Z\backslash E \subseteq X$, $z\backslash E \in X$; thus (1) implies at least one of two facts: $z\backslash E\restriction C{=}z\restriction C \in X_C$, which has already been excluded, or, as is the case, $z\backslash E\restriction B \in X_B$. The desired $w \in X_B$ then follows because $z\backslash E\restriction B = z\restriction B\backslash E = u\backslash E = w$.

We are now ready to prove that $X \in \rho L(s)$. Let $y = s\ {}_B\triangleright_E\ t$; then:

(3) $$y\backslash E = s \qquad y\restriction B = t \qquad y\restriction C = y\backslash E\restriction C = s\restriction C$$

Now, $y \in \tau(P{:}B \parallel Q{:}C)$ follows from (3) by $t \in \tau P$, $s\restriction C \in \tau Q$ (because $s \in \tau R$). Thus, it is enough to show that, for $Z \in \mathsf{p}Act^+$, $Z\backslash E \subseteq X$ implies $Z \in \rho(P{:}B \parallel Q{:}C)(y)$. This follows from the obvious inclusion $Z \cap (B \cup C)^* \subseteq \hat{Z}{:}B \,\overline{\|}\, X_C{:}C$, where $X_C \in \rho Q(y\restriction C)$ (using (3)), and $\hat{Z} \in \rho P(y\restriction B)$ (by (2), using (3)). □

Lemma 3.105 Under the assumptions of Proposition 3.103, $\rho L(s) \subseteq \rho R(s)$ for every $s \in \tau L = \tau R$.

Proof. If $X \in \rho L(s)$ then, by definition of $\backslash$, there is $t \in \tau(P{:}B \parallel Q{:}C)$ such that $t\backslash E = s$ and $Y \in \rho(P{:}B \parallel Q{:}C)(t)$, for any finite Y satisfying $Y\backslash E \subseteq X$. Thus, $t \in (B \cup C)^*$, $t\restriction B \in \tau P$, $t\restriction C \in \tau Q$; moreover, with every Y as above we may associate $Z_B^Y \in \rho P(t\restriction B)$ and $Z_C^Y \in \rho Q(t\restriction C)$ such that

(1) $$Y \cap (B \cup C)^* \subseteq Z_B^Y{:}B \,\overline{\|}\, Z_C^Y{:}C$$

It is possible to assume $Z_C^Y \subseteq X\restriction C$ (otherwise, exploiting $E \cap C = \varnothing$, just let $Z_C^Y \cap X\restriction C$ be the new Z_C^Y). As a result, as Y varies, Z_C^Y may take on only finitely many values. By Lemma 2.11 this implies that, for some $Z_C \subseteq X\restriction C$:

(2) $$\forall Y' \in \mathsf{p}Act^+ : Y'\backslash E \subseteq X \Rightarrow \exists Y \in \mathsf{p}Act^+ : Y\backslash E \subseteq X,\ Y \supseteq Y',\ Z_C^Y = Z_C$$

Clearly:

$$\begin{aligned} X \cap (B \cup C)^* &\subseteq Z_B{:}B \,\overline{\|}\, Z_C{:}C, \quad \text{where} \\ Z_B &= \{z\restriction B \mid z \in X \cap (B \cup C)^*, z\restriction C \notin Z_C\} \end{aligned}$$

Since $Z_C \in \rho Q(s\restriction C)$ (by $s\restriction C = t\backslash E\restriction C = t\restriction C$), to obtain the desired $X \in \rho R(s)$ it suffices to prove $Z_B \in \rho(P\backslash E)(s\restriction B)$ and $s\restriction B \in \tau P\backslash E$. For the latter, recall that $t\restriction B \in \tau P$ and note $t\restriction B\backslash E = t\backslash E\restriction B = s\restriction B$.

The last proof obligation left can be satisfied by assuming $W \in \mathsf{p}Act^+$, $W\backslash E \subseteq Z_B$ and deriving:

(3) $$W \in \rho P(t\restriction B)$$

For this, note that, if $w \in W$, since $w \backslash E \in Z_B$, it is possible to choose $z_w \in X \cap (B \cup C)^*$ such that $z_w \restriction B = w \backslash E$ and $z_w \restriction C \notin Z_C$; from $(z_w \ {}_B\rhd_E \ w) \backslash E = z_w$, it follows:

$$Y_W \backslash E \subseteq X, \text{ where } Y_W = \{z_w \ {}_B\rhd_E \ w \mid w \in W\}$$

By (2), choose $Y \supseteq Y_W$ such that also $Y \backslash E \subseteq X$ and $Z_C^Y = Z_C$. Since $Z_B^Y \in \rho P(t \restriction B)$, (3) will follow if we show $W \subseteq Z_B^Y$.

Suppose then $w \in W$. Thus

$$(4) \quad (z_w \ {}_B\rhd_E \ w) \in Y_W \subseteq Y \cap (B \cup C)^* \subseteq Z_B^Y{:}B \overline{\|} Z_C^Y{:}C = Z_B^Y{:}B \overline{\|} Z_C{:}C$$

where the second inclusion is an instance of (1). That $(z_w \ {}_B\rhd_E \ w) \restriction C \in Z_C$ can be excluded on the grounds that $(z_w \ {}_B\rhd_E \ w) \restriction C = (z_w \ {}_B\rhd_E \ w) \backslash E \restriction C = z_w \restriction C$. So, by (4), $(z_w \ {}_B\rhd_E \ w) \restriction B \in Z_B^Y$; since $(z_w \ {}_B\rhd_E \ w) \restriction B = w$, the desired $w \in Z_B^Y$ follows. □

Chapter 4

Extended Failures and sat Verification

4.1 About This Chapter

The '**sat**' logic outlined in Section 4.2 may express that a property is satisfied by all the failures of the behaviour denoted by a process expression. A calculus for this logic is briefly presented and, in Section 4.3, illustrated by a first non-trivial example.

Like the standard **sat** logic of [Hoare, 1985], that proposed here can describe both input-output relation, for which process traces alone suffice, and, exploiting also refusals, system-environment interaction in the form of deadlock freedom. In the CSP approach and terminology, the former kind of properties is associated with *partial correctness* and *safety*, the latter with *total correctness* and *liveness*. Rather interestingly, verification proofs of the two kinds have proved, both in the standard logic and ours, fairly orthogonal.

Our **sat** logic and underlying model would appear to represent a major advance over the standard ones in at least two respects: (i) they allow a satisfactory specification of an almost totally unreliable communication medium (Section 4.4), and (ii) they can be successfully applied to the verification of diverging systems.

Section 4.5 is devoted to a larger, medium-sized example: a system made up of two entities carrying out a sliding-window protocol across an unreliable network

layer. The overall system, which represents the core of class 4 ISO Transport layer, is first specified at a reasonable level of detail, then verified within the **sat** calculus, with the help of a suitable set of effective proof techniques. Of these, the main are based upon the so-called process-oriented consistency rules, which—in essence—allow properties of longer refusals to be derived from those of shorter ones (see Section 4.2.4). This affords deadlock-freedom proofs which are only slightly more complicated than those in the standard **sat** calculus, in that they need fixpoint induction only to derive properties of standard (one-action) refusals of recursive processes (cf. Section 4.5.4). As a result, the bulk of deadlock freedom proofs may be carried out before recursive processes have been actually defined (cf. Section 4.5.4); this ensures all the typical advantages of decision postponement, as discussed in Section 4.5.2.

In sum, it seems fair to conclude that the work outlined succeeds in proving the practical applicability of our **sat** calculus.

4.2 Outline of a sat Logic

This section illustrates a **sat** logic and calculus for specification and verification. This logical system is a natural extension of that employed in [Hoare, 1985]. Both belong to the fourth kind identified and evaluated in Section 1.5: a specification is viewed as a property to be satisfied by every possible observation about a system. In the proposed framework, extended failures (instead of failures) are adopted as observations, and the associated process domain supplies a model for the **sat** logic.

Formulae of the **sat** logic have the form p **sat** S, where p is a process expression of $\mathrm{PL}^{\oplus}_{\uplus}$ and S—often referred to as an *assertion*—is a predicate logic formula that may contain the variables *tr* and *Ref*. Informally and with some abuse, p **sat** S is valid if S holds whenever *tr* and *Ref* are replaced, respectively, by any trace t of p and any X refused by p after t.[1] We avoid setting up any formal preliminaries, which are not difficult and coincide with those of the renowned book [Hoare, 1985]. In particular, to understand the following it is unnecessary to define precisely a

[1] Of course, traces and refusals belong in fact to $\mathcal{O}[\![p]\!]$, the denotation of p.

particular syntax or semantics for S: symbols occurring in formulae are assumed to be given a fixed interpretation (which is the usual one for standard symbols, and is introduced informally along the way for new symbols). This justifies the liberty we take to blur the distinction between a formula S and its interpretation as a predicate over Act^*, $\mathsf{p}Act^+$ and possibly other domains; accordingly, metalanguage and logic notation will be freely mixed; e.g. $tr \in Act^*$ may occur in S. (Of course, in a more rigorous setting, such an S would either employ type-predicates or be expressed in a many-sorted logic).

Our **sat** calculus comprises healthiness, operator and fixpoint rules. The former include the *consequence rule* and the other standard ones of [Hoare, 1985] (these will not be repeated here); they will be completed in Section 4.2.4 with the introduction of the novel *process-oriented* rules. Only a presentation of operator and fixpoint rules is provided, in Section 4.2.2 below; their soundness proofs will be omitted, being straightforward consequences of the operator definitions in Section 3.3 and, for the fixpoint rule, of the results in Section 3.2.8[2]. Completeness will be investigated in further studies; here it is just worth observing that the restriction of the calculus to $\mathrm{PL}^{\oplus}_{\uplus}$ without recursion is easily proved complete with the usual techniques, based on the notion of strongest specification.

4.2.1 Consequence Rule

The classical weakening rule of [Hoare, 1985]:

$$\frac{P \textbf{ sat } S,\ S \Rightarrow T}{p \textbf{ sat } T}$$

4.2.2 Operator and Fixpoint Rules

The rules for *stop* and the choice operators are straightforward, practically coinciding with those of the ordinary failure-based calculus:

$$\frac{}{stop \textbf{ sat } tr = \langle\rangle \wedge Ref \in \mathsf{p}Act^+}$$

[2]These are summarized in Remark 3.80, for predicates that are the semantic counterpart of **sat** formulae (recall that, thanks to Proposition 3.75, these predicates are specification-oriented).

$$\frac{p \textbf{ sat } S,\ q \textbf{ sat } T}{p \uplus q \textbf{ sat } S \vee T}$$

$$\frac{p \textbf{ sat } S,\ q \textbf{ sat } T}{p \oplus q \textbf{ sat } (tr = \langle\rangle \Rightarrow S \wedge T) \wedge (tr \neq \langle\rangle \Rightarrow S \vee T)}$$

The next rule for multiple action sequence, instead, reflects the fact that failures are extended. In it, the convention is introduced that $S(t, X)$ stands for S with tr and Ref replaced by t and X. This device will be widely employed in the following.

$$\frac{\begin{array}{c} p_a \textbf{ sat } S_a, \text{ for } a \in A \\ (X \in \mathsf{p}Act^+ \wedge X \cap A = \varnothing) \Rightarrow (\forall a \in A\colon S_a(\langle\rangle, X \div a)) \Rightarrow S(\langle\rangle, X), \\ X \in \mathsf{p}Act^+ \Rightarrow S_a(s, X) \Rightarrow S(as, X), \text{ for } a \in A \end{array}}{A; p_A \textbf{ sat } S}$$

The rule for renaming is made simpler by its side-condition:

$$\frac{p \textbf{ sat } T,\ X \in \mathsf{p}Act^+ \Rightarrow T(s, X) \Rightarrow S(f(s), f(X))}{p[f] \textbf{ sat } S} \quad (f \text{ injective})$$

The rule for parallel composition is new, but has the usual conjunctive form:

$$\frac{\begin{array}{l} p_n \textbf{ sat } S_n \text{ for } 1 \leqslant n \leqslant N,\ B = \bigcup_{n=1}^{N} B_n, \\ s \in B^* \wedge X_1, \ldots, X_N, Y \in \mathsf{p}Act^+ \wedge X \subseteq \overline{\|}_{n=1}^{N} X_n{:}B_n \wedge Y \cap B^* = \varnothing \\ \Rightarrow (\bigwedge_{n=1}^{N} S_n(s \restriction B_n, X_n)) \Rightarrow S(s, X \cup Y) \end{array}}{\|_{n=1}^{N} p_n{:}B_n \textbf{ sat } S}$$

The last operator rule to be presented is for hiding:

$$\frac{\begin{array}{c} p \textbf{ sat } T, \\ X \in \mathsf{p}Act^+ \Rightarrow (\forall Y \in \mathsf{p}(X/B) : T(s, Y)) \Rightarrow S(s \backslash B, X) \end{array}}{p \backslash B \textbf{ sat } S}$$

Finally, we present the fixpoint rule. It assumes that, for every $\lambda \in \Lambda$, the natural number $h(\lambda)$ is bounded and p_λ contains no process constant but those in the tuple κ_Λ of distinct constants. Also, note the rule employs a generalized substitution:

$$\begin{aligned} {p_\Lambda}^0[stop_\Lambda] &= stop_\Lambda \\ {p_\Lambda}^{n+1}[stop_\Lambda] &= \langle p_\lambda[{p_\Lambda}^n[stop_\Lambda]/\kappa_\Lambda] \mid \lambda \in \Lambda \rangle \end{aligned}$$

The fixpoint rule is:

$$\frac{\begin{array}{l}(p_\Lambda{}^{h(\lambda)}[stop_\Lambda])_\lambda \textbf{ sat } tr = \langle\rangle \Rightarrow S_\lambda, \text{ for all } \lambda \in \Lambda, \\ \quad p_\lambda \textbf{ sat } S_\lambda \text{ for all } \lambda \in \Lambda \text{ can be proved} \\ \quad \text{by assuming } \kappa_{\lambda'} \textbf{ sat } S_{\lambda'} \text{ for all } \lambda' \in \Lambda\end{array}}{\kappa_\lambda \textbf{ sat } S_\lambda, \text{ for all } \lambda \in \Lambda} \quad \begin{array}{l}(\kappa_\lambda := p_\lambda, h(\lambda) \leqslant H, \\ \lambda \in \Lambda)\end{array}$$

4.2.3 Specification Classes and Two Derived Rules

Under suitable assumptions about specification predicates, the rules for parallel composition and hiding simplify considerably. We begin by identifying some interesting classes of specification predicates.

No-Junk and Subset-Closure

First, it should be obvious that a plausible specification predicate T ought to be satisfiable, i.e. there should exist p such that p **sat** T. A stronger, but still reasonable requirement on T is the following 'no-junk' condition:

if $T(s, X)$, then $s \in \tau\mathcal{O}[\![p]\!]$ and $X \in \rho\mathcal{O}[\![p]\!](s)$ for some p such that p **sat** T

If this condition is violated, T will be unnecessarily weak, in that it can be strengthened without restricting the class of specificands satisfying it. Thus, although we will not explicitly pose a no-junk requirement on specifications, interesting ones probably satisfy it anyway. Consider, e.g.:

$$T(s, X) \equiv s = \langle\rangle \vee X = \{a\}$$

T contains the junk $s = b, X = \{a\}$; indeed, a p such that $b \in \tau\mathcal{O}[\![p]\!]$ and p **sat** T should have $\rho\mathcal{O}[\![p]\!](b) = \{\{a\}\}$, violating subset-closure. On the other hand $T'(s, X) \equiv s{=}\langle\rangle$ does exactly the same job as T, there being no p that satisfies T but not T'.

It is worth noting that if T contains no junk, then it will coincide with a process, in that:

$$\begin{array}{rcll}\{(s, X) \mid T(s, X)\} & = & \{(s, X) \mid s \in \tau Q, X \in \rho Q(s)\} & \text{where:} \\ Q & = & \uplus\{\mathcal{O}[\![p]\!] \mid p \textbf{ sat } T\} & \end{array}$$

Next, we identify the class of subset-closed specification predicates:

Definition 4.1 A specification predicate T is subset-closed if, for all $s \in Act^*$, $X \in \mathbf{p}Act^+$ and $Y \subseteq X$, $T(s, X)$ implies $T(s, Y)$. □

Subset-closure is also a reasonable property because it is implied by the no-junk condition (as a result of the subset-closure of processes).[3] The above example of T containing junk also violates subset-closure ($T(b, \{a\})$ holds, but $T(b, \varnothing)$ does not).

Essentiality

Another useful class is introduced in:

Definition 4.2 T is *B-essential* ($\varnothing \subset B \subseteq Act$) if, for all $s \in Act^*$ and $X \in \mathbf{p}Act^+$, $T(s\restriction B, X \cap B^*)$ implies $T(s\restriction B, X)$. □

A useful specification T for p such that $\tau\mathcal{O}[\![p]\!] \subseteq B^*$ should be B-essential for every $Act \supseteq B$. This guarantees that T is insensitive to the choice of Act, or, in other words, that it needs no adaptation if Act is changed. To see why, let first $Act = B$, p **sat** T, $\mathcal{O}[\![p]\!] = P$ and suppose that, if Act grows, $\mathcal{O}[\![p]\!]$ becomes P'. If $s \in \tau P'$ and $X \in \rho P'(s)$, we could prove (by induction on the structure of p) that $X \cap B^* \in \rho P(s)$ and $s \in \tau P$, which was assumed to be a subset of B^*. It follows that $s = s\restriction B$, $T(s, X \cap B^*)$ and, since T is B-essential for the new Act, $T(s, X)$, which shows that p **sat** T is still true under the new Act.

E.g. For $B = \{a, b\}$ and any $Act \supseteq B$, the assertion $tr = \langle\rangle \Rightarrow a \notin Ref$ is B-essential, and indeed

$$a; b; stop \textbf{ sat } tr = \langle\rangle \Rightarrow a \notin Ref$$

holds independent of Act.

On the contrary, the assertion $tr = \langle\rangle \wedge Ref \subseteq \{a, b\}$ is not B-essential for, e.g., $Act = \{a, b, c\}$. Accordingly, the specification

$$stop \textbf{ sat } tr = \langle\rangle \wedge Ref \subseteq \{a, b\}$$

holds for $Act \subseteq \{a, b\}$, but does not for $Act = \{a, b, c\}$.

[3] In fact, and for the same reason, a specification predicate containing no junk also enjoys the other process properties.

Compactness

This is the last class of specification predicates to be introduced:

Definition 4.3 T is *compact* if, for all $s \in Act^*$ and $X \subseteq Act^+$, $T(s,Y)$ for all $Y \in \mathsf{p}X$ implies $T(s,X)$. □

Constructing Specification Predicates

Specification predicates belonging to the classes introduced above may be constructed or recognized exploiting the following result:

Proposition 4.4

1. A specification $T(s,Y)$ that does not depend on Y is subset-closed, B-essential and compact.
2. The following, typical deadlock-freedom specification is subset-closed, B-essential and compact:

$$T(s,Y) \equiv T_\tau(s) \Rightarrow w(s) \notin Y \qquad (w(s) \in B^* \text{ for } T_\tau(s))$$

3. Conjunction preserves each of subset-closure, essentiality and compactness.
4. Disjunction preserves each of subset-closure and essentiality; finite disjunction preserves compactness supplemented with subset-closure.

Proof. For the finite disjunction case, let T_1, T_2 be compact and subset-closed. Let $s \in Act^*$, $X \subseteq Act^+$, and suppose that, for all $Y \in \mathsf{p}X$, $T_1(s,Y) \vee T_2(s,Y)$ holds. Then, for $Y \in \mathsf{p}X$ define:

$$f(Y) = \begin{cases} 1 & \text{if } T_1(s,Y) \text{ is true,} \\ 2 & \text{otherwise.} \end{cases}$$

We may now exploit Lemma 2.11 to choose $i \in \{1,2\}$ such that, for all $Y \in \mathsf{p}X$, there is $Y' \in \mathsf{p}X, Y' \supseteq Y$ such that $f(Y') = i$, i.e. $T_i(s,Y')$ holds; thus, by subset-closure, also $T_i(s,Y)$ is true. So we have found $i \in \{1,2\}$ such that, for all $Y \in \mathsf{p}X$, $T_i(s,Y)$ holds; hence, by compactness of T_i, $T_i(s,X)$ and $T_1(s,X) \vee T_2(s,X)$ follow. This shows the disjunction of T_1 and T_2 is compact.

Proof of the remaining statements is straightforward. □

Related to Proposition 4.4 is:

Proposition 4.5 The following deadlock-freedom specifications are subset-closed, B-essential and compact:

(a) $T_\tau(tr) \Rightarrow W(tr) \cap Ref = \varnothing$, provided $W(tr) \subseteq B^*$ for $T_\tau(tr)$;

(b) $T_\tau(tr) \Rightarrow W(tr) \not\subseteq Ref$, provided $W(tr) \in \mathsf{p}B^*$ for $T_\tau(tr)$;

(c) $T_\tau(tr) \Rightarrow \exists w \in W(tr) : \forall u \leqslant w : u \notin Ref$, provided $W(tr) \subseteq B^*$ and (taking the minimum with respect to the prefix order relation) $\min W(tr)$ is finite for $T_\tau(tr)$.

Proof. All the required properties are trivial to show (from Proposition 4.4) for (a) and (b); for (c) only compactness is not. Given a suitable $W(\,)$, assume that for $s \in Act^*$ and $X \subseteq Act^+$:

(1) for all $Y \in \mathsf{p}X$: $T_\tau(s) \Rightarrow \exists w \in W(s) : \forall u \leqslant w : u \notin Y$

If $\langle\rangle \in W(s)$, clearly $\langle\rangle \notin X$, so the compactness requirement is satisfied. If $\langle\rangle \notin W(s)$, assume, contradicting compactness:

$$T_\tau(s) \wedge \forall w \in W(s) : \exists u_w \leqslant w : u_w \in X.$$

Let $Y = \{u_{w'} \mid w' \in \min W(s)\}$; then $Y \in \mathsf{p}X$ and provides a contradiction with (1). Indeed, for any $w \in W(s)$, choose $w' \in \min W(s)$ such that $w' \leqslant w$; then $u_{w'} \leqslant w' \leqslant w$ and $u_{w'} \in Y$. □

Some observations are worth making on the assertion patterns (a), (b) and (c) considered in the statement of this lemma:

Remark 4.6

1. Patterns (a) and (b) could be rewritten in the form of a conjunction and a finite disjunction respectively.

2. The finiteness conditions that patterns (b) and (c) place on the set $W(tr)$ serve to ensure compactness for assertions conforming to them. Compactness

may instead not hold for an infinite $\min W(tr)$, even in the restricted case that $W(tr)$ only contains traces over a finite set B. Consider e.g.:

$$W = \{b^n a \mid n \geqslant 0\} \qquad T \equiv \exists w \in W : w \notin Ref$$

Now $T(Y)$ holds for any $Y \in \mathsf{p}W$ (just choose $w = b^n\langle a\rangle$ for $n > \max\{\#y \mid y \in Y\}$); yet $T(W)$ is false, so T is not compact.

Both the need for compactness and the fact that it fails to hold in some cases stem from the same semantic issue: the finiteness constraint on refusals. Other related problems, examined in Section 3.3.9, are the imperfect associativity and operational justification of hiding.

The relationship among these issues may be illustrated by means of an interesting example. Let the process expression p stand for κ_1, where

$$\kappa_n := b^n; a; stop \uplus \kappa_{n+1}$$

It is not difficult to show that p **sat** T and, despite operational intuition, $p\backslash b$ refuses $\{a\}$.[4] Accordingly, it should not be possible to infer $p\backslash b$ **sat** S, with S being $tr = \langle\rangle \Rightarrow a \notin Ref$. Indeed, such an inference is allowed neither by the hiding rule introduced earlier (because $\forall Y \in \mathsf{p}(X/B) : T(t, Y)$ does not imply $S(t\backslash B, X)$), nor by the simpler rule (4-2) below (inapplicable, even though $T(t, X/B)$ implies $S(t\backslash B, X)$, because T is not compact).

3. Last, it may be worth mentioning another problem of assertions like:

$$T_\tau(tr) \Rightarrow \exists w \in W(tr) : w \notin Ref$$

in connection with the cardinality of the set $W(tr)$. If, for all s making $T_\tau(s)$ true, the set $\{u[1] \mid u \in W(s) - \{\langle\rangle\}\}$ turns out to be infinite, then every p will vacuously satisfy the above assertion. Indeed, let p refuse $X \in \mathsf{p}Act^+$ after s such that $T_\tau(s)$; to find $w \in W(s)$ such that $w \notin X$, it suffices to choose in $W(s)$ a w such that no trace in X begins with $w[1]$.

[4] As examined in Section 3.3.9, this fact is related to the transition relation of p not being image-finite ($p \stackrel{\langle\rangle}{\Longrightarrow}$ has an infinite image).

Derived Rules

Weaker but simpler derived rules may now be presented for parallel composition:

$$\frac{\begin{array}{l} p_n \textbf{ sat } S_n \text{ for } 1 \leqslant n \leqslant N,\ B{=}\bigcup_{n=1}^{N} B_n, \\ (\bigwedge_{n=1}^{N}(X_n \in \mathsf{p}Act^+ \wedge S_n(s{\restriction}B_n, X_n))) \Rightarrow \\ S(s{\restriction}B, \overline{\|}_{n=1}^{N} X_n{:}B_n) \end{array}}{\|_{n=1}^{N} p_n{:}B_n \textbf{ sat } S} \quad \begin{array}{l}(S \text{ subset-closed,} \\ B\text{-essential})\end{array} \tag{4-1}$$

and hiding:

$$\frac{p \textbf{ sat } T,\ X \in \mathsf{p}Act^+ \Rightarrow T(s, X/B) \Rightarrow S(s\backslash B, X)}{p\backslash B \textbf{ sat } S} \quad (T \text{ compact}) \tag{4-2}$$

For simplicity, we stipulate:

Assumption 4.7 Henceforth all assertions introduced, unless otherwise stated, admit the application of rules (4-1) and (4-2). □

This assumption is convenient because most of the assertions employed in the following fall into one of the cases dealt with by Proposition 4.5.

4.2.4 Process-Oriented and Consistency Rules

Process-oriented rules may be viewed as a new kind of healthiness rules, in that they reflect and stem from specific process properties.

First, we introduce a rule that stems from the finiteness of refusal functions images:

$$\frac{}{p \textbf{ sat } \mathit{Ref} \in \mathsf{p}Act^+} \tag{4-3}$$

The original **sat** calculus of [Hoare, 1985] did not contain any counterpart of this rule. In fact, it could have been spared here too if the semantic interpretation of **sat** formulae had been defined in a suitable way, limiting the universe of variable *Ref* to $\mathsf{p}Act^+$. Since (in the style of [Hoare, 1985]) we wanted to skirt such intricacies, we preferred to include explicitly in the calculus this assumption (and similar ones, cf. the premise $X \in \mathsf{p}Act^+$ in various rules from Section 4.2.2 onwards).

The following three process-oriented rules result from trace-refusal consistency of processes (see proof of Proposition 4.8). The simplest is:

$$\frac{\begin{array}{l} p \textbf{ sat } S,\ \ X \in \mathsf{p}Act^+ \Rightarrow S(t,X) \Rightarrow T(t,X), \\ \quad X \in \mathsf{p}Act^+ \Rightarrow H(t,X) \Rightarrow S(t,X) \Rightarrow \\ \quad \neg T(t, X \cup \{y(t)\}) \end{array}}{p \textbf{ sat } H \Rightarrow S(tr \cdot y(tr), Ref \div y(tr))} \quad \begin{array}{l}(y(t) \in Act^+ \\ \text{for } t \in Act^*)\end{array} \tag{4-4}$$

Now, two rules combining consistency with $\forall$ introduction:

$$\frac{\begin{array}{l} p \textbf{ sat } S, \\ X \in \mathsf{p}Act^+ \Rightarrow S(t,X) \Rightarrow \forall y \in Y(t) : T(t,X), \\ \quad X \in \mathsf{p}Act^+ \Rightarrow H(t,X) \Rightarrow S(t,X) \Rightarrow \\ \quad \forall y \in Y(t) : \neg T(t, X \cup \{y\}) \end{array}}{p \textbf{ sat } H \Rightarrow \forall y \in Y(tr) : S(tr \cdot y, Ref \div y)} \quad \begin{array}{l}(y \text{ not in } S, \\ Y(t) \subseteq Act^+ \\ \text{for } t \in Act^*)\end{array} \tag{4-5}$$

and $\exists$ introduction:[5]

$$\frac{\begin{array}{l} p \textbf{ sat } S, \\ X \in \mathsf{p}Act^+ \Rightarrow S(t,X) \Rightarrow T(t,X), \\ \quad X \in \mathsf{p}Act^+ \Rightarrow H(t,X) \Rightarrow \\ \quad S(t,X) \Rightarrow \neg T(t, X \cup Y(t)) \end{array}}{p \textbf{ sat } H \Rightarrow \exists y \in Y(tr) : S(tr \cdot y, Ref \div y)} \quad \begin{array}{l}(Y(t) \subseteq Act^+ \text{ for } t \in Act^*, \\ T \text{ compact}, y \text{ not in } S)\end{array} \tag{4-6}$$

Note that all these 'consistency' **sat** rules admit of a simpler form in which H is dropped (by letting it be identically true).

Proposition 4.8 Rules (4-4), (4-5), (4-6) are sound.

Proof. For the first rule, assume its premises and side condition; let P be $\mathcal{O}[\![p]\!]$, and assume $t \in \tau P$, $X \in \rho P(t)$ and $H(t,X)$. This implies $S(t,X)$ and $\neg T(t, X \cup \{y(t)\})$. Moreover, we claim that $X \cup \{y(t)\} \notin \rho P(t)$, or $S(t, X \cup \{y(t)\})$ and the contradictory $T(t, X \cup \{y(t)\})$ would follow. Trace-refusal consistency therefore ensures $t \cdot y(t) \in \tau P$ and $X \div y(t) \in \rho P(t \cdot y(t))$, whence the desired $S(t \cdot y(t), X \div y(t))$ follows.

[5]The side condition that T is compact may be exchanged with the finiteness of $Y(\,)$.

Proof of the $\forall$ introduction rule is an easy extension; we provide only that of the $\exists$ introduction one. Again, assume its premises and side conditions, and $P = \mathcal{O}[\![p]\!]$, $t \in \tau P$, $X \in \rho P(t)$, $H(t, X)$. Then $S(t, X)$, $T(t, X)$ and $\neg T(t, X \cup Y(t))$ hold. We claim that we can:

(1) choose $y \in Y(t)$ and $Y' \subseteq Y(t)$ such that: for all $Z \in \mathsf{p}Y'$, $Z \cup X \in \rho P(t)$, but for some $Z' \in \mathsf{p}Y'$, $Z' \cup \{y\} \cup X \notin \rho P(t)$.

Then, by trace-refusal consistency and refusal subset-closure: $t \cdot y \in \tau P$ and $X \div y \in \rho P(t \cdot y)$, whence the desired $S(t \cdot y, X \div y)$ follows.

To show (1) is possible, infer from its contradiction:

for all $y \in Y(t)$, $Y' \in \mathsf{p}Y(t)$: if $Y' \cup X \in \rho P(t)$, then $Y' \cup \{y\} \cup X \in \rho P(t)$.

Hence, by induction on the size of Y':

for all $Y' \in \mathsf{p}(Y(t))$, $Y' \cup X \in \rho P(t)$.

Using, p **sat** S and $S \Rightarrow T$, by compactness of T, it follows that $T(t, X \cup Y(t))$—a contradiction. □

Motivating Consistency Rules

Consistency rules essentially represent a means whereby, given some process refusals juxtaposed into a longer one, the properties of the former may be combined to derive a property of the latter. Such rules are well motivated by natural questions like the following: "if p never refuses $\langle a \rangle$, how can we conclude it never refuses $\langle aa \rangle$, *even if we don't know anything else about* p?". An answer is provided by rule (4-4), letting H be true, S and T be $\langle a \rangle \notin Ref$, and y be $\langle a \rangle$; the rule implies p **sat** $\langle a \rangle \notin (Ref \div \langle a \rangle)$, i.e. p **sat** $\langle aa \rangle \notin Ref$.

An interesting issue is the relationship between consistency rules and those presented earlier. To begin with, at a semantic level consistency rules only hold for processes, whereas the previous operator-related rules also apply to the more general failure sets over which process operators are in fact defined. At present, it is not clear whether consistency rules can be derived from the rest of the **sat** calculus (through e.g. some form of structural induction on process expressions),

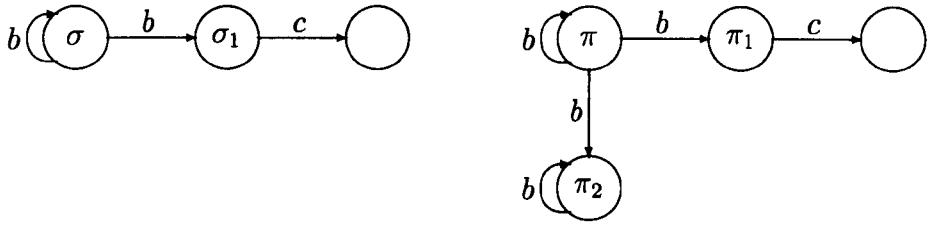


Figure 4.1: σ and π are failure- but not extended failure- equivalent.

or whether they are instead independent or even required for completeness. In any case, they enable us to reason about process expressions known only through the assertions they satisfy. Consider, e.g., the previous example; if p were known to be κ and $\kappa := a;\kappa$, it would be easy to derive p **sat** $\langle aa \rangle \notin Ref$ without consistency rules, but this becomes impossible if p is unknown except for the specification p **sat** $\langle a \rangle \notin Ref$. Section 4.5.4 proves several **sat** formulas in which the process expressions (a sender, a receiver and two media) are only known up to a set of basic assertions they are required to satisfy. As discussed in Section 4.5.2, the ability to reason about specified but undefined process expressions is an important feature of the **sat** approach overall; this ability is enhanced, in the presence of extended refusals, by the introduction of consistency rules.

Another interesting topic for further investigation is the extent to which consistency rules suffice to derive properties of extended refusals from those of simple, one-action refusals. That this extent cannot be complete is readily shown (switching to LTSs to aid intuition) by a counterexample already exploited earlier, shown in Figure 4.1. It is not difficult to realize that every failure (tr, Ref) of σ must satisfy:

$$tr \in b^* \Rightarrow c \notin Ref \vee bc \notin Ref \tag{4-7}$$

whereas the failures of π do not; on the other hand the simple failures of σ and π coincide and hence satisfy exactly the same properties; therefore these cannot suffice to infer that the extended failures of σ satisfy (4-7), or those of π would too. However, no such problem occurs for the non-trivial (but less pathological) examples of Section 4.5.4: in all of them, properties of extended refusals are inferred from those of one-action refusals.

4.3 A Short Example

The **sat**-calculus will now be applied to the verification of a system that provides reliable communication over an unreliable medium. This goal is achieved by a naive, brute-force protocol: all messages are transmitted over and over again across the medium, which can fail in arbitrary ways, but does not corrupt messages; sequence numbers allow the correct message sequence to be reconstructed at the receiving end. Such a protocol is clearly unacceptable in practice, for efficiency reasons; however, what matters here is that its verification employs techniques that are far from trivial and play a key role in important applications (like, e.g., the sliding-window protocol that will be verified later). In fact, even this simplified example already illustrates the main advantage that our **sat**-calculus has over the original one of [Hoare, 1985]: the ability to deal with potentially diverging systems.

4.3.1 Specification

Architectural Specification

The system studied is expected to behave like a reliable buffer conveying *service data units* (sdus) between a sending and a receiving user. In order to ensure reliability, a sender and a receiver module must be added on top of the unreliable medium available. The sender and the receiver are linked by the medium, through which they exchange *protocol data units* (pdus).

Data representation will just be sketched. A pdu p encodes a pair (x, n), where x is a sdu and n a sequence number; formally, this is modelled with a constructor function *pdu*, and accessors *sdu* and *seqno* satisfying:

$$\begin{aligned} seqno(pdu(x,n)) &= n \\ sdu(pdu(x,n)) &= x \\ pdu(sdu(p), seqno(p)) &= p \end{aligned}$$

The system structure is shown in Figure 4.2. Accordingly, the behaviour of the system will be described by a process expression *SP* (for *Service Provider*), whose interface comprises the channels *snd* and *rcv*. Sender, receiver and medium

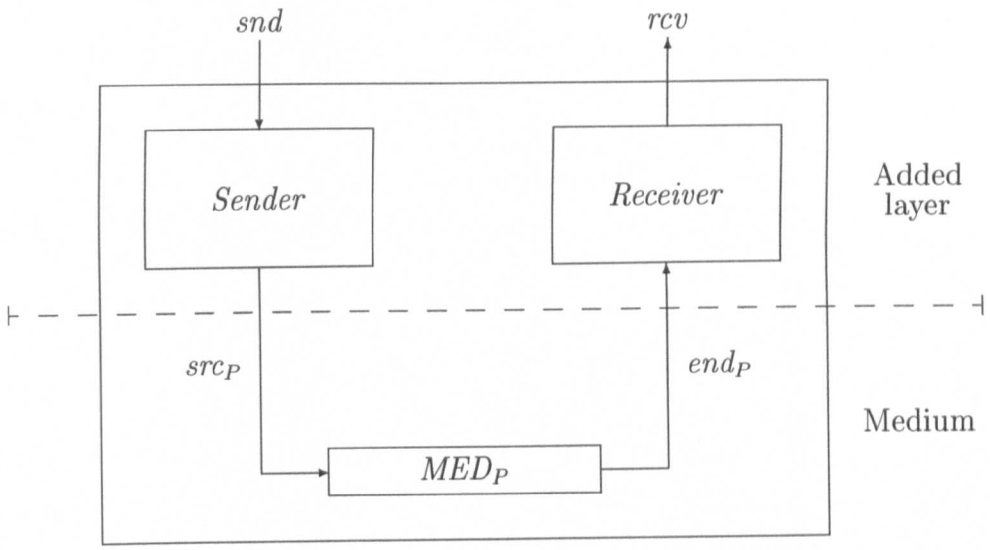


Figure 4.2: The example system SP.

are described by the process expressions *Sender*, *Receiver* and MED_P respectively. *Sender* accepts sdus at *snd* and passes pdus to MED_P at src_P. *Receiver* receives pdus from MED_P at end_P and outputs sdus at *rcv*. Thus, the interfaces of *Sender*, MED_P and *Receiver* are respectively the channel sets:

$$C_S = \{snd, src_P\} \qquad C_P = \{src_P, end_P\} \qquad C_R = \{end_P, rcv\}$$

In the following, it is also convenient to let, for any channel set C:

$$\mathsf{A}_C = \{c!v \mid c \in C\} \qquad B_\alpha = \mathsf{A}_{C_\alpha}, \text{ for } \alpha = S, P, R$$

SP can now be defined:

$$SP \equiv (Sender : C_S \parallel MED_P : C_P \parallel Receiver : C_R) \setminus C_P$$

Specifying the Behaviour of the System and its Components

The desired behaviour of SP can be formally specified in the **sat** logic by:

$$SP \textbf{ sat } PREFIX \wedge INLIV \wedge OUTLIV$$

PREFIX requires that the output sdu sequence should be a prefix of the input one:

$$PREFIX \equiv tr{\downarrow}rcv \leqslant tr{\downarrow}snd$$

INLIV specifies that, if there are no undelivered sdus, the system should accept any input:

$$INLIV \equiv tr{\downarrow}rcv = tr{\downarrow}snd \Rightarrow \forall x\colon snd!x \notin Ref$$

Finally, *OUTLIV* prescribes that, if a sdu x is the next to be delivered according to *PREFIX*, it should not be refused at *rcv*:

$$OUTLIV \equiv (tr{\downarrow}rcv)x \leqslant tr{\downarrow}snd \Rightarrow rcv!x \notin Ref$$

The behaviour of the components will now be described both informally and, for some selected aspects, formally. One of the rewards of the **sat** approach is the ability to carry out partial specification and verification. E.g., in the present example, verifying that *SP* **sat** *OUTLIV* requires only:

Assumption 4.9

$$Sender \textbf{ sat } T_S \qquad Receiver \textbf{ sat } T_R \qquad MED_P \textbf{ sat } T_P$$

for T_S, T_R and T_P allowing Lemma 4.10 to be proved and satisfying the requirements identified below, i.e. implying (4-8), (4-9), (4-10) respectively. □

According to this assumption, it is not necessary for T_S, T_R and T_P to identify *Sender*, *Receiver* and MED_P precisely (i.e. to be their strongest specifications) or even for these process expressions to be actually defined; this definition can be deferred to a later refinement step or even omitted. It should in fact be viewed as an advantage that the results obtained will hold for any *Sender*, *Receiver* and MED_P satisfying T_S, T_R and T_P respectively. These ideas are further developed in Section 4.5.2.

For brevity, we shall refrain from defining T_S, T_R and T_P fully and concentrate instead on their parts that are crucial for deadlock freedom verification.

Sender accepts sdus at *snd* and repeatedly transmits pdus at src_P; in particular, it is always ready to transmit $pdu(x, n)$, where x is the nth sdu received. Thus T_S should imply:

$$\forall n\colon 0 < n \leqslant \#(tr{\downarrow}snd) \Rightarrow src_P!pdu((tr{\downarrow}snd)[n], n) \notin Ref \qquad (4\text{-}8)$$

Receiver receives pdus at end_P, ignoring those out-of-sequence and extracting from those expected a sdu that is then output at *rcv*. More specifically, suppose *Receiver* has already output r sdus; then, if it has just received $pdu(x, r+1)$, it will not refuse to output x at *rcv*; otherwise, it will not refuse any group of out-of-sequence pdus, followed by the expected $pdu(x, r+1)$ at end_P and by the output of x at *rcv*. Formally, all this can be expressed by requiring T_R to imply:

$$\begin{array}{l}\textbf{let } r := \#(tr{\downarrow}rcv),\ l := \#tr \textbf{ in} \\ \text{if } l > 0 \wedge tr[l] = end_P!pdu(z, r+1) \\ \text{then } rcv!z \notin Ref \\ \text{else } \forall u \in \{end_P!p \mid seqno(p) \neq r+1\}^* : \forall d : \\ \qquad u\,\langle end_P!pdu(d, r+1), rcv!d\rangle \notin Ref\end{array} \tag{4-9}$$

The medium MED_P will be required to never reach a state from which a pdu q cannot be output after a suitable preamble; moreover, of course, no prefix of this behaviour should be refused. So T_P must imply:

$$\forall q\colon \exists w \in Preamble_P(tr, q) : \forall u \leqslant w\langle end_P!q\rangle\colon u \notin Ref \tag{4-10}$$

The trace set $Preamble_P(tr, q)$ will be assumed finite. This assumption will be justified in Section 4.4, which contains a detailed discussion on media specification.

4.3.2 Deadlock Freedom Verification

For brevity, we only provide a verification of the *OUTLIV* property (the next sdu to be delivered at *rcv* is not refused). *OUTLIV* is more complicated than *INLIV*, and more interesting than *PREFIX* because it cannot be treated entirely within the known methods of [Hoare, 1985]. However, also these methods are needed (with the full knowledge of T_S, T_R and T_P postulated by Assumption 4.9), in order to show:

Lemma 4.10 Pdus input at src_P or output at end_P faithfully reflect sdus input at *snd*:

$$(Sender : C_S \parallel MED_P : C_P \parallel Receiver : C_R) \textbf{ sat } FIDEL$$

$$FIDEL \equiv \forall z, n\colon pdu(z, n) \in (tr{\downarrow}src_P) \cup (tr{\downarrow}end_P) \Rightarrow (tr{\downarrow}snd)[n] = z \qquad \square$$

The desired result is:

Proposition 4.11 *SP* **sat** *OUTLIV*.

Proof. Lemma 4.12 (given below) implies:

$$(Sender : C_S \parallel MED_P : C_P \parallel Receiver : C_R) \textbf{ sat}$$
$$(tr{\downarrow}rcv)x \leqslant tr{\downarrow}snd \Rightarrow$$
$$\exists z : (z = x \vee pdu(z, 1{+}\#(tr{\downarrow}rcv)) \in tr{\downarrow}src_P \cup tr{\downarrow}end_P) \wedge$$
$$\exists u \in W_x(tr) : u\langle rcv!z\rangle \notin Ref$$

where, for all relevant s, $W_x(s)$ is a finite non-empty subset of $B_P{}^*$ (the set of actions at channels in C_P). This assertion and *FIDEL* imply:

$$OL \equiv (tr{\downarrow}rcv)x \leqslant tr{\downarrow}snd \Rightarrow \exists u \in W_x(tr) : u\langle rcv!x\rangle \notin Ref$$

So the previous lemma, with the conjunction and consequence rules of [Hoare, 1985], imply:

$$(Sender : C_S \parallel MED_P : C_P \parallel Receiver : C_R) \textbf{ sat } OL$$

By Proposition 4.5 OL is compact because $W_x(s)$ is finite, so the desired result would follow by the hiding rule (4-2) from:

$$OL(s, X/B_P) \Rightarrow OUTLIV(s \backslash B_P, X)$$

To see this, assume $(s \restriction rcv)x \leqslant s \restriction snd$; if, by contradiction, $rcv!x \in X$, then $u\langle rcv!x\rangle \in X/B_P$ for all $u \in B_P{}^*$, against $OL(s, X/B_P)$ (note the hypothesis $W_x(s) \neq \varnothing$ is crucial for the contradiction). □

The proof obligation left is the hardest:

Lemma 4.12

$$(Sender : C_S \parallel MED_P : C_P \parallel Receiver : C_R) \textbf{ sat } T$$
$$T(s, Y) \equiv (s{\downarrow}rcv)x \leqslant s{\downarrow}snd \Rightarrow \textbf{let } r := \#(s{\downarrow}rcv) \textbf{ in}$$
$$\exists z : (z = x \vee pdu(z, r{+}1) \in s{\downarrow}src_P \cup s{\downarrow}end_P) \wedge$$
$$(rcv!z \notin Y \vee$$
$$\exists w \in Preamble_P(s \restriction C_P, pdu(x, r{+}1)) : \exists w' :$$
$$w' \leqslant w \wedge w'\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Y)$$

Proof. We observe first that $T(s, Y)$ and $T(s\restriction(C_S \cup C_P \cup C_R), Y)$ are equivalent. By the **sat** rule (4-1) for parallel composition, under Assumption 4.9, the proof amounts to deriving:

(1) $T(s, X_S : C_S \,\overline{\|}\, X_P : C_P \,\overline{\|}\, X_R : C_R)$

from the assumptions:

(2) $X_S, X_P, X_R \in \mathsf{p}Act^+$

and $T_S(s\restriction C_S, X_S)$, $T_P(s\restriction C_P, X_P)$, $T_R(s\restriction C_R, X_R)$. By Assumption 4.9, these imply:

(2S) $\forall n : 0 < n \leqslant \#(s{\downarrow}snd) \Rightarrow src_P!pdu((s{\downarrow}snd)[n], n) \notin X_S$

(2P) $\forall q : \exists w \in Preamble_P(s\restriction C_P, q) : \forall u \leqslant w\langle end_P!q\rangle : u \notin X_P$

(2R) **let** $r := \#(s{\downarrow}rcv)$, $l := \#(s\restriction C_R)$ **in**
if $l > 0 \wedge (s\restriction C_R)[l] = end_P!pdu(z, r{+}1)$
then $rcv!z \notin X_R$
else $\forall u \in \{end_P!q \mid seqno(q) \neq r{+}1\}^* : \forall d :$
$u\,\langle end_P!pdu(d, r{+}1), rcv!d\rangle \notin X_R$

The intended meaning of the set $Preamble_P(\,)$ should give plausibility to the following facts, stipulated later as Assumption 4.16:

(3) For all $s \in {B_P}^*$ and q, $w \in Preamble_P(s, q)$ implies $w \in {B_P}^*$, $w{\downarrow}src_P = q$ and $w{\downarrow}end_P \subseteq (s{\downarrow}src_P) \cup \{q\}$.

Let (1.1) and (1.2) denote respectively the first and second conjunct in the scope of the first $\exists$ of (1). To establish (1), we shall infer (1.1) and (1.2) from the premise:

(4) $(s{\downarrow}rcv)x \leqslant s{\downarrow}snd$, whence, letting $r = \#(s{\downarrow}rcv)$:
$r{+}1 \leqslant \#(s{\downarrow}snd)$ and $(s{\downarrow}snd)[r{+}1] = x$.

We now consider two cases. The first consists in assuming, for some z:

(5) $l = \#(s\restriction C_R) > 0$ and $(s\restriction C_R)[l] = end_P!pdu(z, r{+}1)$

This implies (1.1) directly, and (1.2) because from (2) and (2R) it follows:

$$rcv!z \notin X_S : C_S \,[\![\, X_P : C_P \,]\!]\, X_R : C_R$$

The other case is that (5) does not hold. Using (2P), we choose w such that:

(6) $w \in Preamble_P(s\restriction C_P, pdu(x, r{+}1))$, and $\forall u \leqslant w\langle end_P!pdu(x, r{+}1)\rangle : u \notin X_P$

and we may distinguish two subcases.

Letting $w' = w$ and $z = x$ (which verifies (1.1)), suppose:

(7) $w'\restriction end_P \in \{end_P!q \mid seqno(q) \neq r{+}1\}^*$.

Letting $w_1 = w'\,\langle end_P!pdu(z, r{+}1), rcv!z\rangle$, the following argument can be arranged (relations in the second column[6] follow from (6) and (3), the third '$\notin$' column follows from the fourth):

$w_1\restriction C_S$	$(\in \{\langle\rangle, \langle src_P!pdu(x, r{+}1)\rangle\})$	$\notin X_S$	(2) or (4),(2S)
$w_1\restriction C_P$	$(= w'\langle end_P!pdu(z, r{+}1)\rangle)$	$\notin X_P$	(6)
$w_1\restriction C_R$	$(= (w'\restriction end_P)\,\langle end_P!pdu(z, r{+}1), rcv!z\rangle)$	$\notin X_R$	(7), (2R)

Hence follows (1.2), since:

$$w'\,\langle end_P!pdu(z, r{+}1), rcv!z\rangle = w_1 \notin X_S : C_S \,[\![\, X_P : C_P \,]\!]\, X_R : C_R$$

The last case to consider is that (7) does not hold for $w' = w$. But then we may choose w', z such that

(8) $w'\,\langle end_P!pdu(z, r{+}1)\rangle \leqslant w$ and (7) holds

Hence (1.2) can be derived as in the previous case. Finally, $pdu(z, r{+}1) \in w{\downarrow}end_P$ (from (8)) and $w{\downarrow}end_P \subseteq (s{\downarrow}src_P) \cup \{pdu(x, r{+}1)\}$ (from (6) and (3)); this establishes (1.1). □

[6] The first row has a more liberal '$\in \{\langle\rangle, \langle src_P!pdu(x, r{+}1)\rangle\}$', in lieu of '$= \langle src_P!pdu(x, r{+}1)\rangle$' (from (3)), to suit also the last part of the proof.

4.3.3 A Deadlock Freedom Verification Strategy for Systems with Hidden Channels

From the material in the previous section, a general strategy seems to emerge for proving a deadlock freedom property of the form:

$$p\backslash C \textbf{ sat } T_\tau(tr) \Rightarrow c!v(tr) \notin Ref \quad (c \notin C) \tag{4-11}$$

Validity of (4-11) can be inferred from that of:

$$p \textbf{ sat } T_\tau(tr) \Rightarrow \exists w \in W(tr)\colon w\langle c!v(tr)\rangle \notin Ref \tag{4-12}$$

as stated in:

Proposition 4.13 The **sat** formula (4-11) follows from (4-12), provided, for any relevant argument $s \in Act^*$, $T_\tau(s\backslash C)$ implies $T_\tau(s)$ and $W(s) \in \mathsf{p}(\mathsf{A}_C)^* - \{\varnothing\}$.

Proof. An application of the hiding rule (4-2), via the same argument used for Proposition 4.11 (an instance of that at hand).

Rather than repeating the proof, we stress two interesting facts. (i) Since $T_\tau(tr)$ is part of the specification (4-11) of $p\backslash C$, it can usually be written with occurrences of tr projected onto channels of $p\backslash C$, which are outside C; this means that in $T_\tau(s\backslash C)$ hiding at C is superfluous for the occurrences of s; as a result $T_\tau(s\backslash C)$ will imply $T_\tau(s)$, as required by the hypothesis. (ii) The finiteness of $W(s)$ serves to ensure compactness[7] of the assertion in (4-12), by Proposition 4.5. □

We are now faced with the problem of identifying a suitable $W(\,)$. For this purpose, we shall make recourse to operational intuition and therefore deliberately avoid distinguishing between the process expression p and a LTS producing the failures denoted by p. We proceed by considering the traces of the form $w\langle c!z\rangle$ $(w \in (\mathsf{A}_C)^*)$ which p may perform from states reached after a generic trace s satisfying T_τ. The verifier should identify a finite set of the former traces, and convince himself that at least one of these is not refused by p from any state reached through s. This

[7]Note that giving up compactness and relying on the more complex, full hiding **sat**-rule would not help, essentially owing to refusal finiteness (a counterexample can be constructed along the lines of Remark 4.6, item (2)).

informal analysis, based on suitable deadlock freedom assumptions (reflecting (4-10)) for the media within p, should enable the deadlock freedom requirements for the other components to be checked and if necessary adjusted, or even discovered from scratch.

As an example, let p be the system

$$Sender : C_S \parallel MED_P : C_P \parallel Receiver : C_R$$

of Section 4.3. Suppose p has done s such that $(s{\downarrow}rcv)x \leqslant s{\downarrow}snd$, and let r be $\#(s{\downarrow}rcv)$, the number of sdus output with s; intuition suggests that some of the traces p can produce after s have one of the following forms:

1. $rcv!z$, where $pdu(z, r{+}1)$ is the last pdu arrived at end_P with s;
2. $w'\langle end!pdu(z, r{+}1), rcv!z\rangle$, where z is a sdu that entered MED_P with s, and $w' \in \mathsf{A}^*_{end_P}$ does not contain any pdu with sequence number $r{+}1$;
3. $w'\langle end!pdu(z, r{+}1), rcv!z\rangle$, where $w' = w'_1\langle src_P!pdu(x, r{+}1)\rangle w'_2$, $w'_1, w'_2 \in \mathsf{A}^*_{end_P}$ do not contain any pdu with sequence number $r{+}1$, and z is x or a sdu that entered MED_P with s.

For MED_P to be guaranteed to perform its share $w'\langle end!pdu(z, r{+}1)\rangle$ of cases (2) and (3), we need to constrain the range where w' and z may vary. We recall (cf. (4-10)) that there must exist a $w \in Preamble_P(s\restriction C_P, pdu(x, r{+}1))$ such that MED_P does not refuse any prefix of $w\langle end_P!pdu(x, r{+}1)\rangle$. So $w'\langle end!pdu(z, r{+}1)\rangle$ will be required to be one such prefix.

Further, it is not difficult to argue that, if *Sender* and *Receiver* fulfil the properties formalized by (4-8) and (4-9), they will not refuse to take part in at least a trace as in (1)–(3), whichever state they have reached by participating in s. In fact, these informal and often rough arguments were the starting point of an effort that, through some stepwise refinement, culminated in the formulation of (4-8) and (4-9), and the reasoning employed in subsequent proofs.

Generalizing these results, we may state that the previous step should yield a **sat** formula of the form:

$$p \textbf{ sat } T'_\tau(tr) \Rightarrow \exists z \in V(tr) : \exists w \in W_z(tr) : w\langle c!z\rangle \notin Ref \qquad (4\text{-}13)$$

where (for all relevant argument s) $V(s)$ is finite, and $W_z(s)$ should be finite too, being expressed in terms of (supposedly) finite media preamble sets. Note that the system p is thus guaranteed to perform the desired instance of $w\langle c!z\rangle$ from states reached through traces satisfying T'_τ, rather than the T_τ considered initially in (4-12); this provides for the eventuality that, in the informal analysis step, such alternative states might appear to be easier to deal with. In any case, it is to be expected that, under a suitable safety property, for any trace s of p, $T_\tau(s)$ should imply $T'_\tau(s)$ and $z \in V(s)$ should imply $z = v(s)$. This will allow (4-12) to be inferred from (4-13), which therefore remains as the last obligation of the verification proof outlined.

In the example worked out earlier, the role of obligation (4-13) is played by the **sat** formula proved in Lemma 4.12; this formula is indeed an instance of (4-13) under the definitions:

$$\begin{aligned}
c &= rcv \\
r(s) &= \#(s{\downarrow}rcv) \\
x(s) &= (s{\downarrow}snd)[1{+}r(s)] \\
T'_\tau(s) &= r(s) < \#(s{\downarrow}snd) \\
V(s) &= \{x(s)\} \cup \{y \mid pdu(y, r(s){+}1) \in (s{\downarrow}src_P) \cup (s{\downarrow}end_P)\} \\
W_z(s) &= \{\langle\rangle\} \cup \{w'\langle end_P!pdu(z, r(s){+}1)\rangle \mid \\
&\qquad \exists u \in Preamble_P(s \lceil C_P, pdu(x(s), r(s){+}1)) : w' \leqslant u\}
\end{aligned}$$

We conclude this topic with a useful observation.

Remark 4.14 The exact nature of the set $W(tr)$ in (4-12) is in fact unimportant for applying **sat** rule (4-2) to (4-12) in order to infer the desired (4-11) (cf. Proposition 4.13 and, as an aid to intuition, recall traces in $W(tr)$ are to be hidden anyway). It is only mandatory that $W(s)$ should be non-empty and finite for appropriate s.

Therefore, a derivation of $W_z(s)$ like that sketched above is only necessary to ensure that $W(s)$ (which depends on $W_z(s)$) enjoys non-emptiness and finiteness.

In turn, for this purpose, as shown by the example $W_z(s)$ above, it is not necessary to know media preamble sets exactly, but only to rely on them to possess just the same non-emptiness and finiteness properties. □

4.4 Specifying Unreliable Media

This section deals with the specification of a process *MED* intended to model an unreliable communication medium. *MED* has input channel *src* and output channel *end*, and will usually be considered through its renamed instances, like e.g. MED_P of Section 4.3.

An otherwise unreliable medium is normally at least assumed not to corrupt messages (thanks to redundancy checks); thus any pdu coming out of *end* must have been previously input at *src*. Formally:

$$MED \textbf{ sat } M.REL \tag{4-14}$$
$$M.REL \equiv tr{\downarrow}end \subseteq tr{\downarrow}src$$

4.4.1 Introducing Media Deadlock Freedom

Intuitively, one would expect that *MED* must also guarantee to perform some sort of activity every now and then, or even a deadlocked medium would be acceptable. However, the ordinary **sat** calculus does not allow such requirements to be formalized satisfactorily in a general way. Consider, e.g., a medium that has no buffering capacity and can only fail by losing messages but is otherwise ideal; little can be guaranteed about its (one-action) refusals: we might tentatively require that either the medium does not refuse input, or is ready to output the last message that has just entered it (the previous ones having been already delivered or lost):

$$MED \textbf{ sat } end!last(tr{\downarrow}src) \notin Ref \vee \forall e : src!e \notin Ref$$

But this does not ensure the medium is useful; no conclusion can be drawn as to whether any message will eventually get through; a *MED* that just throws input data away would satisfy the above specification. Of course, in this framework deadlock freedom cannot be verified for systems built on top of such a *MED*. A partial remedy could be based on placing an upper bound on the number of consecutive losses, but choosing such bounds is always arbitrary and leads to treatments that, depending on the bound chosen, are more cumbersome and complex than desirable.

These limitations can be overcome using our extended failures and calculus, which make it possible to specify that the medium is not deadlocked, in that it is

capable of conveying any message d. For this purpose, it will be required that at any stage either *MED* (being empty) does not refuse to input and output d, or that it does not refuse to output the message it holds and then acquire and output d:

$$MED \textbf{ sat } \forall d: \forall u \leqslant \langle src!d, end!d \rangle : u \notin Ref \vee \forall u \leqslant \langle end!last(tr{\downarrow}src), src!d, end!d \rangle : u \notin Ref \tag{4-15}$$

Note that it is explicitly being excluded that two or more inputs are *necessary* for the medium to output d, not that the medium may lose several consecutive inputs. Thus the medium may well e.g. perform initially a trace $\langle src!d, src!d, src!d \rangle$ and reach a state whence $end!d$ is refused; however, both from the initial and final state it should not refuse to perform $\langle src!d, end!d \rangle$.

The deadlock freedom property introduced can be effectively exploited in verification, and indeed was in Section 4.3.2. The form of the assertion (4-10) employed there can be given to that in (4-15) above by rewriting it as:

$$\forall d: \exists w \in Preamble(tr, d) : \forall u \leqslant w\langle end!d \rangle : u \notin Ref$$

having defined the finite set

$$Preamble(tr, d) = \{\langle src!d \rangle, \langle end!last(tr{\downarrow}src), src!d \rangle\}$$

It may also be worth observing here that, for the application discussed, the **sat** methodology turns out to be superior to the purely equational one (case 1, Section 1.5). The above **sat** specification of a medium affected by message loss encompasses many realizations of *MED*, among which, in contrast, an equational specification would be forced to choose a particular one; the chosen *MED* could e.g. be one capable of losing every input message, or just every second one, or just a 0 and no other message, and so on.[8]

4.4.2 Media Deadlock Freedom: the General Case

A medium that cannot be used to transfer a message d is of course unacceptable. Therefore, in general, *MED* should never reach a state from which d cannot be

[8] This problem can be mitigated with a mixed approach, ultimately amenable to case 2, Section 1.5.

output after a suitable preamble including a single input of d. A formalization is best introduced by means of transitional semantics. Below, suppose $MED \stackrel{s}{\Longrightarrow} M$; then there must exist w such that:

$$w{\downarrow}src = d \tag{4-16}$$

$$w{\downarrow}end \subseteq (s{\downarrow}src) \cup d \tag{4-17}$$

$$M \stackrel{w\langle end!d\rangle}{\Longrightarrow} \tag{4-18}$$

These conditions warrant some comments, especially in order to dispel some possible misconceptions:

Remark 4.15

1. (4-17) can be understood as a result of the requirement that any message output must have been input earlier (whence $w{\downarrow}end \subseteq (sw){\downarrow}src$), combined with (4-16).

2. (4-16) rules out media that require multiple inputs of d before d can be output. An example of this could be an otherwise ideal MED that always throws away the first input message; for $s = \langle\rangle$ and M being MED, this makes (4-18) only possible with $w = \langle src!d, src!d\rangle$, against (4-16). Practical applications hardly ever employ such curious media, admitting inputs that have no potential effect; besides, if necessary, their formalization can be tackled anyway along the same lines of the case treated, assuming suitable bounds on the number of multiple inputs required to obtain an output.

3. On the other hand, remark 2 does not imply that an input message cannot be lost, thus requiring a second input to produce a single output. E.g. it is perfectly possible that:

   $$MED \stackrel{src!d}{\Longrightarrow} M,\ M \stackrel{end!d}{\not\Longrightarrow},\ M \stackrel{\langle src!d, end!d\rangle}{\Longrightarrow}$$

   But, in accordance with remark 2, it is also required that $MED \stackrel{src!d,end!d}{\Longrightarrow}$.

4. (4-16) and (4-18) do not prevent an s-derivative M of MED from inputting d and then losing it, reaching a state from which d cannot be output any

more; i.e. there may be $w_1, w_2 \in \mathsf{A}^*_{end}, M'$ such that:

$$M \stackrel{w_1\langle src!d\rangle w_2}{\Longrightarrow} M',\ M' \not\stackrel{w_3\langle end!d\rangle}{\Longrightarrow} \text{ for all } w_3 \in \mathsf{A}^*_{end}.$$

But of course also M' must be able to find its way to outputting d after inputting it again, just as formalized for M above.

5. (4-16) does not prevent M from outputting d without inputting it first (provided d had been input by *MED* with s). It is therefore possible that:

$$M \stackrel{w\langle end!d\rangle}{\Longrightarrow},\ w \in \mathsf{A}^*_{end}.$$ □

By the preceding discussion, we may define a function *pre* mapping every trace s, s-derivative M of *MED* and message d onto a trace $w = pre(s, M, d)$ such that:

$$w{\downarrow}src = d \qquad w{\downarrow}end \subseteq (s{\downarrow}src) \cup d \qquad M \stackrel{w\langle end!d\rangle}{\Longrightarrow}$$

We now let:

$$Pre(s, d) = \{pre(s, M, d) \mid MED \stackrel{s}{\Longrightarrow} M\}$$

Provided $MED \stackrel{s}{\Longrightarrow}$, this set must be non-empty and:

$$\forall s, M : MED \stackrel{s}{\Longrightarrow} M \Rightarrow \forall d : \exists w \in Pre(s, d) : \forall u \leqslant w\langle end!d\rangle : M \stackrel{u}{\Longrightarrow}$$

These conditions can be translated into the denotational-**sat** framework by requiring that

$$MED \textbf{ sat } M.LIV \tag{4-19}$$
$$M.LIV \equiv \forall d : \exists w \in Preamble(tr, d) : \forall u \leqslant w\langle end!d\rangle : u \notin Ref$$

and

Assumption 4.16 Every acceptable medium *MED* is associated with a function *Preamble*. This maps a trace s over $C = \{src, end\}$ and message d onto a finite, non-empty set $Preamble(s, d)$ of traces over C such that $w \in Preamble(s, d)$ implies:

1. $w{\downarrow}src = d$,

2. $w{\downarrow}end \subseteq (s{\downarrow}src) \cup \{d\}$. □

All these requirements, except the image-finiteness of *Preamble*, reflect those already identified operationally in the transitional framework (cf. (4-16) and (4-17)). The finiteness assumption ensures (by virtue of Proposition 4.5) that assertions containing *Preamble* are compact, thus permitting the application of the simpler hiding rule (4-2). This is generally needed for deadlock freedom verification of systems built around media satisfying (4-19), as discussed in Section 4.3.3 and especially in Remark 4.14. That remark and the experience of this work even suggest that for verification no information on media is necessary beyond those laid down in Assumption 4.16; in particular, an exact knowledge of function *Preamble* does not appear to be indispensable.

Justifying Preamble Image Finiteness

We shall now justify also the finiteness assumption for *Preamble* by showing that, for nearly all media, it can also be made for *Pre*, the operational counterpart of *Preamble*.

We first observe that:

$$Pre(s,d) \subseteq \{w \mid MED \stackrel{s\cdot w\cdot\langle end!d\rangle}{\Longrightarrow}, \; w{\downarrow}src = d\}$$

Hence it follows, using the (transitional) τ operator of Definition 3.83:

$$\begin{aligned} Pre(s,d) &\subseteq \{w_1\langle src!d\rangle w_2 \mid w_1, w_2 \in \mathsf{A}^*_{end}, \\ &\qquad\qquad sw_1\langle src!d\rangle w_2\langle end!d\rangle \in \tau MED\} \\ &\subseteq \{w_1\langle src!d\rangle w_2 \mid w_1 \in out(s), \\ &\qquad\qquad w_2\langle end!d\rangle \in out(sw_1\langle src!d\rangle)\} \end{aligned}$$

where we have defined:

$$out(s) = \{u \in \mathsf{A}^*_{end} \mid su \in \tau MED\}, \text{ for } s \in \tau MED$$

The tree $out(s)$ comprises all the output traces that *MED* can perform after trace s, and therefore is a very natural means of describing the characteristics of a medium. Whenever $out(s)$ is finite for all traces s of *MED*, the upper bound derived above

ensures finiteness for $Pre(s, d)$ too. This turns out to be the case for many important classes of media. E.g. a medium that may be affected only by message losses will have ($\preceq$ being the *subsequence* relation):

$$out(s) \subseteq \{u \in \mathsf{A}^*_{end} \mid (su)\downarrow end \preceq s\downarrow src\}$$

A medium that may also fail to preserve message ordering has:

$$out(s) \subseteq \{u \in \mathsf{A}^*_{end} \mid (su)\downarrow end \preceq t, \text{for some permutation } t \text{ of } s\downarrow src\}$$

However, if $out(s)$ is infinite the above argument cannot be used to infer that $Pre(s, d)$ is finite. An example is a medium allowing any loss, reordering and unbounded duplication of undelivered messages, so that:

$$out(s) = \{u \in \mathsf{A}^*_{end} \mid u\downarrow end \subseteq s\downarrow src\}$$

which is an infinite set.

Note however that $out(s)$ is still finite in the presence of bounded duplication. Normally, solving cardinality-related issues by placing bounds is not particularly attractive, because the bound chosen is apt to have a flavour of arbitrariness and furthermore to clutter the resulting specification and proof. The case at hand, however, does not suffer from this flaw, because the bound is encoded and 'encapsulated' in the image-finite function *Preamble*, and this one (as discussed above and in Remark 4.14) need not be known except for what stated in Assumption 4.16. This makes our approach more than satisfactory for practical applications.

However, we will also show how to tackle many cases in which $out(s)$ is infinite. We go back to the definition:

$$Pre(s, d) = \{pre(s, M, d) \mid MED \stackrel{s}{\Longrightarrow} M\}$$

First, we may count on many interesting media MED having, for all s, an image-finite "$MED \stackrel{s}{\Longrightarrow}$" relation (i.e. finitely many s-derivatives), which guarantees that the set $Pre(s, d)$ is finite.

If this fails to happen, recall that function *pre* need only satisfy, for all $w = pre(s, M, d)$:

$$w\downarrow src = d \qquad w\downarrow end \subseteq (s\downarrow src) \cup d \qquad M \stackrel{w\langle end!d\rangle}{\Longrightarrow}$$

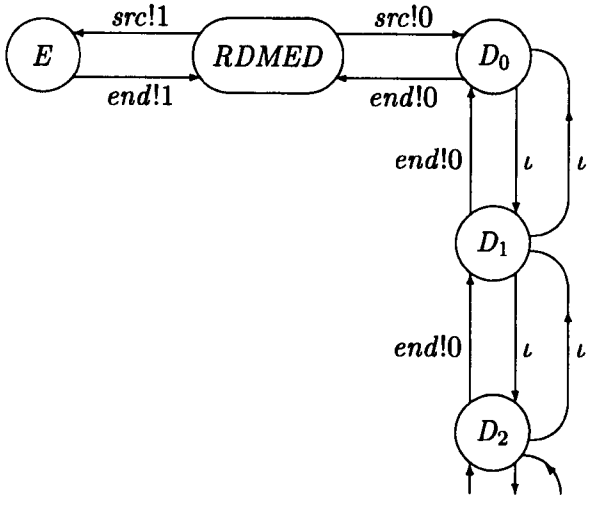


Figure 4.3: LTS describing the behaviours of medium *RDMED*.

Now, suppose that, for fixed s, d and each M such that $MED \stackrel{s}{\Longrightarrow} M$, there is a w as above, bounded in length by some integer, say $n(s)$. Then $pre(s, M, d)$ may be chosen to return this w, so that

$$\begin{aligned} Pre(s,d) &= \{pre(s,M,d) \mid MED \stackrel{s}{\Longrightarrow} M\} \\ &\subseteq (\{src!d, end!d\} \cup \{end!v \mid v \in s{\downarrow}src\})^{n(s)} \end{aligned}$$

and again $Pre(s, d)$ is finite. As explained above, in this case we do not view as a real limitation the recourse to a bound, since this remains implicit by virtue of the introduction of function *Preamble*. Thus, a great deal of the remaining media and applications may be catered for.

In particular, we may treat a medium affected by unbounded duplication, provided from each state it also has a route available along which duplication is bounded; in some sense this may be seen as duplication being reversible up to a bounded threshold. As an example, consider a one-bit buffer *RDMED* that duplicates an undelivered 0; for simplicity we do not define it as a process expression and work out its transitions instead from the LTS in Figure 4.3. We choose:

$$\begin{aligned} pre(s, RDMED, 1) &= \langle src!1 \rangle \\ pre(s, E, 1) &= \langle end!1, src!1 \rangle \\ pre(s, D_k, 1) &= \langle end!0, src!1 \rangle \end{aligned}$$

(we omit making the range of s precise, or showing that, for the relevant s, condition (4-17) is satisfied). As explained before, since *pre* is image-bounded, *Pre* will

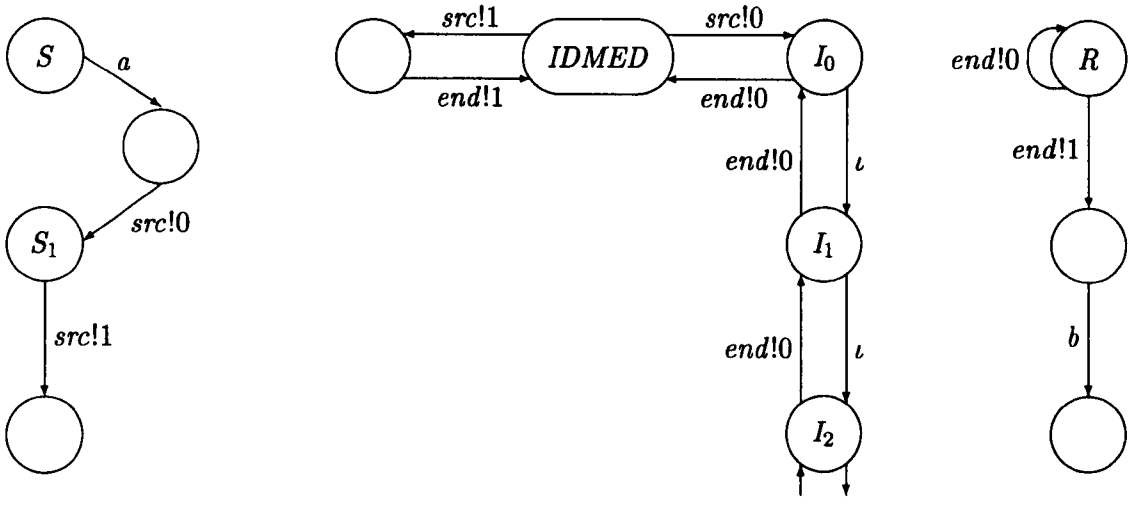


Figure 4.4: LTSs describing the behaviours of *IDMED*, S and R.

be image-finite.

It is worth noting that the internal transitions $D_{k+1} \stackrel{\iota}{\Longrightarrow} D_k$ that undo the duplication of bit 0 might as well be interpreted as losses of the duplicated bit, which—quite surprisingly—means that loss can obviate the undesired effects of duplication.

If these duplication-undoing transitions are removed from the behaviour of *RDMED*, for the resulting medium *IDMED* (Figure 4.4) we can't help defining for, say, $s = \langle end!0 \rangle$ and $k \geqslant 0$:

$$pre(s, I_k, 1) \;=\; \langle end!0 \rangle^{k+1} \langle src!1 \rangle$$

which clearly results in an infinite $Pre(s, 1)$ and ultimately in the impossibility of satisfying Assumption 4.16. Thus, deadlock freedom assertions for systems based on *IDMED* may fail to be compact, which makes verification in the style of Section 4.3.3 impossible. Actually, this is not a new problem, but, like many others before, can be traced to the adoption of finite refusals. Moreover, it should not represent an additional concern, because in our model the medium *IDMED* is of limited use anyway.

To see why, let *SIR* represent the parallel composition of *IDMED* with S and R described in Figure 4.4. Note that we may rely on these LTSs to determine failures and transitions of *SIR*, but for the failures of $HSIR = SIR \backslash \{src, end\}$ we need to

revert to the denotational definition of hiding, owing to its imperfect operational correspondence (cf. Proposition 3.99 and the discussion preceding it). Intuitively, S reacts to a by probing R through the medium with both a 0 and a 1, in the hope that R will respond to one of these bits by performing b, as it does indeed for a 1. This is intended to ensure for *HSIR* the external requirement:

$$HSIR \textbf{ sat } tr = a \Rightarrow b \notin Ref$$

However, it is not difficult to show that instead $\{b\} \in \rho HSIR(a)$, if the refusal function ρ is understood denotationally. Indeed, letting $C = \{src, end\}$, we have:

$$\forall Y \in \mathsf{p}Act^+ \colon Y \setminus C = \{b\} \Rightarrow Y \in \rho SIR(\langle a, src!0\rangle) \tag{4-20}$$

To see this, choose one such Y and observe that, since $Y \setminus C = \{b\}$, its traces must have the form $w'b\,w''$ with $w', w'' \in (\mathsf{A}_C)^*$; since Y is finite, let m be the maximum integer such that $w' = \langle end!0\rangle^{m+1}\langle src!1, end!1\rangle$, if any, or else 0. We now show that $Y \in \rho SIR(\langle a, src!0\rangle)$ (this ρ can be interpreted operationally because *SIR* does not contain hiding); Figure 4.4 should convince the reader that:

1. $SIR \stackrel{\langle a, src!0\rangle}{\Longrightarrow} Q$ iff Q is, for some $k \geqslant 0$, the composition $SIR_{1,k}$ of S_1, I_k and R;
2. $SIR_{1,k} \stackrel{w'\,b\,w''}{\Longrightarrow}$ for $w', w'' \in (\mathsf{A}_C)^*$ holds only if $w' = \langle end!0\rangle^{k+1}\langle src!1, end!1\rangle$.

Therefore $SIR_{1,m+1}$ cannot perform any trace of Y, and the desired (4-20) follows.

4.5 A Case Study: A Sliding Window Protocol

The **sat** calculus will now be applied to the verification of a system that provides reliable communication over an unreliable network. This goal is achieved by a fault-tolerant version of the *sliding-window* protocol, which is at the core of class 4 Transport protocol [ISO, 1988]. It should be stressed that the protocol to be studied is a non-trivial one and will be formally described at a realistic level, despite the omission of some features of a full-fledged Transport protocol, such as connection management, multiplexing or fragmenting.

4.5.1 Informal Description

The system studied is expected to behave like a finite buffer of length L ($L \geqslant 1$). More specifically, it must guarantee reliable communication satisfying the following requirements:

REL data conveyed are neither corrupted nor spuriously created, i.e. any data output must have been previously input;

SEQ data are output in the input order;

BUF data input and still undelivered are at most L;

INLIV the system cannot refuse to input data, unless **BUF** would be violated as a result;

OUTLIV the system cannot refuse to output undelivered data.

Note that compared to the example in Section 4.3, the requirements **BUF** is new and **INLIV** has been adjusted accordingly. This will result in considerable added complexity in the system architecture, the protocol governing it and, consequently, the overall treatment.

The system is built on top of a network that provides communication media satisfying the requirements described in Section 4.4, but otherwise unreliable. Therefore, such a medium satisfies the safety property (4-14), which is tantamount to **REL**, and the deadlock freedom property (4-19). On the other hand, it may lose, duplicate or reorder input data, thus violating **OUTLIV** and **SEQ**; **BUF** and **INLIV** are not guaranteed either.

In order to ensure the missing functionalities, a suitable layer must be added on top of the network. Again, adhering to protocol terminology we shall call *sdus* the data conveyed by the added layer on behalf of its users, and *pdus* the data exchanged by the entities that compose the layer. In our case study there are only two entities, a sender and a receiver. The sender accepts sdus from a user, while the receiver delivers sdus to another user. They also exchange pdus in both directions, through the network. Any pdu p issued by the sender carries a sdu d

and a sequence number n. Data representation is modelled as in Section 4.3, with three functions *pdu*, *sdu* and *seqno* which satisfy:

$$\begin{aligned} seqno(pdu(d,n)) &= n \\ sdu(pdu(d,n)) &= d \\ pdu(sdu(p), seqno(p)) &= p \end{aligned}$$

Pdus issued by the receiver are integers called called *acks* (acknowledgements).

The sender and the receiver obey a sliding window protocol [Tanenbaum, 1988], which is now sketched along with an informal derivation of the 'partial correctness' system properties **REL** and **BUF**.

Since the network satisfies **REL**, the receiver knows that the pdus it receives actually originate from the sender; it can also assume that

S.FIDEL If the sender issues a pdu p such that $sdu(p) = d$ and $seqno(p) = n$, then d was the nth sdu accepted by the sender.

Hence it follows that

R.RECOV the receiver has enough information to reconstruct the sequence of input data, guaranteeing **REL** and **SEQ** for the system.

Since the network satisfies **REL**, the sender knows that the acks it receives actually originate from the receiver; it can also assume that:

R.ACKLIM any ack issued by the receiver is not greater than the number of sdus delivered by the receiver.

Thus, for **BUF** to be satisfied, it is sufficient that:

S.INLIM the sender does not let the number of sdus accepted exceed the highest ack received by more than L.

4.5.2 A Plan of the Formal Treatment

Consider the formula p **sat** S, with p not a process constant. Proving its validity, ultimately involves the **sat** rule for the outermost operator of p; the premises of

this rule will include the formula p_α **sat** S_α for each generic operand p_α of p, and may otherwise refer to S_α but not p_α. Thus the desired conclusion p **sat** S can be inferred simply from assumptions ensuring p_α **sat** S_α for each α, and knowledge of S_α, but not p_α. This pattern of reasoning may of course be applied again to any operand p_α.

The ability to reason about a partially unknown process expression like p above, affords some important advantages:

- greater generality of the results obtained, which will apply to any process expression p whose operands p_α satisfy the specifications S_α;
- deferment of many implementation decisions, irrelevant for correctness, to late design stages (each p_α can be defined after the premises needed to infer p **sat** S have been established);
- the ability to study systems whose components (i.e. p_α) may be affected by largely unpredictable errors; such components are best collectively described through constraints like those introduced in Section 4.4 for media, rather than tied to specific—and arbitrary—realizations as process expressions;
- a means through which verifications of different process expressions may be combined without involving the process expressions themselves; this facilitates bottom-up, top-down and mixed approaches to design.

These observations suggest that a top-down approach to the formal analysis of a system can be conveniently structured in steps according to the following checklist.

1. The system is assumed to be modelled with a process expression having a suitable interface.
2. *External requirements specification.* Informal requirements on the external behaviour of the system are rendered as an assertion that it must satisfy.
3. *Architectural specification.* System components, in number and interface, and their interconnection and internal channels are identified.

4. *Hiding rule inference.* Rule (4-2) is employed to infer that the system satisfies its external specification. Note that among the premises of the rule there must be a **sat** formula for the parallel composition of system components; it may be identified through the strategy proposed in Section 4.3.3, and can be introduced at this stage as an assumption or follow from one—presumably another **sat** formula about the parallel composition.

5. *Component specification.* Components are formally specified in the **sat** logic.

6. *Parallel composition rule inference.* On the assumption that components satisfy their specifications, this rule is employed to infer that their composition satisfies the assumption made in step 4. Thus, that assumption may be discharged.

7. *Component implementation and verification.* Components to be implemented are defined as process expressions, which are then proved to satisfy the respective specifications, discharging the assumptions introduced in step 6.[9]

Of course this ordering and structuring is bound to be, to some extent, arbitrary, and ignores the cyclic refinements of real design. It is however fairly typical and, by leaving component implementation last, or even aside, enjoys the advantages noted earlier. It can also be readily rearranged in a different (partial) order, to reflect other design routes; e.g. if architecture and components (steps 3 and 5) are fixed, determining the properties of external system behaviour (step 2) becomes a goal. Note that, even in the present arrangement, architecture and components may result partly from design choices and partly from design requirements (e.g., as in our examples, that remote sites are to be connected through faulty media).

The specification steps 1, 2, 3 and 5 for our case study are carried out in Section 4.5.3. Being fairly orthogonal, safety and deadlock freedom verifications

[9]This step is skipped for components that need not or cannot be implemented as process expressions. Consider, e.g., a communication medium or an off-the-shelf module, known to the designer only through specifications; or a component that has been implemented and verified in advance (that is, a partially bottom-up design).

On the other hand, the component to be implemented may be complex enough to warrant a further, 'recursive' application of the checklist methodology.

are considered separately; for the latter, steps 4 and 6 are presented in considerable detail in Section 4.5.6. Finally, the component implementation step 7 is discussed in Section 4.5.7.

4.5.3 Formal Specification

Our example communication system will be modelled with a process expression referred to as *TSP* (for Transport Service Provider), which accepts sdus at channel *snd* and delivers them at *rcv*.

External Requirements

The informal requirements of Section 4.5.1 may be formalized thus in the **sat** logic:

$$TSP \textbf{ sat } PREFIX \wedge OUTLIV \wedge INLIV \wedge BUF \tag{4-21}$$

PREFIX is as in Section 4.3 and *OUTLIV* too (except for the form chosen); *INLIV* instead takes *BUF* into account:

$$\begin{aligned} PREFIX &\equiv tr{\downarrow}rcv \leqslant tr{\downarrow}snd \\ OUTLIV &\equiv \#(tr{\downarrow}rcv) < \#(tr{\downarrow}snd) \Rightarrow rcv!(tr{\downarrow}snd)[1{+}\#(tr{\downarrow}rcv)] \notin Ref \\ BUF &\equiv \#(tr{\downarrow}snd) \leqslant \#(tr{\downarrow}rcv){+}L \\ INLIV &\equiv \#(tr{\downarrow}snd) < \#(tr{\downarrow}rcv){+}L \Rightarrow \forall x : snd!x \notin Ref \end{aligned}$$

Architecture

The architecture of system *TSP* is depicted in Figure 4.5. Its internal components are modelled by process expressions *SENDER*, MED_P, MED_A and *RECEIVER*, interacting at the channels:

$$C_{PA} = \{src_P, end_P, src_A, end_A\}$$

The media MED_P and MED_A make up the communication network connecting *SENDER* and *RECEIVER*; they carry pdus from src_P to end_P and acks from src_A to end_A respectively. *SENDER* accepts sdus at *snd* and acks at end_A, and outputs pdus at src_P. *RECEIVER* outputs sdus at *rcv* and acks at src_A, and receives pdus

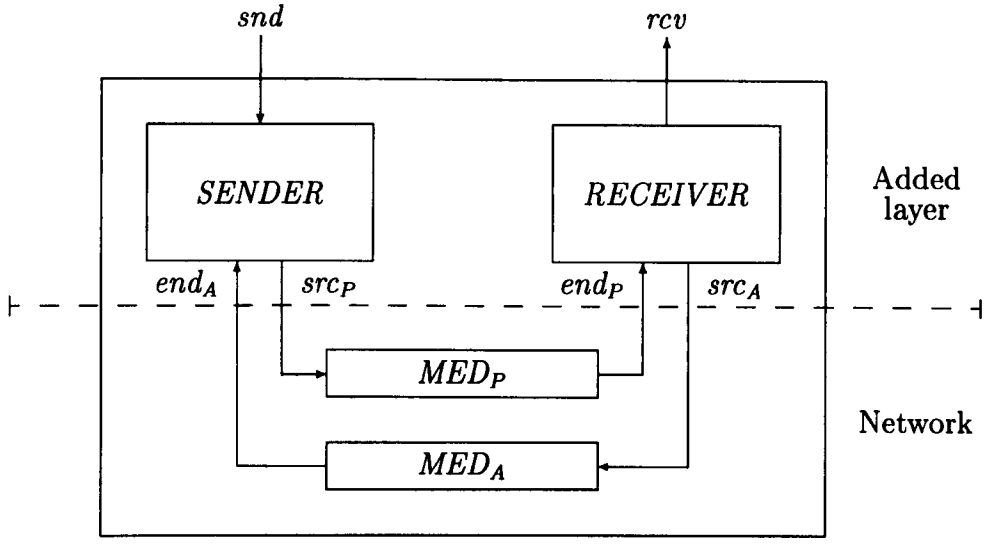


Figure 4.5: Architecture of the system *TSP*

at end_P. Thus, the interfaces of *SENDER*, MED_P, MED_A and *RECEIVER* are respectively the four channel sets:

$$C_S = \{snd, src_P, end_A\} \qquad C_P = \{src_P, end_P\}$$
$$C_A = \{src_A, end_A\} \qquad C_R = \{end_P, src_A, rcv\}$$

As in Section 4.3, we let:

$$\mathsf{A}_C = \{c!v \mid c \in C\} \qquad B_\alpha = \mathsf{A}_{C_\alpha}, \text{ for } \alpha = S, P, A, R$$

TSP can now be defined as:

$$TSP \equiv PSP \backslash C_{PA} \tag{4-22}$$
$$PSP \equiv (SENDER : C_S \parallel MED_P : C_P \parallel MED_A : C_A \parallel RECEIVER : C_R)$$

Component Specification and Basic Properties

Component specification represents step 5 in the design checklist of Section 4.5.2.

As observed before, one of the rewards of the **sat** approach is the ability to carry out partial specification and verification. Thus it will be possible to derive the desired property (4-21) for *TSP* simply by assuming

$$\begin{array}{ll} SENDER \textbf{ sat } T_S & MED_P \textbf{ sat } T_P \\ RECEIVER \textbf{ sat } T_R & MED_A \textbf{ sat } T_A \end{array} \tag{4-23}$$

for suitable T_S, T_P, T_A, T_R, and this result will hold for any choice of *SENDER*, MED_P, MED_A *RECEIVER* orderly satisfying them. Note that at this stage none of these expressions need be known or precisely identified by the respective assertion, which in turn does not have to be stronger than required by proofs. In fact, for the reasons examined in Section 4.5.2, the definition of *SENDER* and *RECEIVER* as process expressions is deferred to a later step, while that of MED_P, MED_A will not even be attempted.

We are not ready, yet, to formulate the full assertion T_α for any $\alpha = S, P, A, R$. For now we shall confine ourselves to some properties, referred to later as *basic*, that will play a role in the quest of the T_αs. Property names will obey the pattern "$\alpha\ldots$", in order to let them be immediately related to the process expression they specify.

The basic safety and liveness properties for media MED_P and MED_A are obtained from the pair *M.REL*, *M.LIV* defined for *MED* in Section 4.4, by renaming channels *src*, *end* as src_P, end_P and src_A, end_A respectively. The resulting assertions will be named *P.REL*, *P.LIV* and *A.REL*, *A.LIV*.

Assertions $S\ldots$ and $R\ldots$ below apply to *SENDER* and *RECEIVER* respectively; they express either 'safety' properties concerning only traces, or deadlock-freedom properties involving also one-action refusals.

Safety assertions *S.FIDEL* and *S.INLIM* formalize the corresponding requirements of Section 4.5.1 for the sender:

$$S.FIDEL \equiv \forall p \in (tr{\downarrow}src_P)\colon sdu(p) = (tr{\downarrow}snd)[seqno(p)]$$

$$S.INLIM \equiv \#(tr{\downarrow}snd) \leqslant \max(tr{\downarrow}end_A){+}L$$

Note that henceforth it will be assumed that $\max(\langle\rangle) = 0$.

S.SDU.LIV states that the sender should not refuse to input a sdu at *snd* so long as this does not violate *S.INLIM*:

$$S.SDU.LIV \equiv \#(tr{\downarrow}snd) < \max(tr{\downarrow}end_A){+}L \Rightarrow \forall x\colon snd!x \notin Ref$$

By *S.ACK.LIV* the sender should accept any ack at end_A:

$$S.ACK.LIV \equiv \forall n\colon end_A!n \notin Ref$$

By *S.PDU.LIV* the sender never refuses to send at src_P a pdu containing the oldest unacknowledged sdu (if any):

$$S.PDU.LIV \equiv \textbf{let } h := \max(tr{\downarrow}end_A) \textbf{ in } h < \#(tr{\downarrow}snd) \Rightarrow src_P!pdu((tr{\downarrow}snd)[h{+}1], h{+}1) \notin Ref$$

Safety assertions *R.RECOV* and *R.ACKLIM* formalize the corresponding informal requirements of Section 4.5.1 for the receiver:

$$R.RECOV \equiv \forall m\colon 1 \leqslant m \leqslant \#(tr{\downarrow}rcv) \Rightarrow pdu((tr{\downarrow}rcv)[m], m) \in (tr{\downarrow}end_P)$$

$$R.ACKLIM \equiv \forall n\colon n \in (tr{\downarrow}src_A) \Rightarrow n \leqslant \#(tr{\downarrow}rcv)$$

The receiver accepts pdus at end_P, ignoring those out-of-sequence and extracting from those expected a sdu that will be output at *rcv*. More specifically, let r be the number of sdus output; by *R.SDU.LIV*, the receiver does not refuse to output the sdu contained in the first pdu arrived with sequence number $r{+}1$:

$$R.SDU.LIV \equiv \textbf{let } r := \#(tr{\downarrow}rcv) \textbf{ in let } u := tr{\restriction}\mathsf{A}^{r+1}_{end_P} \textbf{ in } u \neq \langle\rangle \Rightarrow rcv!sdu(head(u{\downarrow}end_P)) \notin Ref$$

where, for i natural, it has been defined:

$$\mathsf{A}^{i}_{end_P} = \{end_P!q \mid seqno(q) = i\} \tag{4-24}$$

Note that in the simpler example of Section 4.3 the sdu output is instead the one carried by the latest pdu with the expected sequence number (these sdus will coincide anyway, in the complete system, by *S.FIDEL* and *P.REL*).

No data incoming at end_P is refused:

$$R.PDU.LIV \equiv \forall q\colon end_P!q \notin Ref$$

After receiving data at end_P, the receiver does not refuse to ack at src_A with the number of sdus output. However, note that the specification *R.ACK.LIV* does not force the receiver to acknowledge immediately (it may instead output sdus or keep accepting pdus, but will not withdraw its offer to acknowledge):

$$R.ACK.LIV \equiv last(tr{\restriction}\{end_P, src_A\}) \in \mathsf{A}_{end_P} \Rightarrow src_A!\#(tr{\downarrow}rcv) \notin Ref$$

4.5.4 Derived Component Properties

Motivation

In our experience, the single most difficult verification task is the application of the parallel composition inference rule (4-1) (step 6 of the checklist in Section 4.5.2). For the case at hand, letting p_S, p_P, p_A, p_R be the system components *SENDER*, MED_P, MED_A, *RECEIVER* respectively, and α range over S, P, A, R, we can instantiate rule (4-1) thus:

$$\frac{\begin{array}{c} p_\alpha \textbf{ sat } T_\alpha \text{ for all } \alpha, \\ (\bigwedge_\alpha (X_\alpha \in \mathrm{pAct}^+ \wedge T_\alpha(s\restriction C_\alpha, X_\alpha))) \Rightarrow T(s\restriction \bigcup_\alpha C_\alpha, \overline{\|}_\alpha X_\alpha{:}C_\alpha) \end{array}}{\|_\alpha p_\alpha{:}C_\alpha \textbf{ sat } T} \tag{4-25}$$

This rule will be applied twice, in Lemmata 4.28 and 4.29, with T defined so as to be employed in the verification of deadlock freedom properties *INLIV* and *OUTLIV* respectively. If each T_α were simply the conjunction of the basic $\alpha\ldots$ assertions introduced earlier, the left side of the second premise of rule (4-25) would be too weak to imply the right side $T(s\restriction \bigcup_\alpha C_\alpha, \overline{\|}_\alpha X_\alpha{:}C_\alpha)$, thus preventing application of the rule. This is because T must also depend on extended refusals, whereas the basic assertions only refer to one-action refusals.

In connection with the problem of strengthening the assertions T_α and in the light of the checklist in Section 4.5.2 (steps 6, 7), it is now necessary to choose the role of the T_αs in the treatment. There are essentially two alternatives:

1. to assume p_α satisfies T_α;

2. to assume p_α satisfies just the basic $\alpha\ldots$ assertions, and thereby establish the required T_α by exploiting consistency rules (4-4), (4-5), (4-6).

The latter approach is, in our view, preferable, essentially because it is more in keeping with the decision deferring spirit advocated in Section 4.5.2. With this choice, the bulk of deadlock freedom verification (related to (4-25)) can be tackled with a comparatively small set of assumptions; this ensures a high degree of confidence in the results obtained, throughout the entire treatment, even before the process expressions p_α have been defined. In contrast, the former approach

needs uncomfortably heavy assumptions, which can only be discharged after defining each p_α, if possible, or else never; e.g. with reference to Table 4.1, satisfaction of assertion $\alpha.t.\pi.LIV$ by medium MED_α could have only been assumed, but not proved (which we instead do in Lemma 4.24).

As for ease of use, the second approach also seems to have an edge. Within it, if component p_α is recursively defined, recourse to delicate fixpoint induction will only be required to prove the basic assumptions on p_α, rather than the full T_α. Furthermore, the proof of T_α with consistency rules is not burdened with the irrelevant detail possibly added by the definition of p_α. In sum, we believe that in the preferred approach establishing the premises of rule (4-25) is easier; moreover it is in fact only indispensable, unless component implementation is the ultimate goal, for non-basic properties, the basic ones being self-evident. The only price to pay when working with consistency rules is an occasional explicit recourse to integer/structural induction, which in the other approach is disguised by the application of the fixpoint rule to fully defined recursive process expressions.

Technical Results

As argued above, we shall need:

Assumption 4.17 Components *SENDER*, MED_A, MED_P and *RECEIVER* of *TSP* satisfy the basic properties formalized at the end of Section 4.5.3 (refer back to (4-14) and (4-19) for the paradigms reproduced by the media basic properties). □

We are now able to formulate the full component specifications T_S, T_P, T_A, T_R referred to in (4-23) and needed for an effective application of inference rule (4-25).

Proposition 4.18 The following **sat** formulae hold:

$$SENDER \textbf{ sat } T_S \qquad MED_P \textbf{ sat } T_P \qquad MED_A \textbf{ sat } T_A \qquad RECEIVER \textbf{ sat } T_R$$

with each T_α ($\alpha = S, P, A, R$) defined as the conjunction of assertions $\alpha\ldots$ in Table 4.1.

Proof. Satisfaction of basic assertions has been assumed. Satisfaction of the remaining ones is proved in the rest of this section, except for those whose proof is straightforward or similar to others provided. □

$$
\begin{array}{rcl}
S.FIDEL & \equiv & \forall p \in (tr{\downarrow}src_P)\colon sdu(p) = (tr{\downarrow}snd)[seqno(p)] \\
S.INLIM & \equiv & \#(tr{\downarrow}snd) \leqslant \max(tr{\downarrow}end_A){+}L \\
S.SDU.LIV & \equiv & \#(tr{\downarrow}snd) < \max(tr{\downarrow}end_A){+}L \Rightarrow \forall x\colon snd!x \notin Ref \\
S.ACK^*.LIV & \equiv & \forall w \in (\mathsf{A}_{end_A})^*\colon w \notin Ref \\
S.PDU.LIV & \equiv & \textbf{let } h := \max(tr{\downarrow}end_A) \textbf{ in } h < \#(tr{\downarrow}snd) \Rightarrow \\
 & & src_P!pdu((tr{\downarrow}snd)[h{+}1], h{+}1) \notin Ref \\
S.1.LIV & \equiv & \textbf{let } s := tr{\downarrow}snd \textbf{ in let } m := \#s,\ h := \max(tr{\downarrow}end_A) \textbf{ in} \\
 & & \forall u\colon u = \langle\rangle \vee h{<}m \wedge (u = \langle src_P!pdu(s[h{+}1], h{+}1)\rangle) \Rightarrow \\
 & & \forall w \in (\mathsf{A}_{end_A})^*\colon \\
 & & \quad (\forall n\colon m < n{+}L \Rightarrow \forall x\colon u{\cdot}w\langle end_A!n, snd!x\rangle \notin Ref) \\
 & & \wedge\ (\forall r\colon u{\cdot}w\langle end_A!r\rangle \notin Ref) \\
 & & \wedge\ (\forall r\colon h{\leqslant}r{<}m \wedge \max(w{\downarrow}end_A) \leqslant r \Rightarrow \\
 & & \quad u{\cdot}w\langle end_A!r, src_P!pdu(s[r{+}1], r{+}1)\rangle \notin Ref) \\
R.RECOV & \equiv & \forall m\colon 1{\leqslant}m{\leqslant}\#(tr{\downarrow}rcv) \Rightarrow pdu((tr{\downarrow}rcv)[m], m) \in (tr{\downarrow}end_P) \\
R.ACKLIM & \equiv & \forall n\colon n \in (tr{\downarrow}src_A) \Rightarrow n \leqslant \#(tr{\downarrow}rcv) \\
R.SDU.LIV & \equiv & \textbf{let } r := \#(tr{\downarrow}rcv) \textbf{ in let } u := tr{\restriction}\mathsf{A}^{r+1}_{end_P} \textbf{ in} \\
 & & u \neq \langle\rangle \Rightarrow rcv!sdu(head(u{\downarrow}end_P)) \notin Ref \\
R.PDU^*.LIV & \equiv & \forall w \in (\mathsf{A}_{end_P})^*\colon w \notin Ref \\
R.ACK.LIV & \equiv & last(tr{\restriction}\{end_P, src_A\}) \in \mathsf{A}_{end_P} \Rightarrow src_A!\#(tr{\downarrow}rcv) \notin Ref \\
R.1.LIV & \equiv & \forall q\colon \langle end_P!q, src_A!\#(tr{\downarrow}rcv)\rangle \notin Ref \\
R.2.LIV & \equiv & \textbf{let } r := \#(tr{\downarrow}rcv) \textbf{ in } tr{\restriction}\mathsf{A}^{r+1}_{end_P} = \langle\rangle \Rightarrow \\
 & & \forall z\colon \forall q\colon \forall w \in (\mathsf{A}_{end_P})^*\colon (\langle end_P!q\rangle w){\restriction}\mathsf{A}^{r+1}_{end_P} = \langle\rangle \Rightarrow \\
 & & \quad w\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Ref \\
 & & \wedge\ \langle end_P!q\rangle w\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Ref \\
 & & \wedge\ \langle end_P!q, src_A!r\rangle w\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Ref \\
\alpha.REL & \equiv & tr{\downarrow}end_\alpha \subseteq tr{\downarrow}src_\alpha \\
\alpha.LIV & \equiv & \forall d\colon \exists w \in Preamble_\alpha(tr, d)\colon \forall u \leqslant w\langle end_\alpha!d\rangle\colon u \notin Ref \\
\alpha.\pi.LIV & \equiv & \forall d\colon \pi(d) \Rightarrow \exists w \in Preamble_\alpha(tr, d)\colon \\
 & & \exists w' \leqslant w\colon \exists d' \in (tr{\downarrow}src_\alpha){\cup}\{d\}\colon \\
 & & \quad (\neg\pi)(w'{\downarrow}end_\alpha) \wedge \pi(d') \wedge w'\langle end_\alpha!d'\rangle \notin Ref \\
\alpha.t.LIV & \equiv & \forall p\colon \exists u \leqslant \langle src_\alpha!p\rangle\colon \exists q \in (tr{\downarrow}src_\alpha){\cup}\{p\}\colon u\langle end_\alpha!q\rangle \notin Ref \\
\alpha.t.\pi.LIV & \equiv & \forall p\colon \exists u \leqslant \langle src_\alpha!p\rangle\colon \exists q \in (tr{\downarrow}src_\alpha){\cup}\{p\}\colon u\langle end_\alpha!q\rangle \notin Ref \wedge \\
 & & \forall d\colon \pi(d) \Rightarrow \exists w \in Preamble_\alpha(tr{\cdot}u\langle end_\alpha!q\rangle, d)\colon \\
 & & \exists w' \leqslant w\colon \exists d' \in ((tr{\cdot}u){\downarrow}src_\alpha){\cup}\{d\}\colon \\
 & & \quad (\neg\pi)(w'{\downarrow}end_\alpha) \wedge \pi(d') \wedge u\langle end_\alpha!q\rangle w'\langle end_\alpha!d'\rangle \notin Ref
\end{array}
$$

Table 4.1: Properties of the sender, the receiver and medium M_α.

It should be noted that all the following results presuppose Assumption 4.17. We begin with some about the sender:

Lemma 4.19 *SENDER* **sat** $S.ACK^*.LIV$, where

$$S.ACK^*.LIV \equiv \forall w \in (\mathsf{A}_{end_A})^*: w \notin Ref$$

Proof. By structural induction on w.

Basis: *SENDER* **sat** $\langle\rangle \notin Ref$ follows from rule (4-3).

Step: let $a \in \mathsf{A}_{end_A}$; we may assume *SENDER* **sat** S holds for $S \equiv w \notin Ref \wedge a \notin Ref$, the first conjunct being the induction hypothesis and the second $S.ACK.LIV$. We may now apply consistency rule (4-4), with H true, $T(t, X) \equiv a \notin X$ and $y(t) \equiv a$ to infer *SENDER* **sat** $w \notin Ref \div a$, i.e., as desired, *SENDER* **sat** $\langle a \rangle w \notin Ref$. □

Lemma 4.20 *SENDER* **sat** $S.1.LIV$, where $S.1.LIV$ is:

$$\begin{array}{l}
\textbf{let } s := tr{\downarrow}snd \textbf{ in let } m := \#s,\ h := \max(tr{\downarrow}end_A) \textbf{ in} \\
\forall u: u = \langle\rangle \vee h{<}m \wedge (u = \langle src_P!pdu(s[h{+}1], h{+}1)\rangle) \Rightarrow \\
\forall w \in (\mathsf{A}_{end_A})^*: \\
\quad (\forall n: m < n{+}L \Rightarrow \forall x: u{\cdot}w\langle end_A!n, snd!x\rangle \notin Ref) \\
\wedge\ (\forall r: u{\cdot}w\langle end_A!r\rangle \notin Ref) \\
\wedge\ (\forall r: h{\leqslant}r{<}m \wedge \max(w{\downarrow}end_A) \leqslant r \Rightarrow \\
\qquad u{\cdot}w\langle end_A!r, src_P!pdu(s[r{+}1], r{+}1)\rangle \notin Ref)
\end{array}$$

Proof. $S.1.LIV$ is easily split into three conjuncts, which are dealt with separately below.

1. Lemma 4.19 implies *SENDER* **sat** S_1, for

$$S_1 \equiv \forall w \in (\mathsf{A}_{end_A})^*: \forall n: w\langle end_A!n\rangle \notin Ref$$

The $\forall$-consistency rule (4-5) can be applied with H true, $S \equiv S_1 \wedge S.SDU.LIV$,

$$Y(t) \equiv \{w\langle end_A!n\rangle \mid w \in (\mathsf{A}_{end_A})^*\},$$

and $T(t, X) \equiv y \notin X$, to derive:

$$\begin{array}{l}
SENDER \textbf{ sat let } m := \#(tr{\downarrow}snd),\ h := \max(tr{\downarrow}end_A) \textbf{ in} \\
\qquad \forall w \in (\mathsf{A}_{end_A})^*: \forall n: m < \max\{h, n, \max(w{\downarrow}end_A)\}{+}L \Rightarrow \\
\qquad\quad \forall x: w\langle end_A!n, snd!x\rangle \notin Ref
\end{array}$$

Let S_2 be this assertion weakened by replacing "$m < \max\{h, n, \max(w{\downarrow}end_A)\}{+}L$" with "$m < n{+}L$"; then the consequence rule entails *SENDER* **sat** S_2, which is the first conjunct of $S.1.LIV$ for $u = \langle\rangle$.

Now, consistency rule (4-4) is applied with

$$\begin{aligned} S &\equiv S_2 \wedge S.PDU.LIV \\ y(t) &\equiv \textbf{let } h := \max(t{\downarrow}end_A) \textbf{ in } \langle src_P!pdu((t{\downarrow}snd)[h{+}1], h{+}1)\rangle \\ H(t,X) &\equiv \max(t{\downarrow}end_A) < \#(t{\downarrow}snd) \\ T(t,X) &\equiv H(t,X) \Rightarrow y(t) \notin X. \end{aligned}$$

The result is:

$$\begin{aligned} SENDER \textbf{ sat let } & m := \#(tr{\downarrow}snd),\ h := \max(tr{\downarrow}end_A) \textbf{ in } h < m \Rightarrow \\ & \textbf{let } u := \langle src_P!pdu((tr{\downarrow}snd)[h{+}1], h{+}1)\rangle \textbf{ in} \\ & \forall w \in \mathsf{A}^*_{end_A} : \forall n : m < n{+}L \Rightarrow \forall x : u{\cdot}w\langle end_A!n, snd!x\rangle \notin Ref \end{aligned}$$

which completes the proof concerning the first conjunct of $S.1.LIV$.

2. We omit the straightforward proof that *SENDER* satisfies the second conjunct of $S.1.LIV$.

3. For the third conjunct of $S.1.LIV$, we shall apply again the $\forall$-consistency rule (4-5), this time with assertion S defined as:

$$\begin{aligned} & (\forall w \in (\mathsf{A}_{end_A})^* : \forall r : w\langle end_A!r\rangle \notin Ref) \wedge \\ & (\textbf{let } s := tr{\downarrow}snd,\ k := \max(tr{\downarrow}end_A) \textbf{ in} \\ & \quad k < \#s \Rightarrow src_P!pdu(s[k{+}1], k{+}1) \notin Ref) \end{aligned}$$

SENDER satisfies the first conjunct of this S by Lemma 4.19; the second conjunct is simply $S.PDU.LIV$. To complete the premises of the said rule, let H be true, $T(t,X)$ be $y \notin X$, and:

$$Y(t) \equiv \{w\langle end_A!r\rangle \mid w \in (\mathsf{A}_{end_A})^*,\ r \geqslant \max((tw){\downarrow}end_A)\}$$

So, by the rule, *SENDER* satisfies the assertion:

$$\begin{aligned} & \forall y \in \{w\langle end_A!r\rangle \mid w \in (\mathsf{A}_{end_A})^*,\ r \geqslant \max((tr{\cdot}w){\downarrow}end_A)\} : \\ & \textbf{let } s := tr{\downarrow}snd,\ k := \max((tr{\cdot}y){\downarrow}end_A) \textbf{ in} \\ & \quad k < \#s \Rightarrow src_P!pdu(s[k{+}1], k{+}1) \notin Ref \div y \end{aligned}$$

This is easily recognized to imply the third conjunct of $S.1.LIV$ for $u = \langle\rangle$.

The case that u is not $\langle\rangle$ is easily catered for by combining the last result with $S.PDU.LIV$ through a consistency rule. □

For the receiver, the most interesting derived property is $R.2.LIV$:

Lemma 4.21 *RECEIVER* **sat** $R.2.LIV$, where:

$$\begin{aligned} R.2.LIV \;\equiv\; & \textbf{let } r := \#(tr{\downarrow}rcv) \textbf{ in } tr{\restriction}\mathsf{A}^{r+1}_{end_P} = \langle\rangle \Rightarrow \\ & \forall z\colon \forall q\colon \forall w \in (\mathsf{A}_{end_P})^*\colon (\langle end_P!q\rangle w){\restriction}\mathsf{A}^{r+1}_{end_P} = \langle\rangle \Rightarrow \\ & \quad w\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Ref \\ & \wedge\; \langle end_P!q\rangle w\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Ref \\ & \wedge\; \langle end_P!q, src_A!r\rangle w\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Ref \end{aligned}$$

Proof. (Recall $\mathsf{A}^{r+1}_{end_P} = \{end_P!q \mid seqno(q){=}r{+}1\}$, as defined in (4-24)). We begin by applying consistency rule (4-5), with S defined as the conjunction of an instance of $R.PDU.LIV$ and $R.SDU.LIV$, as follows:

$$\begin{aligned} & \textbf{let } r := \#(tr{\downarrow}rcv) \textbf{ in} \\ & \quad \forall w \in (\mathsf{A}_{end_P})^*\colon \forall z\colon w\langle end_P!pdu(z, r{+}1)\rangle \notin Ref \\ & \wedge\; (tr{\restriction}\mathsf{A}^{r+1}_{end_P} \neq \langle\rangle \Rightarrow rcv!sdu(head(tr{\restriction}\mathsf{A}^{r+1}_{end_P}{\downarrow}end_P)) \notin Ref) \end{aligned}$$

To complete the premises of the said rule, let:

$$\begin{aligned} Y(t) \;\equiv\; & \textbf{let } r := \#(t{\downarrow}rcv) \textbf{ in} \\ & \quad \{w\langle end_P!pdu(z, r{+}1)\rangle \mid w \in (\mathsf{A}_{end_P})^*, w{\restriction}\mathsf{A}^{r+1}_{end_P} = \langle\rangle\} \\ H(t, X) \;\equiv\; & \textbf{let } r := \#(t{\downarrow}rcv) \textbf{ in } t{\restriction}\mathsf{A}^{r+1}_{end_P} = \langle\rangle \\ T(t, X) \;\equiv\; & H(t, X) \Rightarrow y \notin X. \end{aligned}$$

So, by the rule, *RECEIVER* satisfies the assertion:

$$\begin{aligned} & \textbf{let } r := \#(tr{\downarrow}rcv) \textbf{ in } tr{\restriction}\mathsf{A}^{r+1}_{end_P} = \langle\rangle \Rightarrow \\ & \forall y \in \{w\langle end_P!pdu(z, r{+}1)\rangle \mid w \in (\mathsf{A}_{end_P})^*, w{\restriction}\mathsf{A}^{r+1}_{end_P} = \langle\rangle\}\colon \\ & \quad (tr{\cdot}y){\restriction}\mathsf{A}^{r+1}_{end_P} \neq \langle\rangle \Rightarrow rcv!sdu(head((tr{\cdot}y){\restriction}\mathsf{A}^{r+1}_{end_P}{\downarrow}end_P)) \notin Ref{\div}y \end{aligned}$$

In the previous assertion, replacing $(tr{\cdot}y){\restriction}\mathsf{A}^{r+1}_{end_P}$ by $end_P!pdu(z, r{+}1)$ yields the first conjunct in $R.2.LIV$. The other two are easily derived by applying consistency rules to combine the previous assertion with, respectively, $R.PDU.LIV$ and $R.1.LIV$. □

Finally, we present some derived properties of the media MED_P, MED_A. They will be referred to a generic MED_α with α ranging over P, A.

Lemma 4.22 Let π be a unary predicate on data. Then MED_α **sat** $\alpha.\pi.LIV$, where

$$\begin{aligned}\alpha.\pi.LIV \;\equiv\; & \forall d\colon \pi(d) \Rightarrow \exists w \in Preamble_\alpha(tr, d)\colon \\ & \exists w' \leqslant w\colon \exists d' \in (tr{\downarrow}src_\alpha){\cup}\{d\}\colon \\ & \quad (\neg\pi)(w'{\downarrow}end_\alpha) \wedge \pi(d') \wedge w'\langle end_\alpha ! d'\rangle \notin Ref\end{aligned}$$

Proof. Note that here predicate $\neg\pi$ is being element-wise extended to data sequences, and therefore $(\neg\pi)(\langle\rangle)$ always holds true. By the consequence rule, it is enough to infer $\alpha.\pi.LIV$ from $\alpha.LIV$ (cf. Table 4.1). Let then $\pi(d)$ hold true and, by $\alpha.LIV$, choose $w \in Preamble_\alpha(tr, d)$ such that:

(1) $\forall u \leqslant w\langle end_\alpha ! d\rangle\colon u \notin Ref$.

If $(\neg\pi)(w{\downarrow}end_\alpha)$, taking $w' = w$ and $d' = d$ yields $\alpha.\pi.LIV$. Otherwise, if π holds on some component of $w{\downarrow}end_\alpha$, there must be w' and d' such that:

(2) $w'\langle end_\alpha ! d'\rangle \leqslant w$, $(\neg\pi)(w'{\downarrow}end_\alpha)$, and $\pi(d')$.

With (1) and (by Assumption 4.16) $d' \in (tr{\downarrow}src_\alpha){\cup}\{d\}$, this ensures that $\alpha.\pi.LIV$ holds. □

An easy consequence is:

Corollary 4.23 MED_α **sat** $\alpha.t.LIV$, where

$$\alpha.t.LIV \;\equiv\; \forall p\colon \exists u \leqslant \langle src_\alpha ! p\rangle\colon \exists q \in (tr{\downarrow}src_\alpha){\cup}\{p\}\colon u\langle end_\alpha ! q\rangle \notin Ref$$

Proof. Let π be the identically true predicate. Then $\alpha.\pi.LIV$ with the bound variables d, w', d' renamed as p, u, q can only be true for instances of u such that $u{\downarrow}end_\alpha = \langle\rangle$ (to make $(\neg\pi)(u{\downarrow}end_\alpha)$ true). Moreover, since $u \leqslant w$ for some $w \in Preamble_\alpha(tr, p)$, and $w{\downarrow}src_\alpha = p$ by Assumption 4.16, it follows that $u \leqslant \langle src_\alpha ! p\rangle$.

Thus $\alpha.t.LIV$ is a consequence of $\alpha.\pi.LIV$, and the corollary follows from Lemma 4.22 by the consequence rule. □

We conclude this section with an application of consistency rules to MED_α:

Lemma 4.24 For a unary predicate π, MED_α **sat** $\alpha.t.\pi.LIV$, where

$$\begin{aligned}\alpha.t.\pi.LIV \;\equiv\; & \forall p : \exists u \leqslant \langle src_\alpha!p\rangle : \exists q \in (tr{\downarrow}src_\alpha)\cup\{p\} : u\langle end_\alpha!q\rangle \notin Ref \wedge \\ & \forall d : \pi(d) \Rightarrow \exists w \in Preamble_\alpha(tr{\cdot}u\langle end_\alpha!q\rangle, d) : \\ & \exists w' \leqslant w : \exists d' \in ((tr{\cdot}u){\downarrow}src_\alpha)\cup\{d\} : \\ & \quad (\neg\pi)(w'{\downarrow}end_\alpha) \wedge \pi(d') \wedge u\langle end_\alpha!q\rangle w'\langle end_\alpha!d'\rangle \notin Ref\end{aligned}$$

Proof. The proof is an application of the $\exists$-consistency rule (4-6), with S being the conjunction of: $\alpha.t.LIV$ (with "$\forall p$" dropped and p fixed), $\alpha.\pi.LIV$, and $\langle\rangle \notin Ref$. Further, let H be true and:

$$\begin{aligned}Y_p(t) \;&=\; \{u\langle end_\alpha!q\rangle \mid u \leqslant \langle src_\alpha!p\rangle,\; q \in (t{\downarrow}src_\alpha)\cup\{p\}\} \\ T(t,X) \;&\equiv\; \exists y \in Y_p(t) : y \notin X\end{aligned}$$

By the rule it follows that, for the p chosen:

$$MED_\alpha \textbf{ sat } \exists y \in Y_p(tr) : \langle\rangle \notin Ref \wedge \alpha.\pi.LIV(tr{\cdot}y, Ref \div y)$$

where $\alpha.\pi.LIV(tr{\cdot}y, Ref \div y)$ can be written:

$$\begin{aligned}&\forall d : \pi(d) \Rightarrow \exists w \in Preamble_\alpha(tr{\cdot}y, d) : \\ &\exists w' \leqslant w : \exists d' \in ((tr{\cdot}y){\downarrow}src_\alpha)\cup\{d\} : \\ &\quad (\neg\pi)(w'{\downarrow}end_\alpha) \wedge \pi(d') \wedge yw'\langle end_\alpha!d'\rangle \notin Ref\end{aligned}$$

Hence, recalling the form of $Y_p()$ and reintroducing the quantifier, we see that MED_α satisfies $\alpha.t.\pi.LIV$. □

4.5.5 System Safety Verification

The 'safety' system properties REL and BUF, and the component properties whence they follow, only constrain process traces; as a result, they can be expressed and reasoned about in a subsystem that falls entirely into the original **sat** logic of [Hoare, 1985].

No new technique or concept is therefore involved in safety verification. Thus, and for the sake of brevity, we will omit the relevant proofs, which the interested

reader can find in [Carchiolo, Faro, & Pappalardo, 1992]. It will suffice here to mention that, as suggested by the informal reasoning sketched at the end of Section 4.5.1, *REL* can be inferred from *S.FIDEL*, *P.REL* and *R.RECOV*, and *BUF* from *S.INLIM*, *A.REL* and *R.ACKLIM*.

The same reasoning should also justify the introduction of two useful safety properties of *PSP*, the parallel composition of system components. The first property requires the highest ack h received by the sender to be less than the number of sdus output by the receiver, and the number of sdus accepted by the sender not to exceed $h+L$. The second property ensures that any pdu in transit through MED_P with sequence number n should carry the nth sdu accepted by the sender. Formally:

Proposition 4.25

$$PSP \textbf{ sat } \textbf{let } m := \#(tr{\downarrow}snd),\ h := \max(tr{\downarrow}end_A),\ r := \#(tr{\downarrow}rcv) \textbf{ in } h \leqslant r \wedge m \leqslant h+L$$

$$PSP \textbf{ sat } \forall z, n\colon pdu(z,n) \in (tr{\downarrow}src_P) \cup (tr{\downarrow}end_P) \Rightarrow z = (tr{\downarrow}snd)[n] \qquad \square$$

4.5.6 System Deadlock Freedom Verification

The Hiding Rule Inference

This is step 4 in the checklist of Section 4.5.2. It represents the core of deadlock freedom verification, being directly involved with properties of external system behaviour. Our immediate goal is now to establish:

$$TSP \textbf{ sat } INLIV \wedge OUTLIV$$

where, by (4-22), *TSP* is $PSP \backslash C_{PA}$, $C_{PA} = \{src_P, end_P, src_A, end_A\}$, and:

$$INLIV \equiv \textbf{let } m := \#(tr{\downarrow}snd),\ r := \#(tr{\downarrow}rcv) \textbf{ in } m < r+L \Rightarrow snd!x \notin Ref$$

$$OUTLIV \equiv \textbf{let } m := \#(tr{\downarrow}snd),\ r := \#(tr{\downarrow}rcv) \textbf{ in } r < m \Rightarrow rcv!(tr{\downarrow}snd)[r+1] \notin Ref$$

According to the strategy devised in Section 4.3.3, we invoke Proposition 4.13 to derive *TSP* **sat** *INLIV* and *TSP* **sat** *OUTLIV* from, respectively:

$$PSP \textbf{ sat } \textbf{let } m := \#(tr{\downarrow}snd),\ r := \#(tr{\downarrow}rcv) \textbf{ in} \quad (4\text{-}26)$$
$$m < r{+}L \Rightarrow \exists w \in WI(tr)\colon w\langle snd!x\rangle \notin Ref$$

$$PSP \textbf{ sat } \textbf{let } m := \#(tr{\downarrow}snd),\ r := \#(tr{\downarrow}rcv) \textbf{ in} \quad (4\text{-}27)$$
$$r < m \Rightarrow \exists w \in WO(tr)\colon w\langle rcv!(tr{\downarrow}snd)[r{+}1]\rangle \notin Ref$$

where, for the appropriate s, $WI(s)$, $WO(s)$ must be finite and non-empty sets of actions over the channels C_{PA}.

Along the lines indicated in Section 4.3.3, we should proceed to the identification of $WI(\,)$ and $WO(\,)$. For $WI(\,)$, we observe that *PSP* should be able to perform traces of the following forms:

1. $snd!x$

2. $u\langle end_P!q\rangle w'_A\langle end_A!n, snd!x\rangle \notin Ref$, where u is $\langle\rangle$ or an input to MED_P, and w'_A is made of outputs from MED_A and perhaps the input of ack n to it.

Let m be the number of sdus accepted by the sender, and h be the highest ack it has received; it is quite obvious that *PSP* will not refuse to perform $snd!x$ if $m < h{+}L$; otherwise, *PSP* should not refuse a trace of the second form above; this can be realized through a series of informal arguments that will not be reported here, but can be recognized in the proof of Lemma 4.28 below. Analogous considerations apply to the identification of $WO(\,)$ (the relevant informal arguments are reflected by the proof of Lemma 4.29).

As discussed in Section 4.3.3, the preceding informal analysis suggests stipulating assumptions that are slightly different from the current obligations (4-26) and (4-27):

Assumption 4.26

$$PSP \textbf{ sat } \textbf{let } m := \#(tr{\downarrow}snd),\ h := \max(tr{\downarrow}end_A),\ r := \#(tr{\downarrow}rcv) \textbf{ in}$$
$$m < h{+}L \vee m = h{+}L \wedge h < r \Rightarrow \exists w \in WI(tr)\colon w\langle snd!x\rangle \notin Ref$$

$$
\begin{aligned}
PSP \textbf{ sat let } & m := \#(tr{\downarrow}snd),\ h := \max(tr{\downarrow}end_A),\ r := \#(tr{\downarrow}rcv) \textbf{ in} \\
& h \leqslant r < m \Rightarrow \textbf{let } x := (tr{\downarrow}snd)[r{+}1] \textbf{ in} \\
& \exists z\colon (z = x \vee pdu(z, r{+}1) \in (tr{\downarrow}src_P) \cup (tr{\downarrow}end_P)) \wedge \\
& \exists w \in WO_z(tr)\colon w\langle rcv!z\rangle \notin Ref
\end{aligned}
$$

where, for the appropriate s and z, $WI(s)$, $WO_z(s)$ are finite, non-empty sets of actions over the channels C_{PA}. □

This yields:

Proposition 4.27 Obligations (4-26) and (4-27) can be discharged.

Proof. We provide only the argument for (4-27) (the other being nearly the same).

Together, the assertions of Proposition 4.25 and the second assertion of Assumption 4.26 imply the assertion in (4-27) (with $WO(\,)$ defined as $WO_x(\,)$, for $x = (tr{\downarrow}snd)[1{+}\#(tr{\downarrow}rcv)]$). Thus, by the consequence rule, (4-27) follows from the noted proposition and assumption. □

The Parallel Composition Rule Inference

We have thus come to step 6 of the checklist of Section 4.5.2. The following two lemmata will establish, for the parallel composition PSP, the assertions whose satisfaction was assumed in the previous step, thus discharging Assumption 4.26. Note that in fact, compared to the assertions appearing in that assumption, the assertions INL and $OUTL$ respectively employed in the two lemmata have a slightly different form, in which $WI(\,)$ and $WO_z(\,)$ do not occur; however, INL and $OUTL$ also provide enough information to allow $WI(\,)$ and $WO_z(\,)$ to be defined, if desired (they were referred to, and constrained, but not defined, in Assumption 4.26).

Lemma 4.28 PSP **sat** INL, where:

$$
\begin{aligned}
INL \equiv\ & \textbf{let } m := \#(tr{\downarrow}snd),\ h := \max(tr{\downarrow}end_A),\ r := \#(tr{\downarrow}rcv) \textbf{ in} \\
& m < h{+}L \vee m = h{+}L \wedge h < r \Rightarrow \forall x\colon \\
& \exists p \in \{pdu(d,k) \mid d \in (tr{\downarrow}snd),\ 1 \leqslant k \leqslant m\}\colon \\
& \exists u \leqslant \langle src_P!p\rangle\colon \exists q \in (tr{\downarrow}src_P) \cup \{p\}\colon \\
& \exists w_A \in Preamble_A(tr \restriction C_A, r)\colon \exists w'_A \leqslant w_A\colon \exists n \in (tr{\downarrow}src_A) \cup \{r\}\colon \\
& \quad (snd!x \notin Ref \vee u\langle end_P!q\rangle w'_A\langle end_A!n, snd!x\rangle \notin Ref)
\end{aligned}
$$

Proof. We observe first that $INL(t, Y)$ and $INL(t\restriction(C_S \cup C_P \cup C_A \cup C_R), Y)$ are equivalent for all t and Y. By the **sat** rule (4-25) for parallel composition, under Proposition 4.18, the proof amounts to deriving:

(1) $INL(t, X_S : C_S \parallel X_P : C_P \parallel X_A : C_A \parallel X_R : C_R)$

from the assumptions:

(2) $X_S, X_P, X_A, X_R \in \mathsf{p}Act^+$

and $\alpha\ldots(t\restriction C_\alpha, X_\alpha)$ for each assertion $\alpha\ldots$ in Table 4.1 (α ranges over S, P, A, R).

In the following, let $m = \#(t{\downarrow}snd)$, $h = \max(t{\downarrow}end_A)$ and $r = \#(t{\downarrow}rcv)$.

Assume the premise of (1), so that either of its two disjuncts must hold: if $m<h+L$, then, for any x, $snd!x \notin X_S$ by $S.SDU.LIV(t\restriction C_S, X_S)$ and, for $\alpha = P, A, R$, $\langle\rangle \notin X_\alpha$ by (2). This implies $snd!x \notin X_S : C_S \parallel X_P : C_P \parallel X_A : C_A \parallel X_R : C_R$, i.e. the first disjunct of the conclusion of (1).

Suppose now the other disjunct in the premise of (1) holds, i.e. $m = h+L$ and $h < r$. Since $L \geqslant 1$, we know:

(3) $h < m$.

By $A.\pi.LIV(t\restriction C_A, X_A)$ (with $\pi(\_)$ being $\_ > h$), for $d = r$, we may choose $w_A \in Preamble_A(t\restriction C_A, r)$, $w'_A \leqslant w_A$ and n such that:

(4) $w'_A \in B_A^*$, $w'_A\restriction src_A \in \{\langle\rangle, \langle src_A!r\rangle\}$ (by Assumption 4.16);

(5) $n \in (t{\downarrow}src_A) \cup \{r\}$, and $n > h$ so (since $m = h+L$) $m < n+L$;

(6) $w'_A\langle end_A!n\rangle \notin X_A$.

Let now p be $pdu((t{\downarrow}snd)[h+1], h+1)$ (subscript $h+1$ is legal by (3)). Exploiting $P.t.LIV(t\restriction C_P, X_P)$, choose $u \leqslant \langle src_P!p\rangle$ and $q \in (t{\downarrow}src_P) \cup \{p\}$ such that

(7) $u\langle end_P!q\rangle \notin X_P$.

We can now establish the second disjunct in the conclusion of (1) (for all x and the selected instances of existentially quantified variables). I.e., for $\alpha = S, P, A, R$ we

show $w\restriction C_\alpha \notin X_\alpha$ where $w = u\langle end_P!q\rangle w'_A\langle end_A!n, snd!x\rangle$. For this purpose, the following argument can be arranged (the second column follows from (4), whereas the third '$\notin$' column is explained by the fourth):

$w\restriction C_S \ (= u(w'_A\restriction end_A)\langle end_A!n, snd!x\rangle)$	$\notin X_S$	$(3,5), S.1.LIV(t\restriction C_S, X_S)$
$w\restriction C_P \ (= u\langle end_P!q\rangle)$	$\notin X_P$	(7)
$w\restriction C_A \ (= w'_A\langle end_A!n\rangle)$	$\notin X_A$	(6)
$w\restriction C_R \ (\in \{\langle end_P!q\rangle, \langle end_P!q, src_A!r\rangle\})$	$\notin X_R$	$R.PDU^*.LIV(t\restriction C_R, X_R)$ or $R.1.LIV(t\restriction C_R, X_R)$

This concludes the proof. □

Lemma 4.29 *PSP* **sat** *OUTL*, where:

$$
\begin{aligned}
OUTL \equiv\ & \textbf{let } m := \#(tr{\downarrow}snd),\ h := \max(tr{\downarrow}end_A),\ r := \#(tr{\downarrow}rcv) \textbf{ in}\\
& h \leqslant r < m \Rightarrow \textbf{let } x := (tr{\downarrow}snd)[r{+}1] \textbf{ in}\\
& \exists z : (z = x \vee pdu(z, r{+}1) \in (tr{\downarrow}src_P) \cup (tr{\downarrow}end_P)) \wedge\\
& \exists w_P \in Preamble_P(tr\restriction C_P, pdu(x, r{+}1)) : \exists w'_P \leqslant w_P :\\
& \exists w_A \in Preamble_A(tr\restriction C_A, r) : \exists w'_A \leqslant w_A :\\
& \textbf{let } p := pdu((tr{\downarrow}snd)[h{+}1], h{+}1) \textbf{ in}\\
& \exists u \leqslant \langle src_P!p\rangle : \exists q \in (tr{\downarrow}src_P) \cup \{p\} :\\
& \exists v_P \in Preamble_P((tr\restriction C_P)u\langle end_P!q\rangle, pdu(x, r{+}1)) : \exists v'_P \leqslant v_P :\\
& \quad (rcv!z \notin Ref \vee\\
& \quad w'_P\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Ref \vee\\
& \quad u\langle end_P!pdu(z, r{+}1), rcv!z\rangle \vee\\
& \quad u\langle end_P!q\rangle w'_A\langle end_A!r\rangle v'_P\langle end_P!pdu(z, r{+}1), rcv!z\rangle \notin Ref)
\end{aligned}
$$

Proof. We observe first that $OUTL(t, Y)$ and $OUTL(t\restriction(C_S \cup C_P \cup C_A \cup C_R), Y)$ are equivalent for all t, Y. By the **sat** rule (4-25) for parallel composition, under Proposition 4.18, the proof will consist in deriving:

(1) $\quad OUTL(t, X_S : C_S \parallel X_P : C_P \parallel X_A : C_A \parallel X_R : C_R)$

from the assumptions:

(2) $\quad X_S, X_P, X_A, X_R \in \mathsf{p}Act^+$

and $\alpha\ldots(t\restriction C_\alpha, X_\alpha)$ for each assertion $\alpha\ldots$ in Table 4.1 (α ranges over S, P, A, R). In the following, let:

$$m = \#(t{\downarrow}snd),\ h = \max(t{\downarrow}end_A)$$
$$r = \#(t{\downarrow}rcv),\ x = (t{\downarrow}snd)[r{+}1] \text{ (provided } r < m)$$

and recall (4-24), whereby:

$$\mathsf{A}^{r+1}_{end_P} = \{end_P!q \mid seqno(q) = r{+}1\}.$$

To prove (1), we assume its premise:

(3) $\quad h \leqslant r < m,$

The proof continues as a case analysis. In each of the four exhaustive and mutually exclusive cases A, B.1, B.2.1 and B.2.2 orderly treated below, the corresponding disjunct in the conclusion of (1) will be shown to hold (for appropriate instances of existentially quantified variables), by exhibiting a suitable trace w such that $w\restriction C_\alpha \notin X_\alpha$ for $\alpha = S, P, A, R$.

Case A: $t\restriction\mathsf{A}^{r+1}_{end_P} \neq \langle\rangle$. Let $z = sdu(head(t\restriction\mathsf{A}^{r+1}_{end_P}{\downarrow}end_P))$ (so $pdu(z, r{+}1) \in t{\downarrow}end_P$) and $w = \langle rcv!z\rangle$. Then $w\restriction C_R \notin X_R$ by $R.SDU.LIV(t\restriction C_R, X_R)$ and, for $\alpha = S, P, A$, $w\restriction C_\alpha = \langle\rangle \notin X_\alpha$ by (2).

Case B: $t\restriction\mathsf{A}^{r+1}_{end_P} = \langle\rangle$. This case is now further split into the alternatives B.1 and B.2 (recall $h \leqslant r$ by (3)).

Case B.1: $h = r$. By $P.\pi.LIV(t\restriction C_P, X_P)$ (with $\pi(\_)$ being $seqno(\_) = r{+}1$), for $d = pdu(x, r{+}1)$, we may choose $w_P \in Preamble_P(t\restriction C_P, pdu(x, r{+}1))$, $w'_P \leqslant w_P$ and z such that:

(4) $\quad w'_P \in B^*_P,\ w'_P\restriction src_P \in \{\langle\rangle, \langle src_P!pdu(x, r{+}1)\rangle\}$ (by Assumption 4.16);

(5) $\quad w'_P\restriction\mathsf{A}^{r+1}_{end_P} = \langle\rangle$, and $pdu(z, r{+}1) \in (t{\downarrow}src_P) \cup \{pdu(x, r{+}1)\}$;

(6) $\quad w'_P\langle end_P!pdu(z, r{+}1)\rangle \notin X_P.$

Let then $w = w'_P\langle end_P!pdu(z, r{+}1), rcv!z\rangle$. By (4), $w\restriction C_S$ can be equal to either $\langle\rangle$ or $src_P!pdu(x, r{+}1)$; the former trace cannot be in X_S by (2), nor can the latter

by $S.PDU.LIV(t\restriction C_S, X_S)$ (use (3)). Also, $w\restriction C_P \notin X_P$ follows from (4), (6), and $(w\restriction C_A)=\langle\rangle \notin X_A$ from (2). Finally, from $R.2.LIV(t\restriction C_R, X_R)$, using (4), (5), it follows:

$$w\restriction C_R = (w'_P\restriction end_P)\langle end_P!pdu(z, r+1), rcv!z\rangle \notin X_R$$

Case B.2: $h < r$. By $A.\pi.LIV(t\restriction C_A, X_A)$ (with $\pi(\_)$ being $\_\geqslant r$), for $d = r$, we may choose $w_A \in Preamble_A(t\restriction C_A, r)$, $w'_A \leqslant w_A$ and k such that:

(7) $w'_A \in B_A^*$, $w'_A\restriction src_A \in \{\langle\rangle, \langle src_A!r\rangle\}$ (by Assumption 4.16);

(8) $k \in (t\downarrow src_A)\cup\{r\}$, $w'_A\downarrow end_A \subseteq \{n \mid n < r\}$, and $k \geqslant r$ (in fact $k = r$ owing to $R.ACKLIM(t\restriction C_A, X_A)$);

(9) $w'_A\langle end_A!r\rangle \notin X_A$ (use $k = r$).

We shall now apply $P.t.\pi.LIV(t\restriction C_P, X_P)$, with the quantified variable p instantiated thus:

(10) $p = pdu((t\downarrow snd)[h+1], h+1)$ (note that subscript $h+1$ is legal because, by (3), $h < m=\#(t\downarrow snd)$).

Thanks to $P.t.\pi.LIV(t\restriction C_P, X_P)$ we can choose u, q such that:

(11) $u \leqslant \langle src_P!p\rangle$, $q \in (t\downarrow src_P)\cup\{p\}$, $u\langle end_P!q\rangle \notin X_P$.

Depending on the sequence number of q, two further alternatives will be considered.

Case B.2.1: $seqno(q) = r+1$. Letting $z = sdu(q)$ implies $pdu(z, r+1)=q \in (t\downarrow src_P)$ by (11) (q cannot be p since $h < r$ in the present case B.2).

We now let $w = u\langle end_P!pdu(z, r+1), rcv!z\rangle$. Then $w\restriction C_S$ is either $\langle\rangle$ or $src_P!p$; neither trace can be in X_S: the former by (2), and the latter by $S.PDU.LIV(t\restriction C_S, X_S)$ (using (3)). Moreover, the two relations $w\restriction C_P \notin X_P$ and $w\restriction C_A = \langle\rangle \notin X_A$ follow from (11) and (2) respectively. Finally, from $R.2.LIV(t\restriction C_R, X_R)$, using (11) and B, it follows: $w\restriction C_R = \langle end_P!pdu(z, r+1), rcv!z\rangle \notin X_R$.

Case B.2.2: $seqno(q) \neq r+1$. Using again $P.t.\pi.LIV(t\restriction C_P, X_P)$, with $\pi(\_)$ being $seqno(\_) = r+1$ and d instantiated as $pdu(x, r+1)$, we supplement (11) by choosing $v_P \in Preamble_P((t\restriction C_P)u\langle end_P!q\rangle, pdu(x, r+1))$, $v'_P \leqslant v_P$ and z such that:

(12) $v'_P \in B^*_P$, $v'_P \restriction src_P \in \{\langle\rangle, \langle src_P!pdu(x, r+1)\rangle\}$ (by Assumption 4.16);

(13) $v'_P \restriction \mathsf{A}^{r+1}_{end_P} = \langle\rangle$, and $pdu(z, r+1) \in (t \downarrow src_P) \cup \{pdu(x, r+1)\}$;

(14) $u\langle end_P!q\rangle v'_P\langle end_P!pdu(z, r+1)\rangle \notin X_P$.

Note that according to $P.t.\pi.LIV(t \restriction C_P, X_P)$, (13) should also permit $pdu(z, r+1)$ to be in $u \downarrow src_P$, which is not allowed for by the constraints posed by (1) on z. However, by (11), the set $u \downarrow src_P$ is either $\varnothing$, or $\{p\}$, and $seqno(p) = h+1$ by (10); so $pdu(z, r+1) \in u \downarrow src_P$ would imply $h = r$, which contradicts the assumption $h < r$ characterizing case B.2.

We can now let w be $u\langle end_P!q\rangle w'_A\langle end_A!r\rangle v'_P\langle end_P!pdu(z, r+1), rcv!z\rangle$, and prove $w \restriction C_\alpha \notin X_\alpha$ for $\alpha = S, P, A, R$. Note that, in order to evaluate projection on each C_α, (7), (11) and (12) are exploited.

Recalling $u \leqslant \langle src_P!p\rangle$ (11), $h \leqslant r < m$ (3), $\max(w'_A \downarrow end_A) < r$ (8), we can use $S.1.LIV(t \restriction C_S, X_S)$ to establish the $\notin$ relation in:

$$w \restriction C_S \in u(w'_A \restriction end_A)\langle end_A!r\rangle\{\langle\rangle, \langle src_P!pdu(x, r+1)\rangle\}, \text{ and } w \restriction C_S \notin X_S.$$

Conditions (14) and (9) are $w \restriction C_\alpha \notin X_\alpha$ for $\alpha = P, A$.

Finally, $R.2.LIV(t \restriction C_R, X_R)$, using B and (for $(\langle end_P!q\rangle(v'_P \restriction end_P)) \restriction \mathsf{A}^{r+1}_{end_P} = \langle\rangle$) B.2.2 and (13), ensures the $\notin$ relation in:

$$w \restriction C_R \in \langle end_P!q\rangle\{\langle\rangle, src_A!r\}(v'_P \restriction end_P)\langle end_P!pdu(z, r+1), rcv!z\rangle, \text{ and}$$
$$w \restriction C_R \notin X_R \qquad \square$$

4.5.7 Component Implementation and Verification

For the case study chosen, we have thus far accomplished all the steps identified in the design checklist of Section 4.5.2, except the last. This one will consist in defining the process expressions *SENDER* and *RECEIVER*, and then proving that they satisfy the relevant basic properties introduced in Section 4.5.3. We shall thereby discharge Assumption 4.17, the last still active, in so far as it is concerned with *SENDER* and *RECEIVER* (media will not be defined, so their properties ultimately need to be assumed instead).

Implementation

SENDER will be defined as an instance of a process constant $SENDER(m, h, uq)$, controlled by parameters m, the number of sdus accepted, h the highest ack received, and uq the sequence of sdus accepted and yet unacknowledged (more precisely, $\#uq$ must be $m-h$ and, for $1 \leqslant i \leqslant m-h$, $uq[i]$ is the $(h+i)$th sdu accepted). $SENDER(m, h, uq)$ may do three things: input an sdu at *snd*, provided $m < h+L$, receive an ack at end_A, or (re)transmit a pdu numbered higher than h at src_P:

$$
\begin{aligned}
&SENDER(m, h, uq) := \\
&\quad m < h+L \rightarrow snd?x;\ SENDER(m+1, h, uq \cdot x) \\
&\oplus\ end_A?n;\ SENDER(m, \max(n, h), uq[1+\max(n, h)-h..]) \\
&\oplus\ h < m \rightarrow src_P!pdu(uq[1], h+1);\ SENDER(m, h, uq)
\end{aligned}
$$

Initially, uq must be $\langle\rangle$, and m and h must be 0, so we let:

$$SENDER \equiv SENDER(0, 0, \langle\rangle)$$

As an example of the unavoidable arbitrary choices inherent in implementation, observe that $SENDER(m, h, uq)$ could just as well be defined to be ready to output, together with $pdu(uq[1], h+1)$, any pdu carrying one of the sdus in uq. Such a behaviour would still be consistent with *S.PDU.LIV* and therefore with the preceding verification, which is based solely (in this respect) on the assumption that *S.PDU.LIV* is satisfied by *SENDER*.

RECEIVER will also be defined as an instance of a parametric process constant, namely $RECEIVER(r, d)$, controlled by an integer r (counting sdus already output at *rcv*) and a sdu d (the next to be output, if any, otherwise the "non-sdu" *NIL*). Thus, if d is not *NIL*, it may be delivered at *rcv*, the count r is incremented and d is reset to *NIL*. When a pdu q arrives, its sdu will be selected for output iff none is already available and q carries the expected sequence number $r+1$. Finally, the receiver is always ready to issue ack r at src_A. Again, this implementation choice is somewhat arbitrary, in that it conveys more detail than specification *R.ACK.LIV*; e.g. alternatively, *R.ACK.LIV* would also let *RECEIVER* output an ack only in reply to an incoming pdu.

Formally, we define:

$$
\begin{aligned}
&RECEIVER(r, d) := \\
&\quad d \neq NIL \rightarrow rcv!d; RECEIVER(r{+}1, NIL) \\
&\oplus\ end_P?q;\ \textbf{let}\ e := \text{if}\ (seqno(q) = r{+}1 \wedge d = NIL)\ \text{then}\ sdu(q)\ \text{else}\ d\ \textbf{in} \\
&\qquad\qquad RECEIVER(r, e) \\
&\oplus\ src_A!r; RECEIVER(r, d)
\end{aligned}
$$

and:

$$RECEIVER \equiv RECEIVER(0, NIL)$$

Fixpoint Induction Verification

In accordance with the design checklist of Section 4.5.2, the only obligation left as regards deadlock freedom verification are the proofs of the basic properties attributed to components *SENDER* and *RECEIVER* with Assumption 4.17.

Since these components are recursively defined, the proofs ought to employ fixpoint induction. While the application of this powerful rule is apt to bring about some complexity, this can be kept to a minimum in the present instance thanks to the design choice discussed in Section 4.5.4 and formalized with Assumption 4.17. Thereby, our assumptions about *SENDER* and *RECEIVER* were limited to properties of their traces and one-action refusals; as a result, the induction proofs turn out to be essentially the same that would be given in the standard CSP **sat** logic. For this reason, and for brevity's sake, we will just report the proof of:

$$RECEIVER\ \textbf{sat}\ R.SDU.LIV \tag{4-28}$$

For this purpose it is convenient to introduce an alternative but obviously equivalent definition of *RECEIVER*. Our aim is to make the application of the fixpoint induction rule more straightforward. For n integer and any data d, we define (recalling *NIL* is not a sdu):

$$
\begin{aligned}
A_{n,d} &\equiv \{rcv!d \mid d \neq NIL\} \cup \{end_P!q \mid sdu(q) \neq NIL\} \cup \{src_A!n\} \\
REC_{n,d} &\equiv A_{n,d}; \langle REC_{f(n,d,a),g(n,d,a)} \mid a \in A_{n,d} \rangle
\end{aligned}
$$

$$f(n,d,a) = \begin{cases} n{+}1 & \text{if } d \neq NIL,\ a = rcv!d \\ n & \text{otherwise} \end{cases}$$

$$g(n,d,a) = \begin{cases} NIL & \text{if } d \neq NIL,\ a = rcv!d \\ e & \text{if } d = NIL,\ a = end_P!pdu(n{+}1,e) \\ d & \text{otherwise} \end{cases}$$

and the formula

$$\begin{aligned} T_{n,d} \equiv\ & \textbf{let } k := \#(tr{\downarrow}rcv) \textbf{ in let } u := tr{\restriction}\mathsf{A}^{n+k+1}_{end_P} \textbf{ in} \\ & \textbf{if } k = 0 \wedge d \neq NIL \textbf{ then } rcv!d \notin Ref \\ & \textbf{else } u \neq \langle\rangle \Rightarrow rcv!sdu(hd(u{\downarrow}end_P)) \notin Ref \end{aligned}$$

Clearly, for $n = 0$ and $d = NIL$, $REC_{n,d}$ is *RECEIVER* and $T_{n,d}$ is *R.SDU.LIV*. Therefore, the desired result (4-28) is just a corollary of:

Theorem 4.30 For all integer n and data d: $REC_{n,d}$ **sat** $T_{n,d}$

Proof. An application of the fixpoint induction rule. The basis is:

$$A_{n,d};\ \langle STOP \mid a \in A_{n,d}\rangle \textbf{ sat } tr = \langle\rangle \Rightarrow d \neq NIL \Rightarrow rcv!d \notin Ref$$

It will be derived from the action sequence rule, which is reproduced here for convenience:

$$\frac{\begin{array}{c} p_a \textbf{ sat } S_a, \text{ for } a \in A \\ (X \in \mathsf{p}Act^+ \wedge X \cap A = \varnothing) \Rightarrow (\forall a \in A\colon S_a(\langle\rangle, X \div a)) \Rightarrow S(\langle\rangle, X), \\ X \in \mathsf{p}Act^+ \Rightarrow S_a(s,X) \Rightarrow S(as,X), \text{ for } a \in A \end{array}}{A;\ p_A \textbf{ sat } S}$$

Taking $S_a \equiv TRUE$, the premises are all easy to establish; we just consider the second, which follows from $X \cap A_{n,d} = \varnothing \Rightarrow d \neq NIL \Rightarrow rcv!d \notin X$, whose validity is an obvious consequence of the way $A_{n,d}$ is defined.

In order to carry out the induction step, we assume the induction hypothesis:

$$REC_{m,v} \textbf{ sat } T_{m,v} \quad \text{for } m \geqslant 0, \text{ any data } v$$

and, for given $n \geqslant 0$ and data d, we try to show:

$$A_{n,d};\ \langle REC_{f(n,d,a),g(n,d,a)} \mid a \in A_{n,d}\rangle \textbf{ sat } T_{n,d}$$

This is again a consequence of the action sequence rule, with S_a instantiated as $T_{f(n,d,a),g(n,d,a)}$. The first premise of the rule follows from the induction hypothesis. The second premise follows from:

$$X \cap A_{n,d} = \varnothing \Rightarrow d \neq NIL \Rightarrow rcv!d \notin X,$$

using the definition of $A_{n,d}$. Finally, the third premise can be established by showing the validity of:

$$X \in \mathsf{p}Act^{+} \Rightarrow T_{f(n,d,a),g(n,d,a)}(s, X) \Rightarrow T_{n,d}(as, X) \qquad \text{for } a \in A_{n,d} \qquad (4\text{-}29)$$

The proof continues by a case analysis.

Case 1. If $a = rcv!d$ and $d \neq NIL$, (4-29) becomes: $X \in \mathsf{p}Act^{+} \Rightarrow T_{n+1,NIL}(s, X) \Rightarrow T_{n,d}(as, X)$. This can be seen to hold using the following equivalences:

$$\begin{aligned} T_{n+1,NIL}(s, X) \;\equiv\; & \textbf{let } k := \#(s{\downarrow}rcv) \textbf{ in let } u := s{\restriction}\mathsf{A}^{n+1+k+1}_{end_P} \textbf{ in} \\ & u \neq \langle\rangle \Rightarrow rcv!sdu(hd(u{\downarrow}end_P)) \notin X \\ T_{n,d}(as, X) \;\equiv\; & \textbf{let } h := 1 + \#(s{\downarrow}rcv) \textbf{ in let } u := s{\restriction}\mathsf{A}^{n+h+1}_{end_P} \textbf{ in} \\ & u \neq \langle\rangle \Rightarrow rcv!sdu(hd(u{\downarrow}end_P)) \notin X \end{aligned}$$

Case 2. If $a = \langle end_P!pdu(n{+}1, e)\rangle$, $e \neq NIL$ and $d = NIL$, (4-29) becomes: $X \in \mathsf{p}Act^{+} \Rightarrow T_{n,e}(s, X) \Rightarrow T_{n,d}(as, X)$. To see this holds, note that for $\#(s{\downarrow}rcv) > 0$ both $T_{n,e}(s, X)$ and $T_{n,d}(as, X)$ are equivalent to:

$$\begin{aligned} & \textbf{let } k := \#(s{\downarrow}rcv) \textbf{ in let } u := s{\restriction}\mathsf{A}^{n+k+1}_{end_P} \textbf{ in} \\ & u \neq \langle\rangle \Rightarrow rcv!sdu(hd(u{\downarrow}end_P)) \notin X \end{aligned}$$

whereas for $\#(s{\downarrow}rcv) = 0$ they both become $rcv!e \notin X$

Case 3. In any other case, the goal (4-29) becomes: $X \in \mathsf{p}Act^{+} \Rightarrow T_{n,d}(s, X) \Rightarrow T_{n,d}(as, X)$. This is easy to show if $a = src_A!n$. If instead $a = \langle end_P!pdu(n{+}1, e)\rangle$ and $d{\neq}NIL$, $T_{n,d}(s, X)$ and $T_{n,d}(as, X)$ can be seen to coincide in the two subcases $\#(s{\downarrow}rcv) > 0$ and $\#(s{\downarrow}rcv) = 0$. □

Chapter 5

Conclusions

5.1 Scope and Results

The standard CSP formalism enjoys several important merits: an intuitively appealing model, based on a natural and adequate notion of observation, namely that of failure; a rich set of operators; an elegant fixpoint semantics; and a useful logic and calculus for reasoning about failures of processes. These techniques have been successfully employed for the analysis of many interesting systems. However, their overall practical applicability is diminished by two weaknesses of standard failure semantics.

The first is the inability to describe systems whose components might choose, at some stage, to exchange any data out of infinitely many. This must often be assumed for design and verification purposes (e.g. for the many protocols relying upon sequence numbers to cope with out-of-sequence received messages). The problem lies in the definition of the hiding operator in standard failure semantics. This work has put forward a solution based on an interesting technical result about infinite sets of sequences.

Another difficulty with standard failure semantics is its treatment of divergence, the phenomenon in which some parts of a system interact by performing an infinite, uninterrupted sequence of externally invisible actions. Within failure semantics, divergence cannot be abstracted from on the basis of the implicit fairness assumption that, if there is a choice leading out of divergence, it will eventually

be made. This 'fair abstraction' is essential for the verification of many important systems, including communication protocols.

The solution proposed in this thesis is an extended failure semantics which records refused traces, rather than just actions. Not only is this approach compatible with fair abstraction, but it also permits, like ordinary failure semantics, verification in a compositional calculus with fixpoint induction. Rather interestingly, these results can be obtained outside traditional fixpoint theory, which cannot be applied in this case. The theory developed is based on the novel notion of 'trace-based' process functions. These can be shown to possess a particular fixpoint that, unlike the least fixpoint of traditional treatments, is compatible with fair abstraction. Moreover, they form a large class, sufficient to give a compositional denotational semantics to a useful CSP-like process language.

Finally, a logic is proposed in which the properties of a process' extended failures can be expressed and analyzed; the methods developed are applied to the verification of two example communication protocols: a toy one and a large case study inspired by a real transport protocol.

The main results of the thesis have been summarized in the outlines opening Chapters 2, 3 and 4.

5.2 Related Work and Further Studies

Our extended failures semantics, and the denotational-style solution it provides to the problem of fair abstraction, would appear to be thoroughly original. Yet, an interesting analogy can be drawn between our approach and some recent work on CSP [Barrett, 1991; Mislove, Roscoe, & Schneider, 1994]: both approaches need to deal with fixpoints of functions over process domains that are not cpos. The analogy, however, stops here. Indeed the respective objectives are different: ours is to reconcile fair abstraction with failures denotational semantics, that of the cited works is to treat unbounded nondeterminism. Also the mathematical techniques employed differ: our process domain is not a cpo, in that it lacks a bottom, but does enjoy a strong completeness property (every process set has a least upper bound); the process domain of [Barrett, 1991] is not complete and some language

operations are not continuous (the remedy is to justify the denotational semantics operationally); the domain of [Mislove, Roscoe, & Schneider, 1994], instead, has a weak completeness property (every set with an upper bound has a least upper bound). In our theory, unbounded nondeterminism does not represent a problem, provided the choices it poses are resolved either all internally or all externally; it remains to be seen whether treating the general case requires an integration with the techniques cited. A related issue for further study is the extension of our work to the case of infinite refusal sets.

Another topic that warrants further investigation is the completeness of the **sat** calculus with and without the process-oriented consistency rules.

Bibliography

Abramsky, S. [1987]. Observation equivalence as a testing equivalence. *Theoretical Computer Science, 53*, 225–241.

Apt, K., Francez, N., & Katz, S. [1988]. Appraising fairness in languages for distributed programming. *Distributed Computing, 2*, 226–241.

Baeten & Bergstra [1991]. Real time process algebra. *Formal Aspects of Computing, 3.*

Baeten, J. C. M. & Weijland, W. P. [1990]. *Process Algebra.* Cambridge University Press.

Baeten, J. C. M., Bergstra, J. A., & Klop, J. W. [1987]. Ready trace semantics for concrete process algebra with the priority operator. *British Comp. Journal, 30*, 498–506.

Baird, B., Istrail, S., & Meyer, A. R. [1988]. Bisimulation can't be traced: Preliminary report. In *Proc. of the 15th ACM Symp. on Principles of Programming Languages*, pp. 229–239.

Barrett, G. [1991]. The fixed point theory of unbounded non-determinism. *Formal Aspects of Computing, 3*, 110–128.

Barringer, H., Kuiper, R., & Pnueli, A. [1986]. A really abstract concurrent model and its temporal logic. In *Proc. of the 13th Annual ACM Symposium on Principles of Programming Languages*, pp. 173–183.

Bartlett, K. A., Scantlebury, R. A., & Wilkinson, P. T. [1969]. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM, 12*, 260–261.

Bergstra, J. A. & Klop, J. W. [1984]. Process algebra for synchronous communication. *Information and Control, 60*, 109–137.

Bergstra, J. A., Klop, J. W., & Olderog, E.-R. [1987]. Failures without chaos: a new failures semantics for fair abstraction. In Wirsing, M. (Ed.), *Proc. of the IFIP Working Conference on Formal Description of Programming Concepts*, pp. 77–101. North-Holland.

Bergstra, J. A., Klop, J. W., & Olderog, E.-R. [1988]. Readies and failures in the algebra of communicating processes. *SIAM Journal on Computing, 17*(6), 1134–1177.

Best, E. [1985]. Concurrent behaviors: Sequences processes and axioms. In Brookes, S. D. et al. (Eds.), *Seminar on Concurrency*, Lecture Notes in Computer Science, vol. 197, pp. 221–245, Carnegie-Mellon University. Springer-Verlag.

Best, E. [1990]. Partial order semantics of concurrent programs. In Baeten, J. C. M. & Klop, J. W. (Eds.), *Concur '90*, Lecture Notes in Computer Science, vol. 458, p. 1. Springer-Verlag.

Brookes, S. D. & Roscoe, A. W. [1985]. An improved failures model for communicating processes. In Brookes, S. D. et al. (Eds.), *Seminar on Concurrency*, Lecture Notes in Computer Science, vol. 197, pp. 281–305, Carnegie-Mellon University. Springer-Verlag.

Brookes, S. D., Hoare, C. A. R., & Roscoe, A. W. [1984]. A theory of communicating sequential processes. *Journal of the ACM, 31*(3), 560–599.

Brookes, S. D. [1983a]. *A Model for Communicating Sequential Processes.* D. Phil. dissertation, Oxford University, Oxford, England.

Brookes, S. D. [1983b]. On the relationship of CCS and CSP. In *Proc. of the 1983 Conference on Automata, Languages and Programming (ICALP 83)*, Lecture Notes in Computer Science, vol. 154. Springer-Verlag.

Burstall, R. M. & Darlington, J. [1977]. A transformation system for developing recursive programs. *Journal of the ACM, 24*, 44–67.

Carchiolo, V., Di Stefano, A., Faro, A., Pappalardo, G., & Scollo, G. [1986]. A LOTOS specification of the PROWAY highway service. *IEEE Transactions on Computers, 35.*

Carchiolo, V., Di Stefano, A., Faro, A., & Pappalardo, G. [1989]. ECCS and LIPS: Two languages for OSI systems specification and verification. *ACM Transactions on Programming Languages and Systems, 11.*

Carchiolo, V., Faro, A., & Pappalardo, G. [1992]. Axiomatic verification of a communication protocol in the single-language and two-language frameworks. Tech. Rep., Istituto di Informatica e Telecomunicazioni, Facoltà di Ingegneria, Università di Catania, Italy.

Christoff, I. [1990]. Testing equivalences and fully abstract models for probabilistic processes. In Baeten, J. C. M. & Klop, J. W. (Eds.), *Concur '90*, Lecture Notes in Computer Science, vol. 458, pp. 126–140. Springer-Verlag.

Clarke, E., Emerson, E., & Sistla, A. [1986]. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems, 8*(2), 244–263.

Davies, J. & Schneider, S. [1993]. Recursion induction for real-time processes. *Formal Aspects of Computing, 3.*

De Nicola, R. & Hennessy, M. [1984]. Testing equivalences for processes. *Theoretical Computer Science, 34*, 83–133.

De Nicola, R. [1987]. Extensional equivalences for transition systems. *Acta Informatica, 24*, 211–237.

Dijkstra, E. W. [1965]. Solution of a problem in concurrent programming control. *Communications of the ACM*, *8*(9), 569.

Emerson, E. A. & Halpern, J. Y. [1986]. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, *33*(1), 151–178.

Gabbay, D., Pnueli, A., Shelah, S., & Stavi, J. [1980]. On the temporal analysis of fairness. In *Proc. of the 7th Annual ACM Symposium on Principles of Programming Languages*, pp. 163–173.

Gerth, R. & Boucher, A. [1987]. A timed failures model for extended communicating processes. In *Proc. ICALP 87*, Lecture Notes in Computer Science, pp. 95–114. Springer-Verlag.

Goguen, J. A. & Burstall, R. M. [1984]. Introducing institutions. In *Proc. of the Logics of Programming Workshop*, Lecture Notes in Computer Science, vol. 164, pp. 221–256, Carnegie-Mellon University. Springer-Verlag.

Groote, J. F. [1990]. Specification and verification of real time systems in ACP. Tech. Rep., Centre for Mathematics and Computer Science, Amsterdam.

Hall, A. [1990]. Seven myths on formal methods. *IEEE Software*, September 1990.

Hennessy, M. & Milner, R. [1985]. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, *32*(1), 137–161.

Hennessy, M. [1991]. A proof system for communicating processes with value-passing. *Formal Aspects of Computing*, *3*(4), 326–345.

Hoare, C. A. R. [1985]. *Communicating Sequential Processes.* Prentice-Hall.

ISO [1988]. *Connection-Oriented Transport Protocol Specification.* International Standard IS 8073, International Organization for Standardization.

ISO [1989]. *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* Draft International Standard IS 8807, International Organization for Standardization.

Jou, C.-C. & Smolka, S. A. [1990]. Equivalences, congruences and complete axiomatizations for probabilistic processes. In Baeten, J. C. M. & Klop, J. W. (Eds.), *Concur '90*, Lecture Notes in Computer Science, vol. 458, pp. 367–383. Springer-Verlag.

Koomen, C. [1985]. Algebraic specification and verification of communication protocols. *Science of Computer Programming*, *5*, 1–36.

Koutny, M., Mancini, L., & Pappalardo, G. [1991]. Formalising replicated distributed processing. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pp. 108–117, Pisa. IEEE.

Koutny, M., Mancini, L., & Pappalardo, G. [1993]. Modelling replicated processing. In Bode, A., Reeve, M., & Wolf, G. (Eds.), *Proc. of the PARLE 93 Parallel Architectures in Europe Conference*, Lecture Notes in Computer Science, vol. 694. Springer-Verlag.

Lamport, L. [1980]. "Sometime" is sometimes "not never"—On the temporal logic of programs. In *Proc. of the 7th ACM Symp. on Principles of Programming Languages*, pp. 174–185.

Lamport, L. [1986]. On interprocess communication: Part I—Basic formalism. *Distributed Computing*, *1*, 77–85.

Larsen, K. G. & Skou, A. [1989]. Bisimulation through probabilistic testing. In *Proc. 16th ACM Symp. on Princ. of Prog. Lang.*, pp. 344–352.

Larsen, K. [1987]. A context dependent bisimulation between processes. *Theoretical Computer Science*, *49*.

Larsen, K. [1990]. Ideal specification formalism = ... In Baeten, J. C. M. & Klop, J. W. (Eds.), *Concur '90*, Lecture Notes in Computer Science, vol. 458, pp. 33–56. Springer-Verlag.

Lehmann, D., Pnueli, A., & Stavi, J. [1981]. Impartiality, justice, fairness: The ethics of concurrent termination. In *Proc. of the 1981 Conference on Automata,*

Languages and Programming (ICALP 81), Lecture Notes in Computer Science, vol. 115. Springer-Verlag.

Loeckx, J. & Sieber, K. [1987]. *The Foundations of Program Verification.* Wiley-Teubner.

Mancini, L. V. & Pappalardo, G. [1988]. Towards a theory of replicated processing. In Joseph, M. (Ed.), *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, vol. 331, pp. 175–192, Warwick. Springer-Verlag.

Meyer, B. [1985]. On formalism in specification. *IEEE Software*, January 1985, 6–26.

Milner, R. [1983]. Calculi for synchrony and asynchrony. *Theoretical Computer Science, 25*, 267–310.

Milner, R. [1989]. *Communication and Concurrency.* Prentice-Hall.

Mislove, M. W., Roscoe, A. W., & Schneider, S. A. [1994]. Fixed points without completeness.

Moller, F. & Tofts, C. [1990]. A temporal calculus of communicating systems. In Baeten, J. C. M. & Klop, J. W. (Eds.), *Concur '90*, Lecture Notes in Computer Science, vol. 458, pp. 401–415. Springer-Verlag.

Olderog, E. & Hoare, C. [1986]. Specification-oriented semantics for communicating processes. *Acta Informatica, 23*, 9–66.

Pappalardo, G. [1987]. Experiences with a verification and simulation tool for behavioural languages. In *Proc. of the VII IFIP International Workshop on Protocol Specification Testing and Verification*, Zurich. IFIP, North-Holland.

Parrow, J. & Gustavson, R. [1984]. Modelling distributed systems in an extension of CCS with infinite experiments and temporal logic. In *Proc. of the 4th IFIP Conference on Protocol, Specification, Testing and Verification.* North-Holland.

Plotkin, G. [1982]. An operational semantics for CSP. Internal Report CSR-114-82, University of Edinburgh.

Pnueli, A. & Zuck, L. [1986]. Verification of multiprocess probabilistic protocols. *Distributed Computing, 1*, 53–72.

Pnueli, A. [1986]. Specification and development of reactive systems. In Kugler, H.-J. (Ed.), *Information Processing 86*, pp. 845–858. IFIP, North-Holland.

Reed, G. M. & Roscoe, A. W. [1988]. A timed model for communicating sequential processes. *Theoretical Computer Science, 58*, 249–261.

Reisig, W. [1985]. *Petri Nets.* Springer-Verlag.

Sannella, D. [1988]. A survey of formal software development methods. Expository Report ECS-LFCS-88-56, University of Edinburgh, LFCS, Department of Computer Science, Edinburgh.

Shankar, K. S. & Lam, S. S. [1987]. Time-dependent distributed systems: Proving safety, liveness and real-time properties. *Distributed Computing, 2*, 61–79.

Tanenbaum, A. [1988]. *Computer Networks* (Second edition). Prentice-Hall.

Tofts, C. [1990]. A synchronous calculus of relative frequency. In Baeten, J. C. M. & Klop, J. W. (Eds.), *Concur '90*, Lecture Notes in Computer Science, vol. 458, pp. 467–480. Springer-Verlag.

Tofts, C. [1994]. Processes with probabilities, priority and time. *Formal Aspects of Computing, 6.*

v. Glabbeek, R. & Weijland, W. P. [1989]. Branching time and abstraction in bisimulation semantics. In Ritter, G. X. (Ed.), *Information Processing 89*, pp. 613–618. North-Holland. Extended Abstract.

v. Glabbeek, R., Smolka, S. A., Steffen, B., & Tofts, C. [1990]. Reactive, generative and stratified models of probabilistic processes. In *Proc. 5th IEEE Symp. on Logic in Comp. Sci.*

v. Glabbeek, R. [1990]. The linear time-branching time spectrum. In Baeten, J. C. M. & Klop, J. W. (Eds.), *Concur '90*, Lecture Notes in Computer Science, vol. 458, pp. 278–297. Springer-Verlag.

v. Glabbeek, R. [1993]. The linear time-branching time spectrum, II. In Best, E. (Ed.), *Concur '93*, Lecture Notes in Computer Science, vol. 715. Springer-Verlag.

Vardi, M. Y. [1985]. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. 26th IEEE Symp. on Found. of Comp. Sci.*, pp. 327–338.

Walker, D. [1987]. Bisimulation equivalence and divergence in CCS. LFCS Report ECS-LFCS-87-29, University of Edinburgh.

Wing, J. [1990]. A specifier's introduction to formal methods. *IEEE Computer*, September 1990, 8–24.