# Concurrent and Real-time Systems: The CSP Approach

*selected chapters*

**Steve Schneider**

**Part I: The language of CSP**
**Part II: Analysing processes**

# *Concurrent and Real Time Systems*

*the CSP approach*

**Steve Schneider**

A Wiley-Interscience Publication

***JOHN WILEY & SONS, INC.***

New York / Chichester / Weinheim / Brisbane / Singapore / Toronto

# *Preface*

*By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems.*

Alfred North Whitehead

This book provides an introduction to Communicating Sequential Processes (CSP) and its use as a formal method for concurrency. The CSP approach has been widely used in the specification, analysis and verification of concurrent and real-time systems, and for understanding the particular issues that can arise when concurrency is present. It provides a good notation which enables specifications and designs to be clearly expressed and understood, together with a supporting theory which allows them to be analyzed and shown to be correct.

Concurrent systems are complicated: they consist of many components which may execute in parallel, and the complexity arises from the combinations of ways in which their parts can interact. The design of such systems requires ways of keeping these interactions under control. Concurrency by its very nature introduces phenomena not present in sequential systems, such as deadlock and livelock. Deadlock can arise when a number of components are each awaiting an interaction from some other component before they can themselves continue. Livelock arises when components descend into an endless sequence of interaction among themselves, excluding any other components and the outside world. These properties arise not from individual components but from the way they are combined. Nondeterminism can also arise naturally in parallel compositions, for example when race conditions arise. The presence of time adds another dimension to the complexity. A theory of concurrency such as CSP provides a way of understanding and thereby controlling such phenomena.

The language of CSP is appropriate for capturing system descriptions at different stages in the development process:

- **Specifications** describe the required or expected behaviour of a system or component. These may be captured within the language of CSP.
- **Design** decisions are concerned with how components might be combined to provide a system meeting a particular specification.
- **Implementation** descriptions contain only those aspects of the language of CSP that can be directly converted into program code.

These levels are not rigid. It may be difficult to tell whether a CSP description is a specification or an abstract design, or to distinguish a more detailed design from an implementation. This allows a stepwise development from specification to implementation, since all intermediate stages may be considered as designs which progressively fill in more detail. One benefit of using a single language is that different levels of description can be compared and related within a single framework.

Even individual features of the language may be used at a number of different levels. Parallel combination may be used at the level of specification to denote conjunction, at the level of design to describe a concurrent architecture, and at the level of implementation to describe how processes must synchronize. Internal choice likewise is appropriate in specifications to denote a disjunction of possibilities, at the level of design to indicate that a number of approaches may be appropriate to provide some service, and at the level of implementation when run-time nondeterminism is present. Other operators of the language are appropriate only at some stages of the development process. Event abstraction is not appropriate in specifications, since it is concerned with internalizing events—and internal events should not appear in specifications. It is used in design, when describing the structure of complex systems which have some internal detail.

The language has been evolving since its inception, even recently as application of the model-checking tool FDR[1] to real problems shows which operators are useful in practice, and motivates new operators and alterations to existing ones.

## Organization

This book is organized into four parts. Parts I and II are concerned with the untimed language and theory of CSP, and Parts III and IV are concerned with the introduction of time into the language and underlying theory. Part I introduces the core language of CSP, explaining in operational terms how CSP processes might execute. Chapter 1 discusses the central notion of processes, introduces the labelled transition system approach to operational semantics, and covers the sequential part of the language: the performance of events, input and output, and

[1]developed and marketed by Formal Systems Europe Ltd

the various forms of choice. Chapter 2 provides the ways in which processes can be combined in parallel and introduces the various forms of concurrency into the language. Chapter 3 completes the discussion of the language, introducing the various abstraction mechanisms provided by CSP, and the ways in which flow of control can be described.

Part II introduces the semantic models which provide ways of understanding the language in terms of how processes can behave, and which provide foundations for specification and verification techniques. Chapter 4 provides the simplest model, using *traces* as observations, and illustrates the denotational approach, particularly the way recursion is treated. Chapter 5 introduces the approach taken to specification of processes, and provides a compositional proof system based upon the traces model for verification of safety properties. These two chapters between them exemplify the approach taken throughout the book to providing a denotational semantics for CSP, and for specifying and verifying processes. Chapter 6 introduces a more detailed kind of observation, the *stable failure* which allows analysis of phenomena such as nondeterminism and deadlock. The stable failures model is closely related to the classical *failures-divergences* model—they are identical for divergence-free processes—and provides a cleaner introduction to the notion of failures. Chapter 7 covers the additional specifications that this model permits; and provides a proof system for their verification. Finally, Chapter 8 introduces the *failures-divergences-infinite traces* model which allows questions of liveness and arbitrary nondeterminism to be properly addressed. This model is an extension of the traditional failures-divergences model to handle unbounded nondeterminism, and relates more crisply to the timed models introduced later in the book.

In Part III, time is introduced into the CSP language. Chapter 9 presents new language constructs to describe timeouts, delays, and timed interrupts, and provides a timed operational semantics for the enhanced language which describes how processes are to be executed with respect to the explicit passage of time. Chapter 10 considers in greater depth the nature and character of the timed labelled transition systems used to provide CSP with a timed operational semantics.

Part IV provides an understanding of the language in terms of timed observations. Chapter 11 introduces timed observations in terms of *timed failures*, and presents the corresponding semantic model, together with the timed failures denotational semantics for CSP. Chapter 12 discusses the use of timed failures as a basis for specification of real-time requirements, and covers a specification macro language for expressing common timed specification idioms. It also provides a compositional proof system for verification of time-sensitive systems with respect to such specifications. Finally, Chapter 13 draws together the untimed and timed approaches to CSP through the theory of *timewise refinement*, and shows how to exploit the links between the various models in order to combine analyses at different levels of abstraction.

Notes on work related to CSP and to timed CSP appear at the end of Parts I and III respectively, and notes on the development of the theory appear at the end of Parts II and IV. Exercises on the material appear at the end of each chapter.

The book has an associated web site

`http://www.cs.rhbnc.ac.uk/books/concurrency`

on which answers to many of the exercises can be found (some with restricted access), as well as a variety of other course material related to this book.

## Course suggestions

This book is intended primarily as a textbook, aimed at final year undergraduates and postgraduates. As such, it can support a variety of courses on CSP.

**Traditional CSP**: The first two parts of the book are self-contained, and provide an introduction to the language and theory of untimed CSP. These two parts (leaving out the difficult material of Chapter 8) can form the basis of a one semester postgraduate or advanced undergraduate course on concurrency. The tool support provided by ProBe and FDR will enhance any such course significantly. If used, emphasis should be placed on process-oriented specification at the expense of property-oriented specification, and the main models to cover will be the traces model and the stable failures model. There may even be some time at the end of the course to cover an introduction to timed CSP.

**Concurrent and real time systems**: A less formal course covering issues in concurrency can instead concentrate on the language of timed and untimed CSP and ignore the semantic models. Parts I and III of the book between them introduce and explain the full CSP language, and are self-contained. This course would still benefit from introducing traces as a basis for verification (and traces refinement in FDR), and a discussion of deadlock and divergence, though failures and divergences semantics would most likely be beyond its scope.

**Real time concurrency**: The third and fourth parts of the book comprise a one-semester course on real time CSP, or the basis for a course on design of real-time systems. Parts III and IV rest to some extent on previous exposure to CSP, and ideally this course would follow a course based on the first half of the book. However, the required CSP can be obtained as the course progresses by dipping into the earlier parts as and when necessary, at the cost of a slower pace. These two parts are self-contained and mostly independent of the first half (apart from Chapter 13, which covers the relationship between untimed and timed CSP).

## Semantic approaches

In this book the CSP language is introduced operationally: CSP programs are defined in terms of how they are to be executed. This is for purely the purposes of explanation—experience has shown the author that CSP operators are easier to understand initially when explained through operational semantics. However, the CSP approach is denotational in nature, the design of the language is driven by denotational considerations, and reasoning and analysis should be carried out at the level of the appropriate denotational model. The operational presentation is essentially for elucidation of the language. Different semantic approaches have relative strengths and weaknesses, and there are benefits to be gained from combining them, as elucidated in Hoare and He's programme to unify theories of programming [48].

The denotational semantics will associate a CSP program with a set of *observations* that may be made of it while it is executing. The denotational observations relate to executions given by the operational semantics and may be extracted directly. However, the benefits of the denotational approach derive from its *compositional* nature: the observations of a program may also be deduced from the observations of its components, without any need to refer to the operational semantics directly.

Different kinds of observation give rise to different semantic models. The four models introduced in this book—the traces model, the stable failures model, the failures-divergences-infinite traces model, and the timed failures model—all arise from progressively more detailed observations of processes, but all models have the same underlying philosophy: a process is determined by what may be observed of it. A program $P$ is always associated with the set

$$\{obs : OBS \mid P \text{ `exhibits' } obs\}$$

where the type of observation $OBS$ determines the model.

An important theme running through this book concerns the relationship between the untimed and the timed versions of CSP. The untimed language particularly is introduced in such a way as to make its relationship with the timed language plain. This book takes the view that analysis of system behaviour is appropriate at a number of levels of abstraction, and provides a unified framework for the results to be combined. Timed systems have a number of functional or logical properties which are independent of time considerations, and these are best treated within the more abstract untimed models without carrying the unnecessary additional baggage of timing information where it is not needed. Timed properties which rely on the timed behaviour must necessarily be verified in the less abstract timed world, but the relationships between the different levels of abstraction allow results to be carried from one level to another.

There are a number of reasons for taking this approach. The untimed theory is more abstract, enabling simpler proofs. It is also more mature, so there is more experience within the CSP community in analyzing and verifying untimed CSP descriptions, and there are more case studies. A third reason concerns computer aided verification, which is presently available for untimed CSP in the form of the Failures Divergences Refinement checker FDR but which is not yet available for timed CSP.

The models presented in this book are those that best support the theory of timewise refinement. In particular, the inclusion of infinite behaviours supports a much cleaner link between the untimed and the timed levels of abstraction, and enables that link to be exploited in the design and development of concurrent real time systems.

Steve Schneider
Royal Holloway
May 1999

# *Acknowledgments*

I first became involved in CSP on the Oxford MSc in Computation a decade ago. The Programming Research Group under Tony Hoare's leadership provided an inspiring and stimulating research environment, and I was fortunate to become a doctoral student at an exciting time in the development of Timed CSP. I enjoyed working closely with Jim Davies, Dave Jackson, Mike Reed and Bill Roscoe, and I am grateful to them for their friendship and advice over the years. I have also benefitted from working with other researchers including Jeremy Bryans, Tony Hoare, Alan Jeffrey, Guy Leduc, Luc Léonard, Mike Mislove, and members of the PRG and the ESPRIT SPEC and CONCUR projects.

Thanks are also due to Paul Baker, Phil Brooke, Jeremy Bryans, John Derrick, Simon Gay, Gavin Lowe, Joel Ouaknine, Bill Roscoe, Peter Ryan, Bryan Scattergood, Andrew Simpson, Dyke Stiles, Helen Treharne and Lok Yeung, for their careful reading of various drafts of this book, and for their comments, insights and valuable suggestions.

Finally, my special thanks go to my family: Elizabeth, Katherine and Eleanor, for their unfailing encouragement, support and sense of perspective, and for providing me with a reason to finish.

SAS

# *Contents*

*Preface* *v*

*Acknowledgments* *xi*

*Part I The language of CSP*

*1 Sequential processes* *1*
- *1.1 Events and processes* *1*
- *1.2 Performing events* *6*
- *1.3 Recursion* *11*
- *1.4 Choice* *18*

*2 Concurrency* *29*
- *2.1 Alphabetized parallel* *29*
- *2.2 Interleaving* *42*
- *2.3 Interface parallel* *48*

*3 Abstraction and control flow* *53*
- *3.1 Hiding* *53*
- *3.2 Event renaming* *60*
- *3.3 Sequential composition* *67*

3.4 Interrupt 70
3.5 Notes 72

Part II Analyzing Processes

4 Traces 85
4.1 Trace semantics 89
4.2 Recursion 115
4.3 Testing 131
4.4 Congruence 134

5 Specification and verification with traces 139
5.1 Property-oriented Specification 139
5.2 Verification 141
5.3 Recursion induction 153
5.4 Case study: Distributed sum 158
5.5 Process-oriented specification 166

6 Stable failures 171
6.1 Observing processes 172
6.2 Process semantics 176
6.3 Recursion 186

7 Specification and verification with failures 193
7.1 Property-oriented specification 193
7.2 Verification 195
7.3 Recursion induction 204
7.4 Process-oriented specification 207
7.5 Case study: Distributed sum 210

8 Failures, divergences, and infinite traces 219
8.1 Observing processes 219
8.2 Process semantics 227
8.3 Recursion 239
8.4 Specification and verification 246
8.5 Recursion induction 250
8.6 Case study: Distributed sum 254
8.7 Must testing and FDI equivalence 254

8.8 Notes 256

Part III Introducing time

9 The timed language 265
9.1 Timed computational model 266
9.2 Transitions 267
9.3 Performing events 268
9.4 Choice 273
9.5 Recursion 284
9.6 Concurrency 288
9.7 Abstraction 296
9.8 Flow of control 303

10 Timed transition systems 311
10.1 Evolution 311
10.2 Executions 313
10.3 Well-timed processes 325
10.4 Notes 327

Part IV Timed analysis

11 Semantics of Timed CSP 333
11.1 Timed observations 334
11.2 Timed failures semantics 345
11.3 Recursion 356
11.4 Testing and timed failures equivalence 361

12 Timed specification and verification 367
12.1 Specification 367
12.2 Verification 373
12.3 Recursion induction 383
12.4 Ill-timed processes 387
12.5 Case study: Fischer's protocol 388

13 Timewise refinement 397
13.1 Trace timewise refinement 398

13.2 Failures timewise refinement 408
13.3 Refinement and parallel composition 411
13.4 Case study: a railway crossing 429
13.5 FDI timewise refinement 434
13.6 Testing and timewise refinement 437
13.7 Notes 440

Appendix A: Event-based time 445
A.1 Standard CSP and tock 446
A.2 Translating from timed CSP 452
A.3 Notes 463

Appendix B: Model-checking with FDR 467
B.1 Interacting with FDR 468
B.2 How FDR checks refinement 471
B.3 Machine readable CSP 476

References 483

Notation 491
Index 496
Index of Processes 504

*Part I*

# *The language of CSP*

# 1 Sequential processes

## 1.1 EVENTS AND PROCESSES

Any approach to describing the world must concentrate on features of interest. Architects, engineers, economists, cartographers, biologists, physicists, and computer scientists all categorize and describe the world from their own particular point of view, appropriate to the phenomena they are trying to understand and control. They will focus on those aspects of the world relevant to their study.

This book is concerned with the description and analysis of systems which consist of interacting components. In such systems it is the myriad possibilities for interaction between components that are difficult to understand. Since we are interested not only in understanding such systems, but also in designing them, the description language used will influence how we think about systems, and will dictate the way in which these systems will be designed.

The language of Communicating Sequential Processes (CSP) was designed for describing systems of interacting components, and it is supported by an underlying theory for reasoning about them. The conceptual framework taken by CSP is to consider components, or *processes*, as independent self-contained entities with particular interfaces through which they interact with their environment. This viewpoint is compositional, in the sense that if two processes are combined to form a larger system, that system is again a self-contained entity with a particular interface—a (larger) process. This is the framework provided by CSP for analyzing the world.

EXAMPLE 1.1 The kitchen of a fast-food outlet might be considered as a process. Its interface will include the door through which the ingredients come in, the counter where the cooked food is passed to the till staff, and the tannoy on which orders come in.

Another process within the fast-food outlet is the customer serving area. The interface here will include the tills, the till counter where the customer's food is placed, the tannoy for relaying orders to the kitchen, the food counter for picking up food placed there by the kitchen.

The kitchen and the customer serving area may be considered as distinct processes, and this separation may be appropriate from the management and company organization point of view. Furthermore, their combination will also be a process, whose interface will include the tills, the till counter, and the door through which ingredients come into the kitchen.

The kitchen itself need not be considered as an atomic process, and may instead be viewed as a combination of more primitive processes, such as a grill process, a deep-fry process, a microwave process, and an ingredients-sort-and-distribute process. $\Box$

Since a process interacts with other processes only through its interface, the important information in the description of a process concerns its behaviour on that interface. In describing systems made up of interacting components and analyzing the effects of their interaction, the appropriate level will abstract away the internal workings of the process and will focus on its activity at the interface: its external activity.

The interface of a process will be described as a set of *events*. An event describes a particular kind of atomic indivisible action that can be performed or suffered by the process. In describing a process, the first issue to be decided must be the set of events which the process can perform. This set provides the framework for the description of the process.

EXAMPLE 1.2 A printer can accept jobs, and it can print them. Its interface may be given as the set $\{accept, print\}$. $\Box$

EXAMPLE 1.3 A telephone has 12 buttons, a handset, and a bell. The handset may be lifted or replaced. Its interface might be given as the set

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \#, *, handset.lift, handset.replace, ring\}$$

This set is precisely the ways in which the telephone can interact with its environment. $\Box$

EXAMPLE 1.4 A lift system which serves floors $0$ to $3$ has an *up* button on each floor (apart from the top), a *down* button on each floor (apart from the bottom) and a *goto*.*i* button for each floor, within the lift. It also has doors at each floor which can open and close. Finally, it has an emergency *halt* button within the lift. Its interface will be described by the following:

$$\begin{aligned}&\{up.0, up.1, up.2, down.1, down.2, down.3,\\&goto.0, goto.1, goto.2, goto.3,\\&open.0, close.0, open.1, close.1, open.2, close.2, open.3, close.3,\\&halt\}\end{aligned}$$

All interaction with the lift is via this set of events. $\Box$

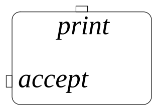


***Fig. 1.1*** A machine with buttons

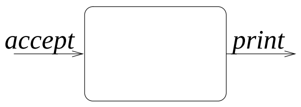


***Fig. 1.2*** A black box with wires

Processes may be thought of in a number of ways: a machine with a collection of buttons corresponding to the events in its interface, as in Figure 1.1; or alternatively as a black box with a collection of wires corresponding to the events in its interface, as illustrated by Figure 1.2.

The interface given for a process can be considered as its static specification. Its dynamic specification describes how it will actually behave at its interface. A process will be willing to engage in interface events only at particular times. More generally, there will be constraints on the sequences of events that it can engage in. For example, the lift in Example 1.4 would be required to alternate on the opening and closing of doors. The dynamic part of the process description will describe its permissible patterns of events.

## Transitions

An operational semantics provides a way of interpreting a language—of stepping through executions of programs written in that language. It describes an operational understanding of the language. CSP is concerned with the performance of events, so the operational semantics will describe at what stages events may occur.

The operational semantics in fact defines how a CSP interpreter should execute. It provides the first possible execution steps (if any) for any CSP process, together with an expression of the subsequent behaviour in the form of another CSP process. An execution step will be described in terms of *labelled transitions* of the form $P_1 \xrightarrow{\mu} P_2$, where $P_1$ and $P_2$ are both processes. This describes a *transition* from $P_1$ to $P_2$, or equivalently a change in state. The label $\mu$ describes the action which accompanies this transition. It can be either an external event (from $P_1$'s interface), a termination event $\checkmark$ (introduced on Page 11), or it might be an internal action $\tau$ which indicates that no interface event accompanied the change of state. The set of all possible external events is denoted $\Sigma$, so $\mu$ will range over $\Sigma \cup \{\checkmark, \tau\}$, which is written $\Sigma^{\checkmark,\tau}$. Roman variables *a*, *b*, *c*, will be used for events that must be external: they range over $\Sigma \cup \{\checkmark\}$, which is abbreviated $\Sigma^{\checkmark}$.

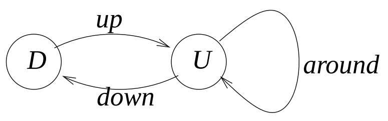


**Fig. 1.3** The finite state machine for $D$ and $U$

The labelled transition $P_1 \xrightarrow{\mu} P_2$ asserts that there is an execution of $P_1$ which begins with the occurrence of the event $\mu$, and its subsequent behaviour is that of process $P_2$. The operational semantics offers a way of stepping through executions one step at a time. Since any execution unfolds one step at a time, this operational semantics provides all the information necessary to step through an execution. At every stage, the rules will describe the next possible steps (if any) for the execution.

EXAMPLE 1.5 The following system has two states, $D$ and $U$, and three transitions between them:

- $D \xrightarrow{up} U$
- $U \xrightarrow{around} U$
- $U \xrightarrow{down} D$

This describes the finite state machine of Figure 1.3. □

Event names will be written in lower case, and process names written in upper case.

## Inference rules

An *inference rule* allows the *deduction* of a predicate from a collection of other predicates. It will be of the following general form:

$$\frac{\begin{array}{l}\text{antecedent } 1\\ \vdots\\ \text{antecedent } n\end{array}}{\text{conclusion}}\quad [\text{ side condition }]$$

This rule allows the conclusion to be deduced if all of the antecedents are true, and the side condition is also true. In the special case where there are no antecedents and no side condition, then the conclusion may be immediately deduced.

A number of conclusions which may all be drawn from the same set of antecedents may be listed as conclusions one after the other beneath the line. This provides an alternative to writing a separate rule with the same antecedents and side condition for each conclusion.

Inference rules will be used in two ways in this book. Firstly, rules may be given to formalize inferences concerning particular kinds of predicate. These rules can be independently checked by considering the meaning of the predicate. For example, the rule *Modus Ponens* can be given in this way:

$$\frac{\begin{array}{l} p \\ p \Rightarrow q \end{array}}{q}$$

If $p$ and $q$ are both logical statements, then *Modus Ponens* allows $q$ to be deduced from the pair of statements $p$ and $p \Rightarrow q$. The proof rule can be checked for soundness by considering the possible meanings of $p$ and $q$: when both antecedents are true, then so too must be the conclusion.

The law of the excluded middle is an example of a rule with no antecedents:

$$\frac{}{p \vee \neg p}$$

The inference rules given in the later chapters concerning **sat** specifications are of this kind: an independent definition of the **sat** relation is given, and the rules are sound with respect to this definition, and provide ways of reasoning about it.

Rules may also be used *axiomatically* to *define* predicates. For example, if the relation 'is a parent of' is already known, then a pair of rules can be used to define the relation 'is an ancestor of':

$$\frac{\begin{array}{l} p \text{ is an ancestor of } q \\ q \text{ is a parent of } r \end{array}}{p \text{ is an ancestor of } r}$$

$$\frac{}{p \text{ is an ancestor of } p}$$

The relation 'is an ancestor of' is defined to hold between two people precisely when the rules can be used to deduce this. Technically, it is the smallest relation closed under these inference rules.

Structured operational semantics are conventionally defined in this way, and this will be the approach taken in this book. The ternary relation $P_1 \xrightarrow{\mu} P_2$ between $P_1$, $P_2$, and $\mu$, asserts that there is a transition labelled $\mu$ between $P_1$ and $P_2$. The relation $\longrightarrow$ will be defined axiomatically through inference rules. A process $P_1$ can perform a $\mu$ transition to $P_2$ precisely when the relation $P_1 \xrightarrow{\mu} P_2$ can be deduced from the rules.

The operational semantics is just this relation between terms of the language and event labels.

## 1.2 PERFORMING EVENTS

The simplest process of all is *STOP*. This process is never prepared to engage in any of its interface events. It might be used to describe the fast-food outlet after it has closed down, or a broken printer that cannot accept or print jobs.

The operational semantics for *STOP* are extremely simple. It has no event transitions. Any execution of *STOP* will be unable to make any progress, and will remain in the same state forever. An explicit description of its interface will describe precisely what it is unable to perform.

### Event prefix

If $P$ is a CSP process, and $a \in \Sigma$ is an event in the interface of $P$, then the following new process may be constructed:

$$a \to P$$

It is pronounced '$a$ then $P$'. This process is initially able to perform only $a$, and after performing $a$ it behaves as $P$. The labelled transition semantics captures this understanding:

$$\boxed{\frac{}{(a \to P) \xrightarrow{a} P}}$$

There are no antecedents and no side condition to this rule. It is always the case that $a \to P$ may perform an $a$ transition and subsequently behave as $P$.

EXAMPLE 1.6 A one-shot printer is described by the process

$$PRINTER0 \quad = \quad accept \to print \to STOP$$

Initially it is able only to *accept* a job, after which it will behave as $print \to STOP$. This subsequent process is able to *print* a job, after which no further action is possible. Its complete maximal execution is described as

$$\begin{array}{l} accept \to print \to STOP \\ \quad \downarrow accept \\ print \to STOP \\ \quad \downarrow print \\ STOP \end{array}$$

The corresponding finite state machine is given in Figure 1.4. □

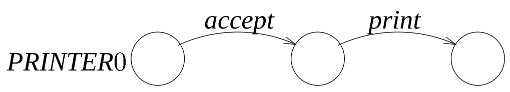


***Fig. 1.4*** The finite state machine for *PRINTER*0

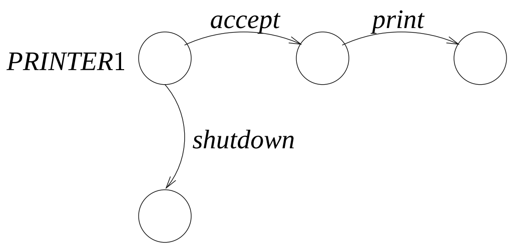


***Fig. 1.5*** The finite state machine for *PRINTER*1

## Choosing between events

If $A \subseteq \Sigma$ is a set of events, and for each $a$ in $A$ the process $P(a)$ is defined, then a new process can be defined:

$$x : A \rightarrow P(x)$$

This is called a menu choice, or prefix choice, since a menu of events $A$ is offered as a prefix to the subsequent behaviour. It is pronounced '$x$ from $A$ then $P(x)$'. This process is prepared initially to engage in any of the events in the set $A$. After an event $a$ is chosen, the subsequent behaviour is that of the process $P(a)$ corresponding to the event $a$.

EXAMPLE 1.7 A printer which initially has a *shutdown* option as well as an *accept* option can be described using this form of choice. The initial choice is between *accept* and *shutdown*. The process following *accept* is to be $print \rightarrow STOP$, and the behaviour subsequent to *shutdown* is simply *STOP*. This situation may be described as follows:

$$\begin{array}{rcl} PRINTER1 & = & x : \{accept, shutdown\} \rightarrow P(x) \\ \text{where} & & \\ P(accept) & = & print \rightarrow STOP \\ P(shutdown) & = & STOP \end{array}$$

The corresponding finite state machine is given in Figure 1.5. □

Prefix choice allows a notation for conditional choices to be introduced to CSP. In a choice $x : A \rightarrow P(x)$, the definition of $P(x)$ might involve a conditional. For example, the printer of the example above might have $P$ defined by

$$\begin{array}{lcll} P(x) & = & \mathit{print} \rightarrow \mathit{STOP} & \text{if } x = \mathit{accept} \\ & & \mathit{STOP} & \text{otherwise} \end{array}$$

or even by

$$\begin{array}{l} \textbf{if } x = \mathit{accept} \\ \textbf{then } \mathit{print} \rightarrow \mathit{STOP} \\ \textbf{else } \mathit{STOP} \end{array}$$

Neither of these is strictly within the language of CSP. Rather, they are constructions used in the definition of a parameterized process $P(x)$. However, they are conventionally used within CSP descriptions, resulting for example in a description of *PRINTER*1 as follows:

$$\begin{array}{lcll} \mathit{PRINTER}1 & = & x : \{\mathit{accept}, \mathit{shutdown}\} \rightarrow & \textbf{if } x = \mathit{accept} \\ & & & \textbf{then } \mathit{print} \rightarrow \mathit{STOP} \\ & & & \textbf{else } \mathit{STOP} \end{array}$$

In the case where the choice set $A$ is finite, of the form $\{a_1, a_2 \ldots a_n\}$, the branches of the choice may be listed explicitly as follows:

$$\begin{array}{l} a_1 \rightarrow P(a_1) \\ \mid a_2 \rightarrow P(a_2) \\ \vdots \\ \mid a_n \rightarrow P(a_n) \end{array}$$

EXAMPLE 1.8 The printer above can be written as follows:

$$\begin{array}{lcl} \mathit{PRINTER}1 & = & \mathit{accept} \rightarrow \mathit{print} \rightarrow \mathit{STOP} \\ & & \mid \mathit{shutdown} \rightarrow \mathit{STOP} \end{array}$$

The events offered by the choice are listed explicitly. □

EXAMPLE 1.9 A printer which begins with a *startup* event:

$$\begin{array}{lcll} \mathit{PRINTER}2 & = & \mathit{startup} \rightarrow & (\mathit{accept} \rightarrow \mathit{print} \rightarrow \mathit{STOP} \\ & & & \mid \mathit{shutdown} \rightarrow \mathit{STOP}) \end{array}$$

The choice is offered after the first event. □

In the case where the set $A$ is empty, the choice is equivalent to $STOP$. No initial events are possible, so there can be no subsequent behaviour.

The transitions for $x : A \rightarrow P(x)$ are given by the following rule:

$$\frac{}{(x : A \rightarrow P(x)) \xrightarrow{a} P(a)} \quad [\, a \in A \,]$$

For each $a \in A$ there is a corresponding transition. There are no other transitions.

## Compound events

Events are considered to be atomic and indivisible in their occurrence. However, a single event may still contain various pieces of information, so events can have some structure. An example of this has already been given in Example 1.4, where events are structured by the kind of event they are, together with the floor they are concerned with. Another instance of a structured event is given by a communication channel which carries messages. In order to model values $v$ being communicated along channel $c$, each possible communication is described as a separate possible event $c.v$ in the interface of the process. If a process $P$ has an input channel $in$ that carries 0s and 1s, then both $in.0$ and $in.1$ will appear in the interface set of $P$. The event $in.0$ describes the appearance of value 0 on channel $in$. Events $in.0$ and $in.1$ are distinct events, though the intention is to consider them both as inputs of particular values along channel $in$.

If $c$ is a particular channel name, and $T$ is the *type* of the channel—the set of values that may be passed along it—then the set $\{c.t \mid t \in T\}$ will be the set of events associated with $c$. For convenience this will be denoted $c.T$. More generally, it is often useful to allow a Cartesian generalization of the 'dot' separator to sets. For example, $c.d.S.T = \{c.d.s.t \mid s \in S \wedge t \in T\}$.

EXAMPLE 1.10 The alphabet of the kitchen given in Example 1.1 might be given by

$door.I \cup counter.F \cup tannoy.O$

where $I$ is the set of all possible ingredients, $F$ is the set of food dishes, and $O$ is the seet of possible orders. □

## Input and Output

If $c$ is a channel name of type $T$, and $v$ is a particular value of type $T$, then the CSP expression

$c!v \rightarrow P$

describes a process which is initially willing to output $v$ along channel $c$, and subsequently behave as $P$. This means that the only event it is initially willing to perform is $c.v$, and its transition semantics is

$$\boxed{\frac{}{(c!v \to P) \xrightarrow{c.v} P}}$$

This process has the same behaviour as $c.v \to P$, but the intention of the designer in considering it as output is made explicit. It is simply a convenient syntactic distinction.

If processes $P(x)$ are defined for each $x \in T$ then the CSP input expression

$$c?x : T \to P(x)$$

describes a process which is initially ready to accept any value $x$ of type $T$ along channel $c$. Its subsequent behaviour, described by $P(x)$, is determined by the value $v$ that it receives as input.

$$\boxed{\frac{}{(c?x : T \to P(x)) \xrightarrow{c.v} P(v)} \quad [\, v \in T \,]}$$

EXAMPLE 1.11 A 'squaring' server could be described by

$$in?x : \mathbb{N} \to out!(x^2) \to STOP$$

The output value is the square of the input. □

EXAMPLE 1.12 If *JOBS* is the set of all possible print jobs that can be accepted by a printer, then a more detailed description of a one-shot printer would be

$$PRINTER3 \quad = \quad accept?j : JOBS \to print!j \to STOP$$

□

EXAMPLE 1.13 A multiplication server could be described by

$$in?m : \mathbb{N} \to in?n : \mathbb{N} \to out!(m * n) \to STOP$$

or alternatively by

$$in?(m, n) : (\mathbb{N} \times \mathbb{N}) \to out!(m * n) \to STOP$$

The first process takes in one input followed by another, and then produces an output. The second process requires the pair of numbers to be submitted as a single input. □

### Successful termination

Successful termination is the point that a process reaches when its execution has completed. The process representing this state is

$$SKIP$$

which can do nothing except indicate that it has reached termination. It achieves this by performing the special termination event $\checkmark$. It does nothing else.

$$\boxed{\frac{}{SKIP \xrightarrow{\checkmark} STOP}}$$

The event $\checkmark$ is a special event used purely to denote termination, so it is not a member of the universal set of events $\Sigma$. It therefore cannot appear as an event prefix $a \rightarrow P$, or as one of the choices in a menu choice: such processes describe behaviour subsequent to their events, and this is inappropriate for termination.

## 1.3 RECURSION

The process constructors introduced thus far allow the construction only of finite processes, which execute for a finite number of steps before stopping. In order to describe infinitely executing processes, a *recursion* construct is introduced. This allows looping executions to be defined. For example, the process $LIGHT = on \rightarrow off \rightarrow LIGHT$, illustrated in Figure 1.6, allows the alternation of the events *on* and *off* indefinitely.

A process *name* $N$ may be used as a component process in a process definition. It is bound by the definition

$$N = P$$

where $P$ is an arbitrary CSP expression which may include process name $N$. The process expression $P$ is the *body* of the recursive definition.

The rule for unwinding a process name $N$ recursively bound to a process definition $P$ is as follows:

$$\boxed{\frac{P \xrightarrow{\mu} P'}{N \xrightarrow{\mu} P'} \; [\, N = P \,]}$$

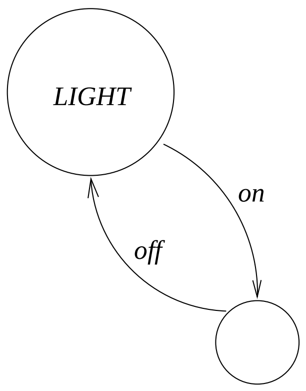


***Fig. 1.6*** The finite state machine for *LIGHT*

This rule states that any execution of $P$ will be an execution of $N$.

Another way to consider $P$ is as a process dependent on $N$. To make this relationship explicit, $P$ may also be written as $F(N)$.

The notation $P_1[P_2/N]$ is used to denote the substitution meta-operation, where all (free) instances of the process name $N$ appearing in $P_1$ are replaced by the process expression $P_2$. For example

$$\begin{aligned}(on \to \mathit{off} \to \mathit{LIGHT})[on \to \mathit{STOP}/\mathit{LIGHT}] &= on \to \mathit{off} \to on \to \mathit{STOP} \\ (on \to \mathit{off} \to \mathit{LIGHT})[Y/\mathit{LIGHT}] &= on \to \mathit{off} \to Y\end{aligned}$$

If $N = P$ is a recursive definition, then $F(Y) = P[Y/N]$ is the function (in $Y$) corresponding to the body of the definition.

EXAMPLE 1.14 The process *LIGHT* is recursively defined as follows:

$$\mathit{LIGHT} = on \to \mathit{off} \to \mathit{LIGHT}$$

Equivalently, $\mathit{LIGHT} = F(\mathit{LIGHT})$, where $F(Y) = on \to \mathit{off} \to Y$.

The execution of *LIGHT* unfolds as follows:

$$\begin{array}{l} LIGHT \\ \quad\downarrow on \\ off \rightarrow LIGHT \\ \quad\downarrow off \\ LIGHT \\ \quad\downarrow on \\ off \rightarrow LIGHT \\ \quad\vdots \end{array}$$

It may alternate between the states *LIGHT* and *off* $\rightarrow$ *LIGHT* forever. □

EXAMPLE 1.15 The one-place buffer *COPY* is initially ready to accept any message of type *T* as input, and will then hold it until it is output.

$$COPY \quad = \quad in?x : T \rightarrow out!x \rightarrow COPY$$

After output, it returns to its initial state. □

EXAMPLE 1.16 A specification of a railway crossing describes the required interactions between the raising and lowering of the gate, and the arrival and departure of a train.

$$\begin{array}{lll} CROSS & = & train.approach \rightarrow train.enter \rightarrow train.leave \rightarrow CROSS \\ & & \mid gate.raise \rightarrow train.approach \rightarrow gate.down \rightarrow \\ & & \qquad train.enter \rightarrow train.leave \rightarrow CROSS \end{array}$$

The initial state has the gate lowered, blocking road vehicles from crossing the rails. Either the gate is raised, or else a train approaches the crossing. If the gate is raised then it must be lowered on the approach of a train. If the train enters the crossing then it must leave before the gate may be raised. The transition graph for *CROSS* is given in Figure 1.7. Compound events are used here simply to associate each event with either the train or the gate. □

## Mutual Recursion

A collection of recursive definitions will bind a number of process names to process definitions. It is often useful to allow the process definitions to contain a number of the names being defined, so that in fact the various processes are defined in terms of each other. This construction is known as *mutual recursion*.

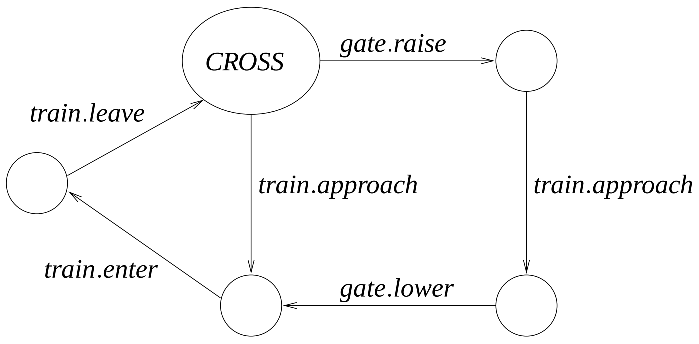


***Fig. 1.7*** Transition graph for *CROSS*

EXAMPLE 1.17 The process *LIGHT* may be defined in terms of a process *ON*, which is itself defined in terms of *LIGHT*:

$$\begin{aligned} \mathit{LIGHT} &= \mathit{on} \to \mathit{ON} \\ \mathit{ON} &= \mathit{off} \to \mathit{LIGHT} \end{aligned}$$

These recursive definitions define two processes, each in terms of the other. □

In order for a set of recursive definitions to be a mutual recursion, each name appearing in any of the process bodies must be bound in one of the recursive definitions. The single definition $\mathit{LIGHT} = \mathit{on} \to \mathit{ON}$ by itself is not suitable as a recursive definition: the process name *ON* must also be bound.

The transition rule for unwinding a recursive definition is exactly the same as that given for a single recursion. The transitions that can be made for a process name $N_i$ in the context of a collection of bindings which binds $N_i$ to $P_i$ are precisely the transitions of $P_i$.

$$\boxed{\quad \frac{P_i \xrightarrow{\mu} P'}{N_i \xrightarrow{\mu} P'} \quad [\ N_i = P_i\ ] \quad}$$

The process *CROSS* defined in Example 1.16 might also have been given using a mutual recursion:

$$\begin{aligned} \mathit{CROSS} &= \mathit{gate.raise} \to \mathit{train.approach} \to \mathit{gate.lower} \to \mathit{ENT} \\ &\quad\ \mid \mathit{train.approach} \to \mathit{ENT} \\ \mathit{ENT} &= \mathit{train.enter} \to \mathit{train.leave} \to \mathit{CROSS} \end{aligned}$$

The behaviour of this version of *CROSS* is indistinguishable from the single recursive process given earlier.

One execution of *CROSS* is as follows:

$$\begin{array}{l}
\mathit{CROSS} \\
\quad \downarrow \mathit{gate.raise} \\
(\mathit{train.approach} \to \mathit{gate.lower} \to \mathit{ENT}) \\
\quad \downarrow \mathit{train.approach} \\
(\mathit{gate.lower} \to \mathit{ENT}) \\
\quad \downarrow \mathit{gate.lower} \\
\mathit{ENT} \\
\quad \downarrow \mathit{train.enter} \\
(\mathit{train.leave} \to \mathit{CROSS}) \\
\vdots
\end{array}$$

This is one of the paths through the transition graph shown in Figure 1.7. The names of the recursive processes used in the definition of *CROSS* have been chosen to reflect the important states of the system: *ENT* is the point at which the train will enter the crossing.

This convention may be used more generally with a family of process names $N(i)$ parameterized by $i \in I$. A mutual recursion will bind them to a family of processes containing these names. Alternatively they will be bound to a family of functions $F(i)$ where each is a function of the family of names $N(i)$.

EXAMPLE 1.18 A heater has four power settings, which can be changed by the events *up* and *down*. We use the four process names $\mathit{HEATER}(0)$, $\mathit{HEATER}(1)$, $\mathit{HEATER}(2)$ and $\mathit{HEATER}(3)$ to describe the four possible states. Their interrelationships are described by mutual recursion:

$$\begin{array}{lcl}
\mathit{HEATER}(0) & = & \mathit{up} \to \mathit{HEATER}(1) \\
\mathit{HEATER}(1) & = & \mathit{up} \to \mathit{HEATER}(2) \mid \mathit{down} \to \mathit{HEATER}(0) \\
\mathit{HEATER}(2) & = & \mathit{up} \to \mathit{HEATER}(3) \mid \mathit{down} \to \mathit{HEATER}(1) \\
\mathit{HEATER}(3) & = & \mathit{down} \to \mathit{HEATER}(2)
\end{array}$$

At any point in the execution, the process will be at one of the $\mathit{HEATER}(i)$ nodes. The value of $i$ might be thought of as the state of the heater, corresponding to the setting on a dial. $\square$

It is appropriate to keep track only of those aspects of internal state that have an impact on the external behaviour patterns of the process. The CSP notation is intended primarily to support description and analysis of processes in terms of their interactions. However, the interactions possible for a process might depend on the value of some internal state variable, and so it is necessary in such situations to keep track of the relevant information, but only in so far as it affects the process' external behaviour. In the *HEATER* example above, the value of the state determines how many *up* and *down* events are possible. The heater might also contain a thermostat, but if its setting does not have any effect on the behaviour under consideration, then its value is superfluous to the description of the process, and should not be included.

EXAMPLE 1.19 A counter can be *incremented* or *decremented* at any point, provided the total number of *decrement* events does not exceed the number of *increment* events. The family of process names $COUNT(i)$ will be used to define *COUNTER*, where $i$ will track the difference between the number of *increment* events and the number of *decrement* events.

$$\begin{aligned} COUNT(0) &= increment \rightarrow COUNT(1) \\ COUNT(i) &= \quad increment \rightarrow COUNT(i+1) \qquad \text{if } i > 0 \\ &\quad\;\; \mid decrement \rightarrow COUNT(i-1) \end{aligned}$$

The counter begins at $0$:

$$COUNTER = COUNT(0)$$

If there could be any number of *increment* and *decrement* events, in any order, then it would be unnecessary to keep track of the difference between them, and the process description

$$C = increment \rightarrow C \mid decrement \rightarrow C$$

would be sufficient. State information should be carried only where it affects the possible executions of the process. □

EXAMPLE 1.20 An *ACCUMULATOR* is used to keep track of running totals of sequences of numbers. It has a *reset* event, a *query* channel on which the current total can be output, and an *add* channel where it is possible to add another number. The family of process names $TOT(i)$ will be used to define this process, where $i$ represents the running total.

$$\begin{aligned} TOT(i) &= reset \rightarrow TOT(0) \\ &\quad \mid query!i \rightarrow TOT(i) \\ &\quad \mid add?x : \mathbb{N} \rightarrow TOT(i+x) \end{aligned}$$

*ACCUMULATOR* can now be defined:

$$ACCUMULATOR = TOT(0)$$

Its initial state will be $0$. □

Indices for recursive process names need not be restricted to numbers: more generally, any kind of index may be used. This allows processes to be parameterized by more abstract values such as sets or strings. They do not need to be directly representable within a computer.

EXAMPLE 1.21 A process which models a set of elements of type $T$ allows elements to be added, and provides information about whether a particular element is in the set. It will be parameterized by $S$, the current set of elements:

$$\begin{array}{rcl} SET & = & SET(\{\}) \\ SET(S) & = & add?x : T \rightarrow SET(S \cup \{x\}) \\ & & \mid query?y : T \rightarrow \left\{ \begin{array}{ll} answer!yes \rightarrow SET(S) & \text{if } y \in S \\ answer!no \rightarrow SET(S) & \text{otherwise} \end{array} \right. \end{array}$$

The response depends both on the parameter $S$ and the input $y$. □

EXAMPLE 1.22 A buffer of infinite capacity is always prepared to input a fresh message, and when it is nonempty it is prepared to output the message at the head of the queue. It may be parameterized by the sequence $s$ of messages currently in the buffer.

$$\begin{array}{rcl} BUFFER(\langle\rangle) & = & in?x : M \rightarrow BUFFER(\langle x\rangle) \\ BUFFER(\langle y\rangle \frown s) & = & in?x : M \rightarrow BUFFER(\langle y\rangle \frown s \frown \langle x\rangle) \\ & & \mid out!y \rightarrow BUFFER(s) \end{array}$$

$x$ and $y$ range over $M$, and $s$ ranges over $M^*$, the (finite) strings of elements of $M$. The notation $\langle m\rangle$ represents a singleton sequence containing just $m$, and '$\frown$' is sequence concatenation, discussed in more detail in Section 4.

An initially empty buffer is described by

$$BUFFER \quad = \quad BUFFER(\langle\rangle)$$

One execution of $BUFFER$ is

$$\begin{array}{l} BUFFER \\ \quad \downarrow in.3 \\ BUFFER(\langle 3\rangle) \\ \quad \downarrow in.8 \\ BUFFER(\langle 3, 8\rangle) \\ \quad \downarrow out!3 \\ BUFFER(\langle 8\rangle) \\ \quad \vdots \end{array}$$

The parameter consists of the items still in the buffer. □

EXAMPLE 1.23 The description of a stack is very similar to that of the buffer:

$$\begin{array}{rcl} STACK(\langle\rangle) & = & push?x : M \rightarrow STACK(\langle x\rangle) \\ STACK(\langle y\rangle \frown s) & = & push?x : M \rightarrow STACK(\langle x\rangle \frown \langle y\rangle \frown s) \\ & & \mid pop!y \rightarrow STACK(s) \end{array}$$

The only difference is that input messages are placed at the beginning of the string rather than at the end. $\square$

## 1.4 CHOICE

Prefix choice has already introduced the possibility of process executions having a number of possible courses of action. Whereas that operator offers a choice between events, this section will introduce choices between processes.

In concurrent systems it is useful to distinguish between the cases where control over resolution of choice resides within a process itself, and where control is outside it. For example, a car showroom advertising cars in any colour might allow either the customer or the manufacturer Henry Ford [1] to make the choice; and these two possibilities are different. The distinction is important in concurrent systems, since problems may arise if two processes have both been given control over a particular choice. If both Henry Ford and the customer are considered to have control over the choice of colour then problems arise if they do not agree. It is therefore important to distinguish between *external choice* where control over the choice is external to a process, and *internal choice* where the environment of the process has no such control.

### External choice

An external choice between two processes is initially ready to perform the events that either process can engage in. The choice is resolved by the performance of the first event, in favour of the process that performs it. This choice is written

$$P_1 \;\square\; P_2$$

and is pronounced '$P_1$ external choice $P_2$'

[1] Ford offered the purchasers of his cars the choice of "any colour as long as it's black".

EXAMPLE 1.24 A particular bus journey is covered by two bus routes: the 37 and the 111. The service offered for that journey is then described as the choice between these two bus services.

$$\mathit{SERVICE} \;=\; \mathit{BUS\_37} \;\Box\; \mathit{BUS\_111}$$

This choice can be described even before the initial events of the *BUS* processes are known.

The bus services are used to travel from the bus station at *A* to a destination at *B*. The pertinent events for this journey in the description of a *BUS* process are *board*, *alight*, and *pay*.

$$\begin{aligned}\mathit{BUS\_37} \;=\; \mathit{board}.37.A \to (\; & \mathit{pay}.90 \to \mathit{alight}.37.B \to \mathit{STOP}\\ & \mid \mathit{alight}.37.A \to \mathit{STOP})\end{aligned}$$

On boarding the bus at *A*, a passenger must either pay the fare and then travel to *B*, or alight again at *A* without traveling.

The rival bus route charges a lower fare.

$$\begin{aligned}\mathit{BUS\_111} \;=\; \mathit{board}.111.A \to (\; & \mathit{pay}.70 \to \mathit{alight}.111.B \to \mathit{STOP}\\ & \mid \mathit{alight}.111.A \to \mathit{STOP})\end{aligned}$$

The description *SERVICE* describes the situation in the bus station. There is a choice of two buses, and the choice between them is resolved when the first one is boarded. □

The transition rules for external choice reflect the fact that the first external event resolves the choice in favour of the process performing the event, and that the choice is not resolved on the occurrence of internal events.

$$\frac{P_1 \xrightarrow{a} P_1'}{\begin{array}{l}P_1 \Box P_2 \xrightarrow{a} P_1'\\ P_2 \Box P_1 \xrightarrow{a} P_1'\end{array}} \qquad \frac{P_1 \xrightarrow{\tau} P_1'}{\begin{array}{l}P_1 \Box P_2 \xrightarrow{\tau} P_1' \Box P_2\\ P_2 \Box P_1 \xrightarrow{\tau} P_2 \Box P_1'\end{array}}$$

Control over resolution of the choice is external because the events of both choices are initially available. Considering processes as machines with buttons, the buttons that are initially enabled are those enabled by either of the choice processes. The choice is made externally because the choice of which button to press is not restricted by the process. The choice of buses is not made by the process *SERVICE*, but this choice is instead offered to the customer.

However, if the same event is offered by both of the choice processes, then an external agent will not have control over which process is chosen. An external agent has control only over the choice of initial event, not over the possible subsequent behaviours in the case where both processes offer the chosen initial event.

EXAMPLE 1.25 The previous description of the bus service provided at the bus station was appropriate in the case where the passenger looks at the number on the front of the bus and so can distinguish the events $board.37.A$ and $board.111.A$.

The buses available to a passenger who cannot read the bus number are better described simply as two buses $BUS\_1$ and $BUS\_2$.

$$\begin{aligned}
BUS\_1 &= board.A \rightarrow (\ pay.90 \rightarrow alight.B \rightarrow STOP \\
&\qquad\qquad\qquad \mid alight.A \rightarrow STOP) \\
BUS\_2 &= board.A \rightarrow (\ pay.70 \rightarrow alight.B \rightarrow STOP \\
&\qquad\qquad\qquad \mid alight.A \rightarrow STOP)
\end{aligned}$$

The choice $BUS\_1 \,\square\, BUS\_2$ still offers the option of boarding either bus, but since the two *board* events are not distinguished, the passenger has no control over which bus is boarded. Ignoring the number on the front of the bus results in the inability to distinguish routes. The passenger becomes unable to choose which fare to pay, since no further control over the choice is possible. In the previous case, the passenger could control the fare by ensuring the appropriate bus was boarded. □

## Indexed external choice

The binary form of external choice can be generalized to an external choice between any finite number of processes. If $I$ is a finite indexing set (which can be empty) such that $P_i$ is defined for each $i \in I$, then it may be given an operational semantics as follows:

$$\boxed{\frac{P_j \xrightarrow{a} P'}{\square_{i \in I} P_i \xrightarrow{a} P'}\ [\, j \in I\,] \qquad \frac{P_j \xrightarrow{\tau} P'_j}{\square_{i \in I} P_i \xrightarrow{\tau} \square_{i \in I} P'_i}\ [\, j \in I\,]}$$

where $P'_i = P_i$ for $i \neq j$.

In fact the relationship between indexed external choice and binary external choice is captured by the following alternative definition of indexed external choice in terms of the binary operator.

$$\begin{aligned}
\square_{j \in \{i\}} P_j &= P_i \\
\square_{j \in I \cup \{i\}} P_j &= (\square_{j \in I} P_j) \,\square\, P_i
\end{aligned}$$

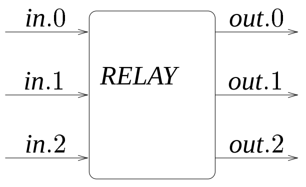


***Fig. 1.8*** The *RELAY* process

The external choice operator is associative, in the sense that $P_1 \;\Box\; (P_2 \;\Box\; P_3)$ and $(P_1 \;\Box\; P_2) \;\Box\; P_3$ have the same execution possibilities[2]. It is also commutative: $P_1 \;\Box\; P_2$ and $P_2 \;\Box\; P_1$ also have the same possible execution patterns. These two properties ensure that the order in which components are added to an indexed choice is irrelevant to the resulting behaviour of the choice. The only information required is the identity of the actual processes to be combined.

EXAMPLE 1.26

$$RELAY \quad = \quad \Box_{i \in I}\, in.i?x : T \rightarrow out.i!x \rightarrow RELAY$$

This process describes a relay service between a number of channels of the form $in.i$ and $out.i$ where $i$ is in some (finite) indexing set $I$. It is prepared to input a message $x$ along any of the $in.i$ channels, and then output it along the corresponding output channel $out.i$.

Observe that this description has exactly the same transitions as the alternative description

$$RELAY2 \quad = \quad in?i?x : I \times T \rightarrow out!i!x \rightarrow RELAY2$$

The difference is in the intention of the designer. In the first case, the model is of a process with a number of channels of the form $in.i$ and $out.i$, each of type $T$. The picture of this process is given in Figure 1.8.

In the second case, the model is of a process with a single input channel and a single output channel of type $I \times T$. This is pictured in Figure 1.9.

An event of the form $in.2.7$ can be considered either as the message '7' on the channel $in.2$, or as the message '$2.7$' on channel $in$. The transition system treats these both as the same single event.

[2]Technically, they are *strongly bisimilar* (see [77])—any internal or visible transition that one process can perform can be matched by the other. This is what is meant in this chapter by two processes having the same execution possibilities.

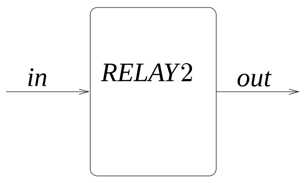


**Fig. 1.9** The *RELAY*2 process

Observe however that *RELAY*2 is well-defined even in the case where $I$ is infinite, whereas *RELAY* is not well-defined in this case. This is because external choices $\Box_{i \in I} P_i$ are not permitted over infinite sets $I$. □

EXAMPLE 1.27 A mail system connects a set of nodes *NODE*. It may accept an input at any node $l$ consisting of a destination and message $(d.m)$. This is captured as an input $in_l?(d.m)$. When there are such pairs $(d.m)$ in the system, then it may also perform $out_d!m$ corresponding to outputting message $m$ at the destination node $d$.

This specification of a mail system may be described using indexed external choice within a mutually recursive definition. The *MAIL* processes are indexed by multi-sets (or bags), which maintain the number of copies of each element. A fresh copy is added each time a message is input, and one copy is removed when output occurs.

$$\begin{array}{rcl} MAIL & = & MAIL_{\{\}} \\ MAIL_{\{\}} & = & \Box_{l \in NODE}\, in_l?(d.m) \to MAIL_{\{(d.m)\}} \\ MAIL_B & = & \Box_{l \in NODE}\, in_l?(d.m) \to MAIL_{B \uplus \{(d.m)\}} \\ & & \Box\ \Box_{(d.m) \in B}\, out_d!m \to MAIL_{B \setminus \{(d.m)\}} \end{array}$$

where $B$ ranges over non-empty bags, $\uplus$ is bag union, and $\setminus$ is bag subtraction. □

## Internal choice

A process considered as a specification describes a contract between the customer and the system designer. It encapsulates the behaviour of the system that is acceptable to the customer, and gives the designer the requirements that must be met.

The internal choice operator is commonly used as a specification construct. It is a choice over which the user has no control, and for this reason it is often called nondeterministic choice. The process

$$P_1 \sqcap P_2$$

pronounced '$P_1$ internal choice $P_2$', describes a choice between $P_1$ and $P_2$, and the choice is resolved by the process itself, without any influence from its environment.

Transition rules given for a specification construct cannot completely characterize the nature of this construct, since they provide a particular approach to implementation. One way of implementing the choice construct is to resolve the choice immediately. This is accompanied by a silent transition, due to the state change from $P_1 \sqcap P_2$ to one of its components. Either of the choices is possible:

$$\frac{}{\begin{array}{c}(P_1 \sqcap P_2) \xrightarrow{\tau} P_1 \\ (P_1 \sqcap P_2) \xrightarrow{\tau} P_2\end{array}}$$

These rules describe an operational understanding of one way this choice could be implemented, though its use is more often as a specification construct. The process $P_1 \sqcap P_2$ is a process which is guaranteed to behave on any particular execution either as $P_1$ or as $P_2$. As a specification, if $P_1 \sqcap P_2$ describes the customer requirement then the implementor is free to provide either $P_1$ or $P_2$ for any execution and the customer will find either acceptable.

There are a number of ways a system designer might choose to provide a system which meets the specification $P_1 \sqcap P_2$:

- $P_1$ and $P_2$ could both be developed, and whenever the process is run then a coin is tossed to decide which one to provide.
- $P_1$ and $P_2$ could both be constructed, and whenever the process is run then resource considerations determine which one is provided
- $P_1$ alone is provided
- $P_2$ alone is provided

These possibilities are illustrated in Figure 1.10. In each case a black box labelled with $P_1 \sqcap P_2$ is provided, but the implementations inside the boxes are different.

EXAMPLE 1.28 A mail router program might offer one of two routes: $ROUTER = VIA\_A \sqcap VIA\_B$. Whenever this program is invoked, the choice is resolved at run-time internally by considering the network traffic. The user is not concerned with the route, but simply in the correct delivery of the message, and is therefore happy to devolve responsibility for making the choice to the *ROUTER* program. □

EXAMPLE 1.29 A bus company guarantees to provide buses between *A* and *B*, but does not guarantee any particular route. There are two routes, the 37 and the 111. The passenger is happy to accept either, so the service offered by $BUS\_37 \sqcap BUS\_111$ is acceptable. The bus

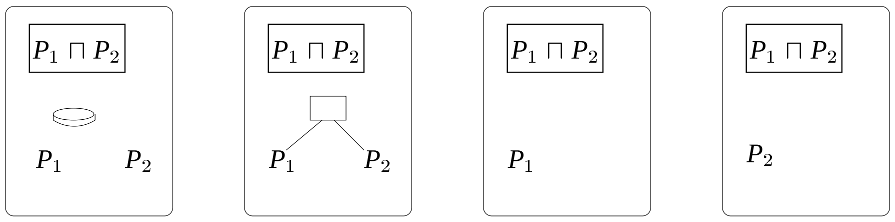


***Fig. 1.10*** Implementing $P_1 \sqcap P_2$

company decides to scrap the 37 bus service and run only the 111. This is indistinguishable to the customer from the situation where the decision to run the 111 in preference to the 37 is in fact made every morning. □

EXAMPLE 1.30 A customer who will accept a car of any colour must necessarily find a black car acceptable. A manufacturer who guarantees to provide a car meeting the specification $CAR_{black} \sqcap CAR_{coloured}$ may decide always to provide $CAR_{black}$. □

## Indexed internal choice

Since the internal choice operator corresponds to the disjunction operator as used in specification, it is natural to generalize it. If $J$ is a set of indices (which may be finite or infinite[3], but must be non-empty) and $P_i$ is defined for each $i \in J$, then the process

$$\bigsqcap_{i \in J} P_i$$

is a process which can behave as any of the $P_i$. As a specification this process describes the requirement that any execution should be appropriate to at least one of the $P_i$.

Operationally the indexed internal choice operator resolves immediately to one of its arguments:

$$\boxed{\frac{}{(\bigsqcap_{i \in J})P_i \xrightarrow{\tau} P_j} \quad [\, j \in J \,]}$$

EXAMPLE 1.31 The range of possibilities for a random number generator might be described by the infinite choice

$$\bigsqcap_{n \in \mathbb{N}} out!n \to STOP$$

[3]technically, there is a given universal set of indices which contains $J$

Any positive integer might be output. CSP does not express the probabilities of the numbers, it simply records the fact that they are possible. □

EXAMPLE 1.32 A process which can perform some event from the set of events $A$, but where its environment has no control over which could be described as follows:

$$\sqcap_{a \in A} a \rightarrow STOP$$

A process $D$ which can repeatedly perform some event from the set $A$ could be defined recursively as follows:

$$D = \sqcap_{a \in A} a \rightarrow D$$

The choice can be resolved differently each time round the recursive loop. □

## Exercises

EXERCISE 1.1 Give suitable interface sets for the following. In each case you should decide the events that would be required in a description of how the process behaves.

1. A video recorder
2. A vending machine
3. An automated teller machine
4. A personal computer
5. A computer chip
6. A telephone answering machine
7. A multiplexor
8. An analogue to digital converter

EXERCISE 1.2 Write a CSP description of a square-root server with channels *in* and *out*.

EXERCISE 1.3 Write a CSP description of a multiplication component which has three input channels $in_1$, $in_2$, and $in_3$, and one output channel *out*. It reads in one number from each input channel (in any order) and outputs their product.

EXERCISE 1.4 Write a CSP description of a small fast food outlet which serves only two items: burgers at 75p, and chicken at 95p. The sequence of interactions involves placing an order for one of the items, paying for the order, and receiving the order. Only one customer at a time can be handled.

EXERCISE 1.5 Give the transition graph for process *PRINTER*2 of Example 1.9.

EXERCISE 1.6 Give the transition graph for the process *HEATER* of Example 1.18.

EXERCISE 1.7 Give the transition graph for the process *COPY* of Example 1.15.

EXERCISE 1.8 What are the possible executions for $X = X$ ?

EXERCISE 1.9 Define a variant of the *COPY* process which accepts a value on its input channel, and stops if that value is $0$, otherwise it outputs it and begins again.

EXERCISE 1.10 Define a process with an interface consisting of the events *press* and $\mathit{finish}.\mathbb{Z}$. It accepts a number of *press* events, and then outputs along the channel *finish* the number of *press* events that have occurred, after which it stops.

EXERCISE 1.11 Are the choices in the following processes internal or external?

1. A shop which offers discounts of 10%, 30% or 50% on sale items
2. A cafe which offers *tea* or *coffee*
3. A mail-order book company which offers the choice between sending back the form within two weeks, or receiving the book-of-the-month.
4. A lottery ‘lucky dip’ machine which gives any 6 numbers of 49 possibilities.

EXERCISE 1.12 Write a process which offers a choice between three bus routes.

EXERCISE 1.13 Write a CSP process which describes the choices presented by a sweet trolley containing two pieces of cheese cake, one piece of apple pie and one piece of chocolate cake.

EXERCISE 1.14 Write a process which describes the pattern of choices presented by the maze in Figure 1.11

1. if the alphabet is $\{\mathit{east}, \mathit{west}, \mathit{north}, \mathit{south}, \mathit{in}, \mathit{out}\}$
2. if the alphabet is $\{\mathit{left}, \mathit{right}, \mathit{forward}, \mathit{back}, \mathit{in}, \mathit{out}\}$, where for example *right* means ‘turn right and then move’.

EXERCISE 1.15 Give the transition graph of the process *SERVICE* of Example 1.24.

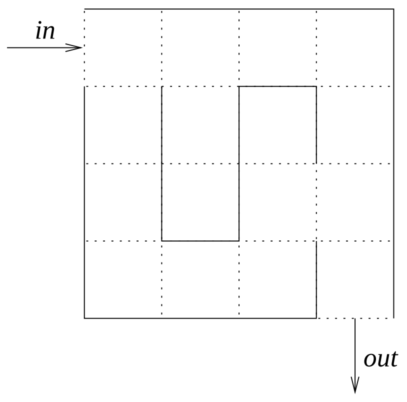


**Fig. 1.11** A maze offering choices

# 2 Concurrency

When two processes are executed concurrently, they constitute a parallel combination. Each process executes independently, in accordance with its prescribed patterns of behaviour, but its range of possibilities will be influenced by the other process.

The way parallel CSP processes interact is intimately bound up with the view of events as synchronizations. Any event which appears in the interface of both processes must involve both processes whenever it occurs. This is the mechanism by which parallel processes interact—by *synchronizing* on events in the interface between them. Synchronization is symmetric and instantaneous, and occurs only when both participants engage in it simultaneously, much like a handshake. For this reason it is often known as *handshake synchronization*. A single event occurs—the handshake—with a number of participants. Examples of handshake synchronizations include: the passing of a baton in a relay race; the delivery of a registered letter; the closure of a contract; becoming married; inserting a coin in a vending machine; starting a phone conversation; sending an email message; adding a job to a printer queue.

## 2.1 ALPHABETIZED PARALLEL

A process description includes a dynamic part, captured using the process description language of CSP, which describes the possible patterns of events or synchronizations. When processes are put together in parallel, it is also necessary to specify how they will interact. One way of achieving this is by providing an explicit description of the interface or alphabet of each process: its static specification.

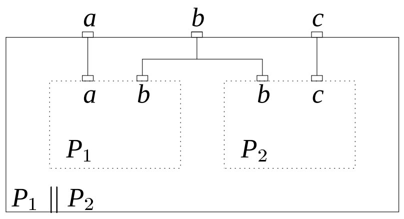


***Fig. 2.1*** Synchronization with buttons

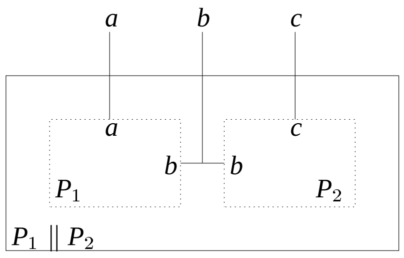


***Fig. 2.2*** Synchronization with wires

The interface of a process is the set of all the events that the process has the potential to engage in. If a process is considered as a black box, the interface consists of the names of all its buttons. If the process is viewed as a component with wires, then the interface provides a list of all the wires. In either case, it describes all of the events that the process is potentially able to perform. Since all external events are synchronizations between processes and their environment, the interface is the point at which a process and its environment influence each other.

A parallel combination of two processes $P_1$ and $P_2$ whose interfaces are given as $A \subseteq \Sigma$ and $B \subseteq \Sigma$ respectively is described as

$$P_1 \;{}_A\|_B\; P_2$$

(pronounced ‘P1 parallel on A, B P2’). In this combination $P_1$ can perform only events in $A$, $P_2$ can perform only events in $B$, and they must simultaneously engage in events in the intersection of $A$ and $B$. This is illustrated by Figures 2.1 and 2.2. It is conventional for the interface $A$ of process $P_1$ to contain at least all of the events used in the definition of $P_1$. Similarly, $B$ should contain all of the events appearing in $P_2$.

Parallel components of the combination $P_1 \;{}_A\|_B\; P_2$ must also agree on termination, even though the ✓ event does not appear explicitly in the interface sets $A$ and $B$. This means that a parallel combination does not terminate until all of its components are terminated.

There are two rules that define the possible transitions of a parallel combination. One rule describes the independent execution of each of the components, and the other describes the performance of a joint step. The set $A^{\checkmark}$ is defined to be $A \cup \{\checkmark\}$.

$$\frac{P_1 \xrightarrow{\mu} P_1'}{\begin{array}{l} P_1 \;{}_A\|_B\; P_2 \xrightarrow{\mu} P_1' \;{}_A\|_B\; P_2 \\ P_2 \;{}_B\|_A\; P_1 \xrightarrow{\mu} P_2 \;{}_B\|_A\; P_1' \end{array}} \quad [\, \mu \in (A \cup \{\tau\} \setminus B) \,]$$

$$\frac{\begin{array}{l} P_1 \xrightarrow{a} P_1' \\ P_2 \xrightarrow{a} P_2' \end{array}}{P_1 \;{}_A\|_B\; P_2 \xrightarrow{a} P_1' \;{}_A\|_B\; P_2'} \quad [\, a \in A^{\checkmark} \cap B^{\checkmark} \,]$$

The first transition rule states that each side can execute independently through performing events which are not in the common interface. The second transition rule states that if both components can perform an event $a$ which appears in each of their interfaces, then the parallel combination can also perform it.

These rules also capture the fact that the interfaces of the components do not change as the processes execute: they remain fixed throughout the life of the parallel combination.

EXAMPLE 2.1 The parallel combination

$$(a \to P_1) \;{}_{\{a\}}\|_{\{a,b\}}\; (a \to P_2 \mid b \to P_3)$$

can perform an $a$ transition initially, and reach $P_1 \;{}_{\{a\}}\|_{\{a,b\}}\; P_2$, or else it can perform a $b$ transition initially to reach $(a \to P_1) \;{}_{\{a\}}\|_{\{a,b\}}\; P_3$.

However, the same processes combined in a different way

$$(a \to P_1) \;{}_{\{a,b\}}\|_{\{a,b\}}\; (a \to P_2 \mid b \to P_3)$$

can perform only $a$ initially, since the presence of $b$ in both interfaces means that both components are required to co-operate on its occurrence; and the left-hand process is not able initially to perform $b$. □

EXAMPLE 2.2 A race between two competitors should have a single *start* event which both of the participants synchronize on. However, each competitor will independently finish, so two events *finish*$_1$ and *finish*$_2$ are used to describe these separate events. Participant 1 will be described as $start \rightarrow finish_1 \rightarrow STOP$, and participant 2 similarly. The race can then be described as

$$\begin{array}{l} RACE = \\ \quad start \rightarrow finish_1 \rightarrow STOP \;{}_{\{start,finish_1\}}\|_{\{start,finish_2\}}\; start \rightarrow finish_2 \rightarrow STOP \end{array}$$

The fact that each *finish* event occurs in only one interface set captures the fact that the competitors can finish independently of each other. □

EXAMPLE 2.3 On entering a restaurant, the cloakroom attendant might help the customer off or on with her coat, as captured by the events *coat.off* and *coat.on* respectively, storing and retrieving coats as appropriate. This activity might be described by the following process description:

$$\begin{array}{lcl} ATT & = & coat.off \rightarrow store \rightarrow ATT \\ & & \Box \\ & & retrieve \rightarrow coat.on \rightarrow ATT \end{array}$$

with an interface described as

$$\alpha_{ATT} \quad = \quad \{coat.off, coat.on, store, retrieve\}$$

The dining behaviour of the customer is as follows:

$$\begin{array}{lcl} CUST & = & enter \rightarrow coat.off \rightarrow eat \rightarrow coat.on \rightarrow CUST \\ \alpha_{CUST} & = & \{coat.off, coat.on, enter, eat\} \end{array}$$

In the parallel combination

$$ATT \;{}_{\alpha_{ATT}}\|_{\alpha_{CUST}}\; CUST$$

the events *enter* and *eat* are performed solely by *CUST*; the events *coat.on* and *coat.off* are synchronizations between *CUST* and *ATT*; and the events *store* and *retrieve* are entirely under the control of *ATT*—the attendant is left to deal with the coat appropriately. □

EXAMPLE 2.4 A pay and display parking permit machine accepts cash, and issues tickets and change. A designer might decide to implement these functions using two separate processes:

$$\begin{array}{rcl} TICKET & = & cash \rightarrow ticket \rightarrow TICKET \\ CHANGE & = & cash \rightarrow change \rightarrow CHANGE \end{array}$$

The machine is then captured as the parallel combination of these two components:

$$MACHINE \quad = \quad TICKET \;{}_{\{cash,ticket\}}\|_{\{cash,change\}}\; CHANGE$$

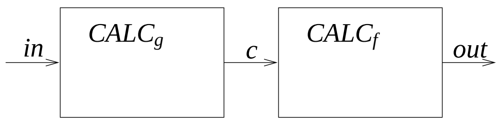


***Fig. 2.3*** A pipeline of two calculations

The two components both participate in the *cash* event, but they exercise independent control over the *ticket* and *change* events. □

EXAMPLE 2.5 A process to calculate and output $f(g(x))$ from input $x$ may be constructed from two processes which calculate $f$ and $g$.

$$\begin{aligned} CALC_g &= in?x : \mathbb{Z} \rightarrow c!g(x) \rightarrow CALC_g \\ CALC_f &= c?y : \mathbb{Z} \rightarrow out!f(y) \rightarrow CALC_f \end{aligned}$$

The composition of $f$ and $g$ may be calculated by the parallel combination of $CALC_g$ and $CALC_f$. The output of $CALC_g$ is connected to the input of $CALC_f$, resulting in a pipeline of two processes which together compute $f(g(x))$ from input $x$. This is described by

$$CALC_g \;{}_{in.\mathbb{Z} \cup c.\mathbb{Z}}\|_{c.\mathbb{Z} \cup out.\mathbb{Z}}\; CALC_f$$

which is illustrated in Figure 2.3. □

A parallel combination of processes is itself a process, with an alphabet and a range of possible executions. Synchronization on an event $a$ by two concurrent components results in a single occurrence of that event, and there can be no information contained in the occurrence of $a$ as to the number of participants. A further process may be run in parallel with the existing combination, also synchronizing on $a$. This approach to parallel composition results in a mechanism for multi-way synchronization. No matter how many parallel components a system contains, all those with an event $a$ in their alphabet are required to participate in every occurrence of it.

EXAMPLE 2.6 A team of furniture removers contains a number of people with responsibilities for moving different kinds of furniture. Pianos require several people to lift them, so the event *lift_piano* will be in the alphabet of several members of the team, and they must all synchronize on this event for it to occur. The event *lift_piano* will occur only once, no matter how many participants it has. This event is a multi-way synchronization—it can occur only when all its participants are ready. There is also the possibility of adding further participants by extending the system: if a new person is added to the team with piano-lifting among their responsibilities, then they will also participate in the event *lift_piano*. □

A parallel component of a system constrains the occurrence of all events in its alphabet, since the co-operation of that component is required for the performance of such events. The event *lift_piano* can be prevented from occurring by any team member who is not ready to perform it. This means that any constraint on the occurrence of events can be introduced through the addition of a parallel component which enforces that constraint. Introduction of constraints may be carried out at the specification level, where different facets of desired behaviour are captured by different process descriptions which are then combined in parallel. It may also be carried out at the design level, where responsibility for enforcing different constraints is assigned to different processes.

EXAMPLE 2.7 A number of shopping opportunities are described by the process *SHOPPING*. An item may be selected by the customer; the customer might pay; or the customer might leave, in which case the other possibilities are lost until the shop is re-entered.

$$\begin{aligned} \mathit{SHOPPING} &= \mathit{select} \rightarrow \mathit{SHOPPING} \\ &\quad \Box\ \mathit{pay} \rightarrow \mathit{SHOPPING} \\ &\quad \Box\ \mathit{leave} \rightarrow \mathit{enter} \rightarrow \mathit{SHOPPING} \end{aligned}$$

There are a number of restrictions that the shop places on the free performance of events as described by *SHOPPING*. One is concerned with ensuring that goods are paid for; and another requires goods to be selected before payment.

Security ensures that selected goods are paid for by the time the customer leaves the shop:

$$\begin{aligned} \mathit{SECURITY} &= \mathit{select} \rightarrow \mathit{WATCH} \\ &\quad \Box\ \mathit{pay} \rightarrow \mathit{SECURITY} \\ &\quad \Box\ \mathit{leave} \rightarrow \mathit{SECURITY} \\[1ex] \mathit{WATCH} &= \mathit{select} \rightarrow \mathit{WATCH} \\ &\quad \Box\ \mathit{pay} \rightarrow \mathit{SECURITY} \end{aligned}$$

This process is concerned with tracking the occurrence of events *select* and *pay* and restricting the possibility of the event *leave* under the appropriate circumstances. The event *enter* is not part of its alphabet.

The shop operating under this constraint is described by

$$\begin{aligned} &\mathit{SECURE\_SHOP} = \\ &\quad (\mathit{enter} \rightarrow \mathit{SHOPPING})\ {}_{\{enter,leave,select,pay\}}\|_{\{leave,select,pay\}}\ \mathit{SECURITY} \end{aligned}$$

The cash tills impose another constraint: that payment is possible only after some item has been selected. The initial situation is captured by *BROWSING*, whose alphabet is

$\{pay, select\}$. Before any item has been selected, payment is not possible. Once an item has been selected, then payment is possible, though further items may also be selected.

$$\begin{aligned}
BROWSING &= select \rightarrow TILL \\
TILL &= select \rightarrow TILL \\
&\quad \Box\ pay \rightarrow BROWSING
\end{aligned}$$

The alphabet of this process does not contain either *enter* or *leave*, since it is concerned only with the relationship between the events *select* and *pay*.

The shop as a whole, integrating this last constraint, is described by

$$SHOP \;=\; SECURE\_SHOP \;{}_{\{enter,leave,select,pay\}}\|_{\{select,pay\}}\; BROWSING$$

Each parallel component restricts the behaviour of the whole to some degree. The process $enter \rightarrow SHOPPING$ restricts all occurrences of *select* and *pay* to between *enter* and *leave*; the process *SECURITY* restricts occurrences of *leave*; and the process *BROWSING* restricts occurrences of *pay*. All three processes are concerned with the events *pay* and *select*, and so all three processes participate in those events—each occurrence is a multi-way synchronization. Since these events are all part of the alphabet of *SHOP*, and hence available for further synchronization, they may be further constrained by additional parallel components. □

The default interface for a process *P* is its alphabet $\alpha(P)$: those events mentioned in the process description. If the required interface in a process description is precisely this set, then it need not be mentioned explicitly. In the case of $P_1 \;{}_{\alpha(P_1)}\|_{\alpha(P_2)}\; P_2$ the alphabets may be dropped and $P_1 \parallel P_2$ written instead. The process *SHOP* described above could be rewritten as

$$SHOP \;=\; ((enter \rightarrow SHOPPING) \parallel SECURITY) \parallel BROWSING$$

since the interface sets given in for each component process are precisely their alphabets. The definition of the alphabet operator $\alpha(P)$ is given in Figures 3.7 and 3.8 on Pages 75 and 75.

## Deadlock

The introduction of concurrency brings with it the possibility of deadlock, which is not possible for purely sequential programs. In a concurrent system the execution of one process might inhibit or temporarily suspend the execution of another. There are a number of situations in which this might arise, such as resource sharing, or awaiting communication. It is thus possible that every process in a concurrent system is waiting for some other process. Since no process is actively executing, each process will remain blocked, forever waiting for some other process to make progress. This unfortunate phenomenon is called *deadlock*: each process individually would be able to continue execution if it were in a different environment, but all are prevented from doing so. Incompatible states between parallel components often arise as a result of unforeseen and unexpected sequences of interactions.

The combined state of all the deadlocked components is known as the *deadlock state*. From the point of view of an execution, a sequence of transitions has resulted in a deadlocked process if the final process description of the sequence has no possible transitions. In this sense, the process *STOP* can be considered as a deadlocked process: no further progress can be made, although the execution has not completed normally.

EXAMPLE 2.8 Two children share an paint box and an easel. Whenever they wish to paint, they first search for the easel and the box until both are found. After they have finished painting, they drop the box and then the easel.

$$\begin{aligned} ISABELLA \;=\;& isabella.get.box \to isabella.get.easel \to isabella.paint \to \\ & \quad isabella.drop.box \to isabella.drop.easel \to ISABELLA \\ & \Box\; isabella.get.easel \to isabella.get.box \to isabella.paint \to \\ & \quad isabella.drop.box \to isabella.drop.easel \to ISABELLA \end{aligned}$$

$$\begin{aligned} KATE \;=\;& kate.get.box \to kate.get.easel \to kate.paint \to \\ & \quad kate.drop.box \to kate.drop.easel \to KATE \\ & \Box\; kate.get.easel \to kate.get.box \to kate.paint \to \\ & \quad kate.drop.box \to kate.drop.easel \to KATE \end{aligned}$$

The easel and the box can each be held only by one child at a time:

$$\begin{aligned} EASEL \;=\;& isabella.get.easel \to isabella.drop.easel \to EASEL \\ & \Box \\ & kate.get.easel \to kate.drop.easel \to EASEL \end{aligned}$$

$$\begin{aligned} BOX \;=\;& isabella.get.box \to isabella.drop.box \to BOX \\ & \Box \\ & kate.get.box \to kate.drop.box \to BOX \end{aligned}$$

The combination of the children and the painting equipment is described as

$$PAINTING \;=\; ISABELLA \parallel KATE \parallel BOX \parallel EASEL$$

The arrangement works well on the whole, but occasionally both children decide at about the same time to paint. If this happens, then there is a danger that one of them will find the easel, and the other will find the box, after which neither of them can make any further progress (see Exercise 2.3). The two items could be released, but their owners are not ready to release them:

they are each waiting for the other item to become available first, a classic deadlock situation. In this case the system could be made deadlock-free by insisting that the items are acquired in a particular order: the box first, and then the easel. Restricting the possibilities on the children, to only the first branch of the choice in each case, results in an overall improvement to the system. $\Box$

EXAMPLE 2.9 A team of two furniture movers is required to move a number of pianos and tables, and each piece of furniture requires two people to lift it. Each remover independently makes his own decision as to which piece of furniture to lift first—this is an internal choice, since in each case it is a decision made by the furniture remover.

$$\begin{aligned} PETE &= lift\_piano \to PETE \\ &\quad \sqcap\ lift\_table \to PETE \\[1ex] DAVE &= lift\_piano \to DAVE \\ &\quad \sqcap\ lift\_table \to DAVE \\[1ex] TEAM &= DAVE \parallel PETE \end{aligned}$$

If each remover makes the same choice, then they are able to co-operate and synchronize on lifting the item. However, if their choices differ, then the result is deadlock.

*DAVE* comprises part of *PETE*'s environment, and so both *PETE* and his environment have control over the way the choice is resolved. Since it is not possible for two independent parties both to exercise complete and independent control over resolution of a choice, the result might be deadlock.

If *PETE* instead offered an external choice

$$\begin{aligned} PETE' &= lift\_piano \to PETE' \\ &\quad \Box\ lift\_table \to PETE' \end{aligned}$$

then he gives up control over the choice and instead is prepared to go along with the decision of the environment. Since there is now only one agent making the choice, the resulting team $PETE' \parallel DAVE$ will no longer deadlock. $\Box$

## Indexed alphabetized parallel

The binary parallel composition operator may be generalized to model situations where there are a number of concurrent interacting components. If $I$ is a finite set of indexes such that $P_i$ and $A_i$ are defined for each $i \in I$, then the following system may be defined:

$$\parallel^{i \in I}_{A_i} P_i$$

This describes a combination of components where each $P_i$ has interface $A_i \subseteq \Sigma$. Any event $a$ must be performed in a multi-way synchronization by all those processes $P_i$ which have $a \in A_i$: all parties interested in a particular event must be involved in all of its occurrences.

The indexed parallel composition may be defined using successive applications of the binary parallel operator. If $I$ contains only two indexes $i_1$ and $i_2$, then the indexed parallel is equivalent to the binary form:

$$\|_{A_i}^{i \in \{i_1, i_2\}} P_i \quad = \quad P_{i_1} \;{}_{A_{i_1}}\|_{A_{i_2}}\; P_{i_2}$$

An inductive definition is provided for the case where the set $I$ has more than two elements, defining the case with $n+1$ elements in terms of the definition for $n$ elements. The interface of $\|_{A_i}^{i \in I} P_i$ is the union of all the individual interfaces: $\bigcup_{i \in I} A_i$. The addition of one more component $P_j$ ($j \notin I$) with interface $A_j$ results in the parallel composition $\|_{A_i}^{i \in I \cup \{j\}} P_i$.

$$\|_{A_i}^{i \in I \cup \{j\}} P_i \quad = \quad (\|_{A_i}^{i \in I} P_i) \;{}_{\cup_{i \in I} A_i}\|_{A_j}\; P_j$$

It transpires that the binary parallel operator is associative, in the sense that the combination $P_1 \;{}_A\|_{B \cup C}\; (P_2 \;{}_B\|_C\; P_3)$ has the same executions as $(P_1 \;{}_A\|_B\; P_2) \;{}_{A \cup B}\|_C\; P_3$, and commutative: $P_1 \;{}_A\|_B\; P_2$ has the same executions as $P_2 \;{}_B\|_A\; P_1$. These two facts mean that the same process will result whichever order the processes are added to the parallel combination. As in the binary case, the interface sets $A_i$ may be dropped from the process description, in which case the alphabet of each process is taken as its default interface. The resulting combination is written $\|^{i \in I} P_i$.

EXAMPLE 2.10 A group of people are all required to be present for a meeting to be quorate. If *NAMES* is the set of all the people's names, then the alphabet and behaviour of a particular person may be described as follows:

$$\begin{array}{rcl} A_n & = & \{enter.n, leave.n, meeting\} \\[1em] PERSON_n & = & enter.n \rightarrow PRESENT_n \\[0.5em] PRESENT_n & = & leave.n \rightarrow PERSON_n \\ & & \Box\; meeting \rightarrow PRESENT_n \end{array}$$

The situation is then described as

$$GROUP \quad = \quad \|_{A_n}^{n \in NAMES} PERSON_n$$

The event *meeting* is in the alphabet of all the components, so it can occur when they are all able to perform it. This can only be when *enter.n* has occurred for all $n \in NAMES$ without a

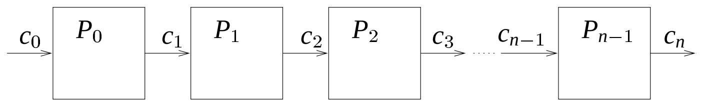


***Fig. 2.4*** A chain of buffers

subsequent *leave*.*n*. All the events of the form *enter*.*n* and *leave*.*n* appear in the alphabet of only one process, $PERSON_n$, so their occurrence is entirely under the control of that process. □

EXAMPLE 2.11 A chain of processes which act simply as buffers, passing on an item of data, is given in Figure 2.4. There are $n$ such components, indexed by the set $\{0, \ldots, n-1\}$.

For every $i$ between $0$ and $n-1$, the buffer $P_i$ and its interface $A_i$ are given by the following definition,

$$\begin{aligned} A_i &= c_i.\mathbb{Z} \cup c_{i+1}.\mathbb{Z} \\ P_i &= c_i?x : \mathbb{Z} \to c_{i+1}!x \to P_i \end{aligned}$$

The chain of buffers is then defined as

$$CHAIN = \Big\|_{A_i}^{i \in \{0 \ldots n-1\}} P_i$$

Each occurrence $c_i.m$ of a message passing along the chain involves exactly two participants, $P_{i-1}$ and $P_i$, apart from the two end channels $c_0$ and $c_n$ which in each case involve only one component. □

EXAMPLE 2.12 A network which sorts a set of $n+1$ numbers into ascending order may be constructed from cells each of which sort two numbers at a time, as in Figure 2.5. Here the set of numbers is input along $v_{0,0}$ and the $h_{0,i}$ for $0 \leqslant i < n$. The numbers appear in ascending order along the $v_{j,n+1} (0 \leqslant j \leqslant n)$.

Each cell $C_{i,j}$ inputs a number on its left and a number from above, outputs the smaller of these below, and the larger to its right. For internal cells, these are inputs to neighbouring cells; and for the cells in the bottom row, they are the outputs of the array.

For $i < j$ the $C_{i,j}$ are defined as follows:

$$C_{i,j} = h_{i,j}?x \to v_{i,j}?y \to v_{i,j+1}!\min\{x,y\} \to h_{i+1,j}!\max\{x,y\} \to C_{i,j}$$

Cells on the diagonal of the array are defined slightly differently, reflecting their different connections:

$$C_{i,i} = h_{i,i}?x \to v_{i,i}?y \to v_{i,i+1}!\min\{x,y\} \to v_{i+1,i+1}!\max\{x,y\} \to C_{i,i}$$

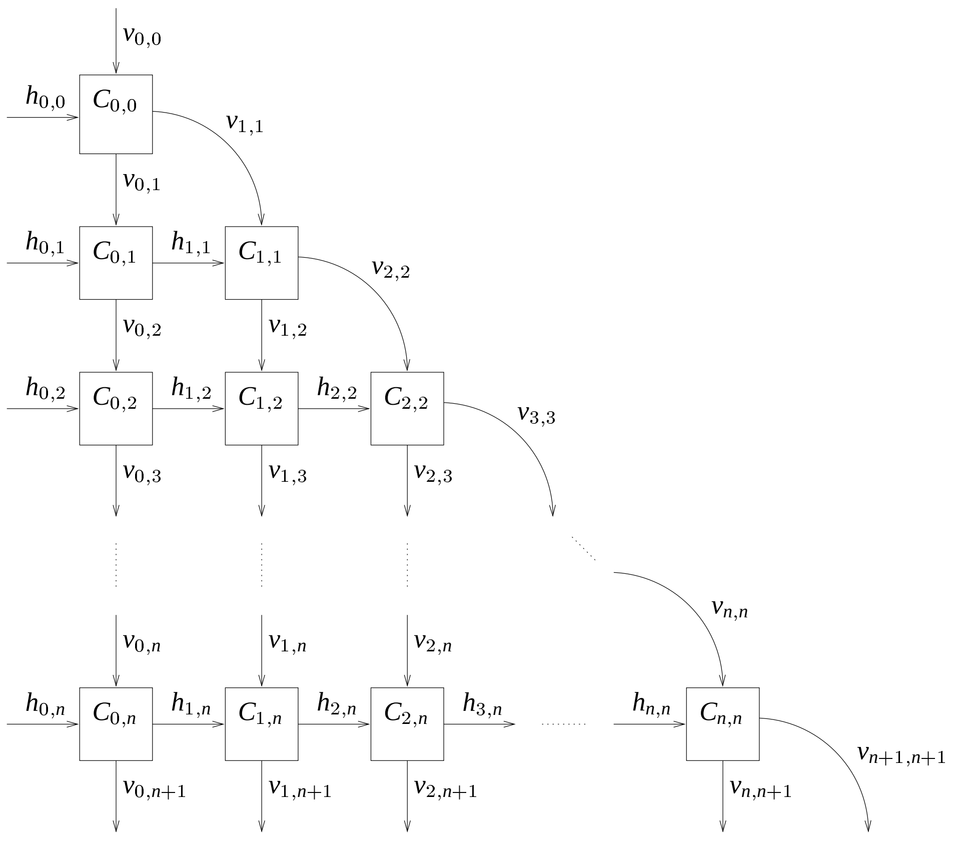


***Fig. 2.5*** An array for sorting

The network for sorting may then be defined using indexed parallel composition:

$$SORTER \quad = \quad \|^{0 \leqslant i \leqslant j \leqslant n} C_{i,j}$$

The indexing set is given implicitly: the predicate $0 \leqslant i \leqslant j \leqslant n$ is shorthand for $(i,j) \in \{(k,l) \mid 0 \leqslant k \leqslant l \leqslant n\}$. □

EXAMPLE 2.13 A collection of $2^n$ nodes in a network can be efficiently connected for the purposes of routing messages by assigning each node to the corner of a hypercube. Its coordinates will be an $n$-tuple, with each value of the tuple either 0 or 1. The set of possible coordinates is given by $COORD = \{0,1\}^n$.

To specify the connections between nodes, it is necessary to decide which pairs of nodes should be adjacent. These will be those pairs whose coordinates differ in exactly one place.

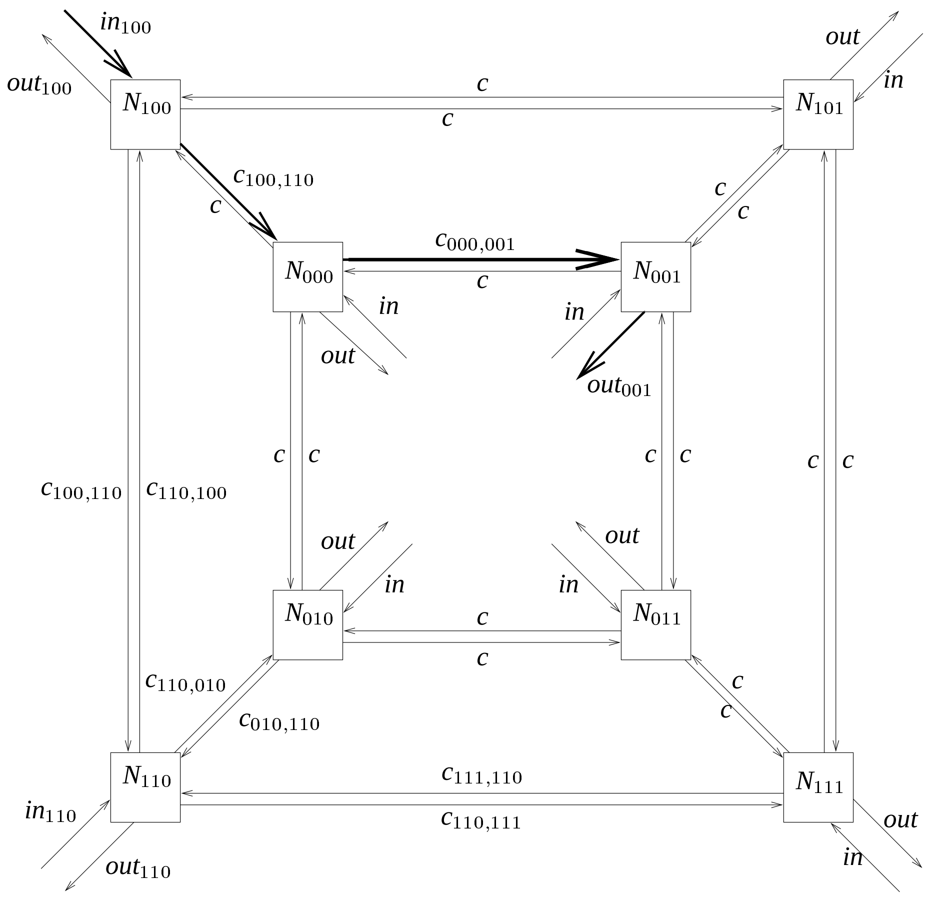


***Fig. 2.6*** Message routing through a hypercube of $2^3$ nodes; a route from 100 to 001 emphasized

The set of coordinates adjacent to $l$ is given by $adj(l)$:

$$adj(l) \quad = \quad \{k : COORD \mid \exists\, i \bullet (k(i) \neq l(i) \wedge \forall\, j \neq i \bullet k(j) = l(j))\}$$

This network is pictured in Figure 2.6. The notation $l(i)$ denotes the $i$th bit of coordinate $l$. For example, if $l = 001$ then $l(0) = 0$, $l(1) = 0$, and $l(2) = 1$.

In order to route messages, it is sensible to send a message to a node closer to its destination.

$$\begin{aligned} & next(l, d) \\ & \quad = \quad \{k : COORD \mid \exists\, i \bullet (k(i) \neq l(i) \wedge k(i) = d(i) \wedge \forall\, j \neq i \bullet k(j) = l(j))\} \end{aligned}$$

The set $next(l, d)$ is the set of those coordinates which are adjacent to $l$ and which differ from $d$ on fewer bits than $l$ does: the neighbours of $l$ which are closer to the destination $d$.

A node $N_l$ will have channels to and from all of its neighbours: for each $k$, $l$, the channel $c_{k,l}$ carries messages from $N_k$ to $N_l$. It also has channels $in_l$ and $out_l$ for input and output outside the network. The interface of $N_l$ will be

$$A_l \quad = \quad \{in_l, out_l\} \cup \{c_{k,l} \mid k \in adj(l)\} \cup \{c_{l,k} \mid k \in adj(l)\}$$

The cell $N_l$ itself is always prepared to accept messages (unless it is waiting to output), and it maintains a list of all the messages it has outstanding. When this list is nonempty, $N_l$ is also ready to send the oldest message to the next node in the network.

$$\begin{array}{rcll}
N_l(\langle\rangle) & = & \Box_{k \in adj(l)}\, c_{k,l}?(d.m) \rightarrow N_l(\langle(d,m)\rangle) & \\
& = & \Box\; in_l?(d.m) \rightarrow N_l(\langle(d,m)\rangle) & \\
N_l(\langle(d,m)\rangle \frown s) & = & out_l!m \rightarrow N_l(s) & \text{if } d = l \\
& & (\Box_{k \in next(l,d)}\, c_{l,k}!(d.m) \rightarrow N_l(s)) & \text{otherwise} \\
& & \Box & \\
& & \Box_{k \in adj(l)}\quad c_{k,l}?(d'.m') \rightarrow & \\
& & \qquad N_l(\langle(d,m)\rangle \frown s \frown \langle(d',m')\rangle) &
\end{array}$$

Finally, the network may be described using indexed parallel composition:

$$MAILER \quad = \quad \|_{A_l}^{l \in COORD}\, N_l(\langle\rangle)$$

Each node is initially empty. □

## 2.2 INTERLEAVING

Synchronization on common events provides one form of parallel execution of processes. Concurrent execution of processes $P_1$ and $P_2$ where no such synchronization is required is described by the combination

$$P_1 \;|||\; P_2$$

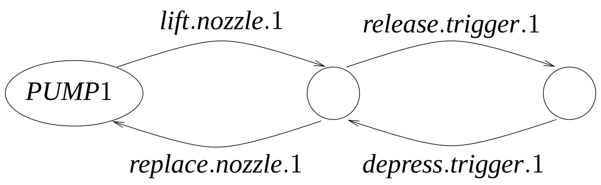


**Fig. 2.7** Transitions for *PUMP*1

pronounced '$P_1$ interleave $P_2$'. In this process, the components $P_1$ and $P_2$ execute completely independently of each other, and do not interact on any events apart from termination. Each event is performed by exactly one process. The operational semantics rules are straightforward:

$$\boxed{\frac{P_1 \xrightarrow{\mu} P_1'}{\begin{array}{l} P_1 \mid\mid\mid P_2 \xrightarrow{\mu} P_1' \mid\mid\mid P_2 \\ P_2 \mid\mid\mid P_1 \xrightarrow{\mu} P_2 \mid\mid\mid P_1' \end{array}} \; [\,\mu \neq \checkmark\,] \qquad \frac{P_1 \xrightarrow{\checkmark} P_1' \quad P_2 \xrightarrow{\checkmark} P_2'}{P_1 \mid\mid\mid P_2 \xrightarrow{\checkmark} P_1' \mid\mid\mid P_2'}}$$

Unlike synchronous parallel, there is no event that both $P_1$ and $P_2$ engage in simultaneously (except termination).

EXAMPLE 2.14 A garage has two petrol pumps: *PUMP*1 and *PUMP*2. The petrol supply service offered by the garage is described as

$$FORECOURT \;=\; PUMP1 \mid\mid\mid PUMP2$$

A customer will always interact with only one pump on any transaction; and the pumps operate independently. The description of *PUMP*1 is a sequential process, defined by a mutual recursion.

$$\begin{array}{rcl} PUMP1 & = & lift.nozzle.1 \rightarrow READY1 \\ READY1 & = & replace.nozzle.1 \rightarrow PUMP1 \\ & & \Box \\ & & depress.trigger.1 \rightarrow release.trigger.1 \rightarrow READY1 \end{array}$$

The state graph corresponding to *PUMP*1 is given in Figure 2.7.

The description of *PUMP*2 is entirely similar:

$$\begin{array}{rcl} PUMP2 & = & lift.nozzle.2 \rightarrow READY2 \\ READY2 & = & replace.nozzle.2 \rightarrow PUMP2 \\ & & \Box \\ & & depress.trigger.2 \rightarrow release.trigger.2 \rightarrow READY2 \end{array}$$

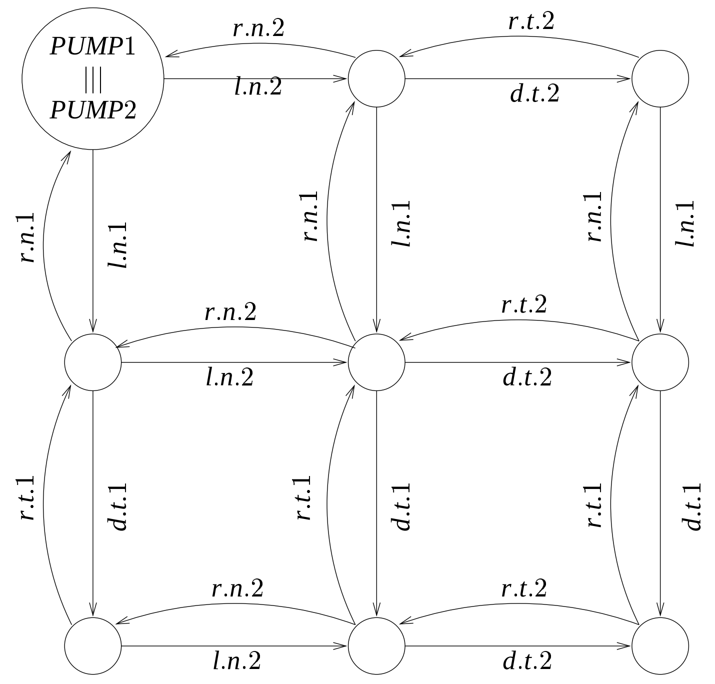


***Fig. 2.8*** Transitions for $PUMP1 \mid\mid\mid PUMP2$

The executions of the two pumps are unrelated. The process *PUMP*1 will remain ready to engage in the event *lift*.*nozzle*.1 until it occurs, independently of the progress that *PUMP*2 makes. The state transitions for the combination of processes are given in Figure 2.8. □

When two processes $P_1$ and $P_2$ have alphabets $\alpha(P_1)$ and $\alpha(P_2)$ that do not intersect, then their parallel combination $P_1 \;{}_{\alpha(P_1)}\|_{\alpha(P_2)}\; P_2$ will behave the same as their interleaved combination $P_1 \mid\mid\mid P_2$. Since they do not have any events in common to interact on, execution of each component process will be independent in both cases. Given the definitions of *PUMP*1 and *PUMP*2 an alternative definition of *FORECOURT* would be $PUMP1 \parallel PUMP2$. However, for the purposes of design, use of the $\mid\mid\mid$ operator describes the design decision to provide the *FORECOURT* service as an interleaving of two processes, before those processes are described any further.

Interleaved processes do not synchronize on events even when their alphabets do overlap. In the case where both components are able to perform the same event, only one of the processes will actually engage in any particular occurrence of that event. In such a case, the environment which is interacting with the interleaved combination has no control or influence over which of the two processes actually performs it.

EXAMPLE 2.15 A company that offers a telephone ordering service will operate a number of phone lines.

$$LINE_n \quad = \quad ring \rightarrow connect.n \rightarrow order \rightarrow disconnect \rightarrow LINE_n$$

A line operated by person $n$ will repeatedly take calls as follows: the phone will firstly ring, then a connection will be made to $n$, then the order takes place, and finally the call is disconnected, and $n$ is ready to receive the next order.

The service employs Chris and Sandy to run one line each, both connected to the same phone number, modelled by the fact that they each have the same event *ring* in their alphabet. The service is described by

$$SERVICE \quad = \quad LINE_{chris} \mid\mid\mid LINE_{sandy}$$

In this situation, only one of the phones will ring when a customer dials in. The customer has no control over which, and hence has no control over whether the next event in the call will be $connect.chris$ or $connect.sandy$. A customer who wishes to talk only to Sandy may be disappointed.

There is an internal choice between the two components over which of them performs the event *ring*. The choice of which of the *LINE* processes takes the call is made internally by *SERVICE*. It cannot be made externally, by the customer, since initially there is only *ring* on offer; a choice cannot be resolved externally if there is only one event to choose. □

The interleaving operator is appropriate for describing a number of identical resources which are all available for use.

EXAMPLE 2.16 A fax machine may be described as

$$FAX \quad = \quad accept?d : DOCUMENT \rightarrow print!d \rightarrow FAX$$

The machine is initially ready to accept any document. After accepting a document $d$, it prints $d$ and reverts to its initial state.

A collection of four fax machines may be connected to the same phone number (with four lines): any of them is suitable for processing incoming faxes.

$$FAXES \quad = \quad (FAX \mid\mid\mid FAX) \mid\mid\mid (FAX \mid\mid\mid FAX)$$

This system provides the facility for processing up to four incoming faxes at the same time. It can accept up to four faxes before printing. An incoming fax has no influence over which machine is actually chosen to process it, but in this case this makes no difference since all choices have exactly the same behaviour. □

## Dynamic process creation

Dynamic process creation can be modelled by use of an interleaving construction within a recursive loop. In general, a fresh copy of the entire process description is generated every time a recursive invocation occurs. Dynamic process creation occurs when such invocations take place while the parent process continues to execute. This may be set up in a recursive definition which contains a number of paths, some of which concern execution of the parent process, and where others contain recursive calls.

EXAMPLE 2.17 A mail forwarder receives messages on channel *in* and forwards them along channel *out* at some later stage. It should always be willing to accept messages.

One approach to designing such a process would be to allow it to accept a message, and then create two processes: one to take responsibility for sending the message on, and the other to be a fresh copy of the original. This approach is described in CSP as follows:

$$NODE \quad = \quad in?x \rightarrow (NODE \mid\mid\mid OUT(x))$$

The process $OUT(x)$ describes the part of the process which deals with sending $x$ along channel *out*.

There are two views with respect to dynamic process creation. The parent process might be considered as the process $N = in?x \rightarrow N$ which is responsible for accepting inputs; and a child output process is generated every time an input occurs. Alternatively, the parent process might be considered as $in?x \rightarrow OUT(x)$, where a fresh version of the entire process, ready to handle the next input, is generated every time an input occurs. Both of these viewpoints is consistent with the description above.

This structure is given before the process $OUT(x)$ is defined, and there are different possibilities for this process definition.

The expectation may be that the thread described by $OUT(x)$ should finish after $x$ has been output. This would be described by

$$OUT(x) \quad = \quad out!x \rightarrow SKIP$$

After the output has occurred, then this thread of execution will finish—there is nothing further that it is required to do. All further activity of the process will come from other parallel components. Observe that any process $P \mid\mid\mid SKIP$ has the same executions as $P$, so a garbage collector could remove component processes that have finished without affecting the execution.

An alternative approach to defining the spawned thread might allow it to continue with a fresh version of *NODE* after its initial output. This possibility is described by

$$OUT(x) \quad = \quad out!x \rightarrow NODE$$

so that after one input and its corresponding output the result is two interleaved versions of *NODE*. The number of active threads of control in this process will grow without limit, with a fresh thread generated on every input, and no thread ever finishing. Despite this proliferation, this implementation has no more executions than the previous tidier definition.

A different requirement might be that $x$ should be logged as well as output. This could be achieved sequentially as $out!x \to log!x \to SKIP$, or alternatively by two interleaved threads $out!x \to SKIP \mid\mid\mid log!x \to SKIP$. $\Box$

EXAMPLE 2.18 A book shop operates a system whereby customers pay for books at a cashier's counter and collect them at a different counter where they had previously been lodged. The operation of the book counter may be described as follows:

$$\begin{aligned} BOOK \quad = \quad & lodge \to issue\_chit \to BOOK \\ & \Box \\ & receive\_receipt \to claim \to BOOK \end{aligned}$$

A customer may *lodge* a chosen book with the counter, and have a chit issued in return. In order to *claim* the book to take away, a receipt must be provided. This may be obtained from the cashier, described as follows:

$$CASHIER \quad = \quad receive\_chit \to payment \to issue\_receipt \to CASHIER$$

Book chits must be issued by the book counter before they can be received by the cashier:

$$CHIT \quad = \quad issue\_chit \to (CHIT \mid\mid\mid receive\_chit \to SKIP)$$

Similarly, receipts must be issued before they can be exchanged for books:

$$RECEIPT \quad = \quad issue\_receipt \to (RECEIPT \mid\mid\mid receive\_receipt \to SKIP)$$

Each of these components imposes some constraint on the customer; and they are all in place together. The complete payment system which the customer must navigate is captured as the parallel combination

$$CASHIER \parallel BOOK \parallel CHIT \parallel RECEIPT$$

The first two processes represent the parts of the system which the customer interacts with. The other two processes describe the relevant properties of the objects used by the customer and by the book shop: that chits and receipts can be created only by the book counter and the cashier respectively. The book shop's payment system relies on the fact that chits and receipts cannot be forged. $\Box$

### Indexed interleaving

The interleaving parallel operator is associative, in the sense that $P_1 ||| (P_2 ||| P_3)$ has exactly the same execution possibilities as $(P_1 ||| P_2) ||| P_3$; and commutative, in the sense that $P_1 ||| P_2$ and $P_2 ||| P_1$ have the same execution possibilities. It can therefore be generalized to finite combinations of processes. The generalization takes the form

$$|||_{i \in I} P_i$$

where $I$ is a finite set of indexes, and $P_i$ is defined for each $i \in I$. An alternative way of writing an indexed interleaving in the special case where the indexing set is an interval of integers $\{i \mid m \leqslant i \leqslant n\}$ is

$$|||_{i=m}^{n} P_i$$

EXAMPLE 2.19 A node similar to the *NODE* process of Example 2.17, but which can hold a maximum of $n$ messages, could be described as an interleaved combination of $n$ versions of *COPY*:

$$n\_NODE \quad = \quad |||_{0 \leqslant i < n} C(i)$$

where each $C(i)$ is defined to be *COPY*, for $0 \leqslant i < n$. To describe an interleaved combination of $n$ copies of the same process $P$ there is a convenient shorthand

$$|||_{0 \leqslant i < n} P$$

so an alternative description of *n_NODE* would be given by

$$n\_NODE \quad = \quad |||_{0 \leqslant i < n} COPY$$

□

## 2.3 INTERFACE PARALLEL

Synchronizing parallel and interleaving parallel can be blended into a hybrid form of parallel combination, which requires synchronization only on those events appearing in a common interface $A \subseteq \Sigma$ (as well as termination). This is described as

$$P_1 \underset{A}{||} P_2$$

The operational semantics is straightforward:

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{a} P_2'}{P_1 \underset{A}{\|} P_2 \xrightarrow{a} P_1' \underset{A}{\|} P_2'} \quad [\, a \in A^{\checkmark} \,]$$

$$\frac{P_1 \xrightarrow{\mu} P_1'}{\begin{array}{l} P_1 \underset{A}{\|} P_2 \xrightarrow{\mu} P_1' \underset{A}{\|} P_2 \\ P_2 \underset{A}{\|} P_1 \xrightarrow{\mu} P_2 \underset{A}{\|} P_1' \end{array}} \quad [\, \mu \notin A^{\checkmark} \,]$$

$P_1$ and $P_2$ co-operate on any event drawn from $A$, and interleave on events not in $A$.

EXAMPLE 2.20 A runner in a race engages in two events, *start* and *finish*:

$$RUNNER \quad = \quad start \rightarrow finish \rightarrow STOP$$

Two runners should synchronize on the *start* event, but they *finish* independently.

$$RACE \quad = \quad RUNNER \underset{\{start\}}{\|} RUNNER$$

□

## Indexed interface parallel

This parallel operator is associative provided the same interface set is used throughout: $P_1 \underset{A}{\|} (P_2 \underset{A}{\|} P_3)$ has the same executions as $(P_1 \underset{A}{\|} P_2) \underset{A}{\|} P_3$. It is also commutative. This allows the operator to be generalized as follows:

$$\underset{A\, i \in I}{\|} P_i$$

where $I$ is a finite indexing set, and $P_i$ is defined for each $i \in I$. It describes the process where any occurrence of an event from $A$ must involve all of the $P_i$. An occurrence of any event not in $A$ involves exactly one of those processes.

EXAMPLE 2.21 A marathon involving $30,000$ runners could be described as

$$MARATHON \quad = \quad \underset{\{start\}}{\|}{}_{i=1}^{30,000} RUNNER$$

All runners start at the same time, but each of them finishes independently. □

EXAMPLE 2.22 A function applied to a particular argument can be computed in two ways: using algorithm $g$ and using algorithm $h$. These two functions should agree on the value they compute for any particular input $x$, so the intention is that $g(x) = h(x)$ for any input $x$.

A module is written for each algorithm. The communication pattern of the modules is written as

$$\begin{aligned} G &= in?x : T \to out!g(x) \to SKIP \\ H &= in?x : T \to out!h(x) \to SKIP \end{aligned}$$

These modules can be run concurrently, but there are a number of ways in which this may be accomplished.

1. A fault-tolerant approach would run $G$ and $H$ in parallel, synchronizing on input and output. The combination $G \parallel H$ accepts one input which is received by both $G$ and $H$, and also synchronizes on output. This means that an output can occur only if both modules agree on its value. If the modules disagree, then a deadlock occurs and successful termination cannot occur.

2. To receive the result of the fastest calculation, an independent approach could be adopted, interleaving $G$ and $H$. The combination $G \mid\mid\mid H$ has to accept the input twice, since each module accepts its input independently of the other. If only one input is provided, then only one of the modules is executed, though the user has no control over which. Furthermore, the combination does not ensure that the same input is provided to each module.

3. The combination $G \underset{in.T}{\parallel} H$ allows a single input to be received by both modules, but allows for independent output, so a result can be obtained after the first module has completed its calculation. It cannot terminate until both outputs have occurred.

Different flavours of concurrency are appropriate for different requirements. □

## Exercises

EXERCISE 2.1 Give the transition graph for *MACHINE* of Example 2.4.

EXERCISE 2.2 Give the transition graph for $CUST \parallel ATT$ of Example 2.3. What behaviour does this parallel combination exhibit that you would not expect to find in a real cloakroom system? Amend the descriptions of the interacting parties *CUST* and *ATT* appropriately to remove this possibility.

EXERCISE 2.3 Give the transition graph for *PAINTING* of Example 2.8, and use it to identify all the ways in which deadlocks can occur.

EXERCISE 2.4 The book shop of Example 2.18 does not contain sufficient detail to prevent fraud: it allows any book to be claimed with any receipt. Adapt the description to keep track of the identity of the book that has been lodged throughout the payment procedure, so that customers can only take the books that have been paid for.

EXERCISE 2.5 A dishonest shopper will select an item, and will then either leave without paying, or else will pay if the circumstances in the shop make the first course of action infeasible. This can be described by the following process:

$$\begin{aligned} DCUSTOMER \quad = \quad & enter \to select \to (\; pay \to leave \to DCUSTOMER \\ & \qquad\qquad\qquad\quad \Box\; leave \to DCUSTOMER) \end{aligned}$$

What is the expected behaviour of this customer in parallel with the *SHOP* process of Example 2.7? What difference does it make to the expected behaviour if the external choice is replaced with an internal one?

EXERCISE 2.6 Draw the hypercube in the case where $n = 4$. How many ways are there for a message to get from $(1, 0, 0, 1)$ to $(0, 0, 1, 0)$ using the message routing algorithm?

EXERCISE 2.7 Consider the hypercube network of Example 2.13.

1. Not all of the channel names in Figure 2.6 have been given their subscripts. Give the subscripts for the remaining channels.
2. Is *MAILER* deadlock-free?
3. Is it deadlock-free if the next destination for a message is chosen internally by nodes, rather than by an external choice, as follows:

$$\begin{aligned} N_l(\langle\rangle) \quad = \quad & \Box_{k \in adj(l)}\, c_{k,l}?(d.m) \to N_l(\langle(d,m)\rangle) \\ & \Box\; in_l?(d.m) \to N_l(\langle(d,m)\rangle) \end{aligned}$$

$$\begin{aligned} & N_l(\langle(d,m)\rangle \frown s) \\ & \quad = \quad out_l!m \to N_l(s) \qquad\qquad\qquad\qquad \text{if } d = l \\ & \qquad\quad\; (\textstyle\sqcap_{k \in next(l,d)}\, c_{l,k}!(d.m) \to N_l(s)) \qquad \text{otherwise} \\ & \qquad\quad\; \Box \\ & \qquad\quad\; \Box_{k \in adj(l)}\;\; c_{k,l}?(d'.m') \to \\ & \qquad\qquad\qquad\qquad N_l(\langle(d,m)\rangle \frown s \frown \langle(d',m')\rangle) \end{aligned}$$

4. Is it deadlock-free if nodes can hold at most one message, i.e. they block input when they hold a message (rather than being able to hold arbitrarily many), as follows:

$$\begin{array}{rcll} N_l(\langle\rangle) & = & \Box_{k \in adj(l)} c_{k,l}?(d.m) \to N_l(\langle(d,m)\rangle) & \\ & = & \Box\ in_l?(d.m) \to N_l(\langle(d,m)\rangle) & \\ & & & \\ N_l(\langle(d,m)\rangle) & = & out_l!m \to N_l(\langle\rangle) & \text{if } d = l \\ & & (\Box_{k \in next(l,d)} c_{l,k}!(d.m) \to N_l(\langle\rangle)) & \text{otherwise} \end{array}$$

5. What is the maximum number of nodes a message will pass through in a network of $2^n$ nodes?

EXERCISE 2.8 Consider the array *SORTER* of Example 2.12:

1. Is it deadlock-free?
2. Is it deadlock-free if the order of each cell's output is reversed (so output occurs on the $h$ channel before the $v$ channel) as follows:

$$C_{i,j} \quad = \quad h_{i,j}?x \to v_{i,j}?y \to h_{i+1,j}!\max\{x,y\} \to v_{i,j+1}!\min\{x,y\} \to C_{i,j}$$

3. Is it deadlock-free if the order of both inputs and outputs for a cell is reversed as follows:

$$C_{i,j} \quad = \quad v_{i,j}?y \to h_{i,j}?x \to h_{i+1,j}!\max\{x,y\} \to v_{i,j+1}!\min\{x,y\} \to C_{i,j}$$

Which other orders of inputs and outputs avoid deadlock?

EXERCISE 2.9 Show that interface parallel is not associative in general when the event sets are different, by finding processes $P_1$, $P_2$, and $P_3$ and sets $A$ and $B$ such that $P_1 \underset{A}{\|} (P_2 \underset{B}{\|} P_3)$ is different from $(P_1 \underset{A}{\|} P_2) \underset{B}{\|} P_3$.

# 3 Abstraction and control flow

## 3.1 HIDING

When a collection of processes has been combined into a system, there will often be communications between the components which should be internal. Such events are inappropriate to include at the system interface, which should contain only those events through which the system interacts with its environment.

When processes are placed in parallel, the events on which they synchronize remain in the interface of the combination. This is the mechanism which supports multi-way synchronization. It also means that even when all the intended participants in an event have been described and composed in parallel, the event is still in the interface of the combination. A new CSP operator, *event hiding*, is required to encapsulate the event within the process, and to remove it from the interface.

A set of events $A$ may be encapsulated within a process $P$ using the notation

$$P \setminus A$$

(pronounced '$P$ hide $A$'). This operation describes the case where all participants of all events in $A$ are already known and described in $P$. All these events are removed from the interface of the process, since no other processes are required to engage in them. These events become *internal* to the process $P$. The operational rules reflect this:

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \quad [\, a \in A \,]$$

$$\frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A} \quad [\, \mu \notin A \,]$$

The process $P \setminus A$ may make the same transitions as $P$, but all the events in $A$ are renamed to the internal event $\tau$. Termination cannot be hidden, so the event ✓ must not appear in the set $A$.

This operator is used in design and in implementation. It explains a process in terms of internal activity, so it is used in a description of how a particular end is accomplished. It is not appropriate at the level of specification, since at that level internal events should not even be mentioned: specifications of processes should be concerned purely with the behaviour on their external events.

EXAMPLE 3.1 A spy listens out for particular pieces of information, and then relays them to a master spy who logs them. In order for the spy to be effective, it is important that the relaying of information is kept hidden from its environment.

$$\begin{aligned} SPY &= listen?x : T \to relay!x \to SPY \\ MASTER &= relay?y : T \to log!y \to MASTER \end{aligned}$$

The combination of the master and the spy is described by

$$(SPY \parallel MASTER) \setminus relay.T$$

The only visible activity the spy is involved in is listening. □

EXAMPLE 3.2 A stop-and-wait protocol implements a one-place buffer. It consists of two halves, $S$ and $R$: a message is input to $S$, passed to $R$, and finally output from $R$.

$$\begin{aligned} S &= in?x : T \to mid!x \to ack \to S \\ R &= mid?y : T \to out!y \to ack \to R \end{aligned}$$

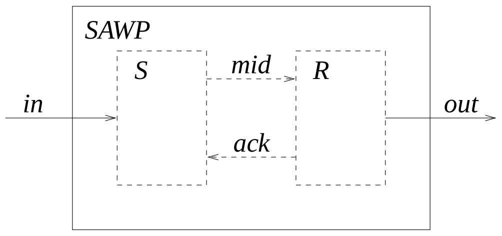


***Fig. 3.1*** A stop-and-wait protocol

Having accepted a message, the sender *S* passes the message to *R* along channel *mid*, and then waits for an acknowledgement before accepting the next message. The receiver *R* accepts messages along *mid*, and sends an acknowledgement once a message has been output.

The two halves of the protocol are designed to combine in parallel. The channel *mid* and the acknowledgement event *ack* are private connections and should have no participants other than *S* and *R*. The protocol is then described as

$$SAWP \quad = \quad (S \parallel R) \setminus (mid.T \cup \{ack\})$$

This is pictured in Figure 3.1 □

EXAMPLE 3.3 Each cell $N_l$ in the network of cells connected as vertices of a hypercube, described in Example 2.13 as *MAILER*, has interface

$$A_l \quad = \quad \{in_l, out_l\} \cup \{c_{k,l} \mid k \in adj(l)\} \cup \{c_{l,k} \mid k \in adj(l)\}$$

The $in_l$ and $out_l$ channels are intended for communication with the users of the network, and the *c* channels are used for the cells to pass messages between each other. The intention is that no external parties are involved in the communications on the *c* channels. The only processes involved in communications on any particular channel $c_{k,l}$ are the cells $N_k$ and $N_l$. In order to encapsulate the *c* channels within the process *MAILER* the hiding operator is used:

$$MAIL\_SERVICE \quad = \quad MAILER \setminus \{c_{i,j} \mid i,j \in COORD \wedge j \in adj(i)\}$$

The only external events that *MAIL_SERVICE* can perform are communications along the channels $in_l$ and $out_l$ for each *l*; only through these events can it interact with its environment. □

A process exercises complete control over its internal events. With this control over when internal events are performed comes the responsibility to perform them: internal events should not be delayed indefinitely once they are enabled, since otherwise progress could not be expected. In the Stop-and-Wait Example 3.2 the environment can expect a message to be

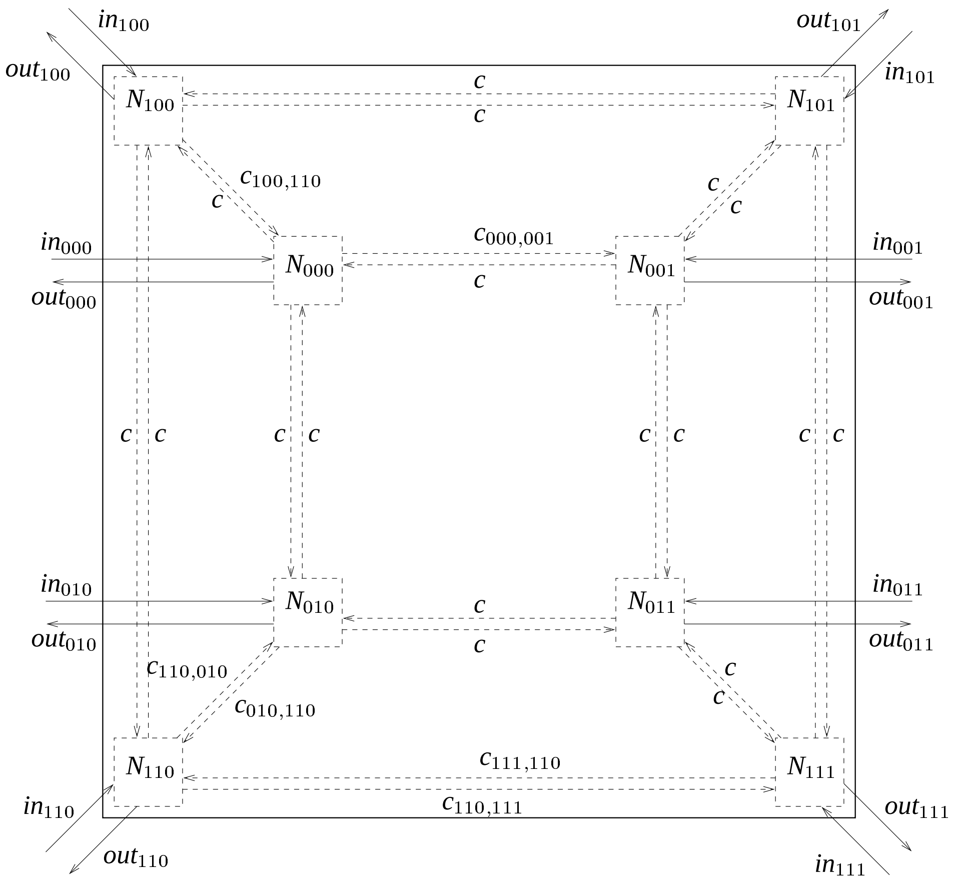


**Fig. 3.2** The process *MAIL_SERVICE*

offered as output after it has been input. The message must be passed internally along *mid* after it has been received on the *in* channel, and *SAWP* cannot refuse to perform a *mid* event, or indeed an *ack* event, once it is enabled.

When the events offered by an external choice are hidden, the environment no longer has any control over how the choice is resolved: it is resolved internally.

EXAMPLE 3.4 A fax is to be sent to someone who has two fax machines. A secretary is given the fax to send (modelled by the event *in.x*). It can be sent to the first, modelled by the channel *send*.1, or it can be sent to the second, modelled by channel *send*.2. The secretary is prepared to send it to either number, and offers the choice to her boss, modelled as an external choice. Sometime later a receipt is obtained indicating successful transmission to the corresponding machine. This situation is described by *SEC*.

$$\begin{aligned} SEC \quad = \quad & in?x \rightarrow (\ send.1!x \rightarrow received.1 \rightarrow STOP \\ & \qquad\qquad \Box\ send.2!x \rightarrow received.2 \rightarrow STOP) \end{aligned}$$

If the boss does not wish to be involved in the choice between different fax numbers, she delegates the choice by hiding the channels *send*.1 and *send*.2, giving complete control over them to the secretary.

$$SEC \setminus (send.1.T \cup send.2.T)$$

The hiding of these events removes them from those communications on which the boss and the secretary have to agree. They are encapsulated within the process *SEC*, indicating that all participants (in this case just one) have been identified. Although the secretary has complete control over which one to perform, she is still obliged to perform one of them: the boss can expect a receipt. From the point of view of the boss, this choice will now be resolved internally. After giving a fax to the secretary she has no control over which of the two machines will receive the fax, and will only find out which it was once the receipt is obtained. Observe that if the receipts were indistinguishable (both modelled by the single event *receipt*) then the boss would have no way of determining which way the choice was made. □

EXAMPLE 3.5 In Example 2.13, when a cell $C_l$ is waiting to send a message to an adjacent cell, it offers an external choice of all the possibilities. It is willing to send its message to any cell that is ready to receive it, and its environment—the rest of the network and the rest of the world—will determine how the choice is made. Since all cells are always ready to receive messages, the choice is available externally. When the rest of the world is excluded by hiding the communication channels between cells to obtain *MAIL_SERVICE* (pictured in Figure 3.2), the choice must be made internally within the process *MAIL_SERVICE* itself. The environment is not concerned with the routes that messages travel, only with the assurance that they will arrive. □

When only one event of an binary external choice is made internal, the process is required to make a choice between performing the internal one autonomously, or waiting for a synchronization on the external one. If its environment is not prepared to engage in the external event, then its responsibility to perform the internal event means that it cannot wait indefinitely for the external one, since this would involve indefinitely delaying the internal event. On the other hand, if the environment is prepared to engage in the external event, then one of two things could happen: either the choice has not yet been made, and the external event can occur and resolve the choice in its favour; or the internal event has already occurred, since the environment cannot prevent it from occurring, and the external event is no longer available.

EXAMPLE 3.6 A printer queue which can hold one message at a time is described as follows:

$$\begin{aligned} PRINTQ \quad = \quad & in?x : JOB \rightarrow (\ print!x \rightarrow out!x \rightarrow PRINTQ \\ & \qquad\qquad \Box\ dequeue \rightarrow PRINTQ) \end{aligned}$$

When a job is queued, it will either be sent to the printer and received as output, or else it could be removed from the queue. The user is not normally involved in the communications between the queue and the printer, so the communications along the *print* channel will be internal. The process which the user interacts with is

$$PRINTQ \setminus print.JOB$$

The user has no control over when the job will be sent to the printer. After inputting a job, it may be possible to dequeue it if it has not yet reached the printer, but the other possibility, entirely outside the control of the user, is that it may already have been sent to the printer and the option of dequeuing has been withdrawn. □

EXAMPLE 3.7 A course of action might be made available for a particular interval, but is then timed out if it has not yet been chosen. Although timed CSP will enable a more precise description of this kind of behaviour, it is possible to analyze it in terms of a timeout event. In this case a choice is offered between the initial event, and the timeout event. For example, a special offer is available only for a limited period, after which the offer lapses and purchase must then be at the standard rate:

$$OFFER \quad = \quad ((cheap \rightarrow STOP) \,\Box\, (lapse \rightarrow standard \rightarrow STOP)) \setminus \{lapse\}$$

The user has no control or influence over when the cheap offer will end, so the timeout event *lapse* is made internal. It is possible to buy at the cheap price if the offer has not yet lapsed, but it is also possible that the cheap price has been retracted at the point the purchaser is ready to buy, and that only the standard price is available. □

EXAMPLE 3.8 A stop-and-wait protocol which permits its input to be overwritten once if it has not already passed along the *mid* channel, might be described as follows:

$$\begin{aligned} S2 &= in?x \rightarrow (S2 \,\Box\, (mid!x \rightarrow ack \rightarrow S2)) \\ R2 &= mid?y \rightarrow out!y \rightarrow ack \rightarrow R2 \end{aligned}$$

After an input, the sender $S2$ is prepared either to pass the input along *mid*, or to accept another input which displaces the previous one. The receiver $R2$ is exactly the same as the original receiver $R$ of Example 3.2. The two halves of the protocol are combined as $S2 \parallel R2$, and the internal channels are hidden:

$$SAWP2 \quad = \quad (S2 \parallel R2) \setminus mid.T \cup \{ack\}$$

After its first input $in.x$, the process $SAWP2$ is in the position where a choice is to be made between an external event $in.w$ and an internal event $mid.x$. If at this point the environment simply waits for output to be offered, and offers no further input, then the internal event must occur, and output is indeed offered. If instead the environment offers a second input, then there are two possibilities: the internal event has not yet occurred, and the second input is accepted; or the choice has already been made in favour of the internal communication, in which case the second input will be refused. The environment is unable to prevent this second possibility. $\Box$

The internalization of events may introduce the possibility of internal events occurring indefinitely. Since the environment has no control over the internal events of a process, this means that the process is free to execute these internal events forever and consequently avoid any further interaction with its environment. This possibility is called *divergence*. When applying the hiding operator care should be taken to avoid divergence, since progress cannot be guaranteed for any divergent process—it may consume computing resources forever without any assurance that it will ever again respond to its environment.

EXAMPLE 3.9 A parity server offers alternating bits on its output channel:

$$PARITY \quad = \quad out!0 \rightarrow out!1 \rightarrow PARITY$$

If its output channel is hidden, then it simply repeats *out*.0 and *out*.1 internally forever. The process $PARITY \setminus out.\{0, 1\}$ behaves as an internal loop, unable to interact with its environment, but consuming computing resources and thereby possibly preventing other processes from executing. In this respect it is worse than a deadlocked process, which is also unable to interact with its environment but at least is not consuming resources. $\Box$

EXAMPLE 3.10 A process reads data from two input channels, and outputs on a single channel. It is able to be ready only on one input channel at a time, but it may switch between them by means of an event *switch*. At any point where it is waiting for input, it offers a choice between accepting input on that channel, and switching to wait on the other channel.

$$\begin{aligned} POLL \quad = \quad & in.1?x \rightarrow out!x \rightarrow POLL \\ & \Box\ switch \rightarrow (\ in.2?x \rightarrow out!x \rightarrow POLL \\ & \qquad\qquad\qquad \Box\ switch \rightarrow POLL) \end{aligned}$$

If the *switch* event is hidden, to enable the polling process to switch channels independently of its environment, then the process $POLL \setminus \{switch\}$ is obtained. Unfortunately, this process may spend its entire time switching between channels in preference to ever accepting any message on either of them. At any stage, there is a choice between an external communication or an internal *switch*. Although the external communication is possible, so is the internal event. Both are of equal priority, but the internal event is entirely under the control of *POLL* and cannot be prevented from occurring. Since this is true at every stage of an execution, it is possible that *POLL* simply resolves the choice in favour of *switch* every time, resulting in an execution consisting entirely of internal events. This is possible even if its environment is offering input on both *in*.1 and *in*.2—there is no guarantee that either input will ever be accepted. $\Box$

## 3.2 EVENT RENAMING

When a process' pattern of communication has been described in terms of particular events, it is possible to obtain a new process by renaming those events. Its executions are essentially those of the original process but where the events are renamed. This allows the reuse of particular descriptions of process behaviour without the need to rewrite the process in full and replacing all the original event names with the new ones.

A total function on events $f : \Sigma^{\checkmark} \to \Sigma^{\checkmark}$ is used to describe the required change of event names. For any such function, there are two ways a process can have its events renamed: by applying the function $f$ to each event, or by applying its inverse $f^{-1}$ to each event. The interface through which the process interacts with its environment is transformed by $f$ and $f^{-1}$ respectively. The function $f$ must map external events to external ones—it cannot be used to make events internal. Termination cannot be renamed, so $f(a) = \checkmark \Leftrightarrow a = \checkmark$ for any event renaming function.

### Forward renaming

The process $f(P)$ is able to perform an event $f(a)$ precisely when the process $P$ could perform the corresponding event $a$. Furthermore, $f(P)$ can perform internal events whenever $P$ can. This behaviour is captured by the following transition rules:

$$\frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')}$$

$$\frac{P \xrightarrow{\tau} P'}{f(P) \xrightarrow{\tau} f(P')}$$

If the function $f$ is one-one, then the process $P$ might be thought of as capturing a generic behaviour pattern, with $f(P)$ a particular instance of it. Whenever $f(P)$ offers or performs some event $b$, then that corresponds to $P$ offering or performing $f^{-1}(b)$. Choices offered by $P$ become choices offered by $f(P)$.

EXAMPLE 3.11 The children of Example 2.8 have similar patterns of behaviour. The behaviour of Isabella was described by

$$\begin{aligned} ISABELLA \quad = \quad & isabella.get.box \to isabella.get.easel \to isabella.paint \to \\ & \quad isabella.drop.box \to isabella.drop.easel \to ISABELLA \\ & \Box\ isabella.get.easel \to isabella.get.box \to isabella.paint \to \\ & \quad isabella.drop.box \to isabella.drop.easel \to ISABELLA \end{aligned}$$

and the behaviour of Kate was obtained by taking the description *ISABELLA* and replacing all occurrences of the name *isabella* with the name *kate*. Instead of doing this explicitly, this may be accomplished by means of event renaming. Let the function $f : \Sigma^{\checkmark} \to \Sigma^{\checkmark}$ be defined by

$$\begin{aligned} f(isabella.x) &= kate.x \\ f(y) &= y \qquad \text{if } y \text{ is not of the form } isabella.x \end{aligned}$$

We will adopt the convention in function definition that the function is defined to be the identity on events which are not explicitly covered by the definition. Hence the above function could have been defined simply by the clause $f(isabella.x) = (kate.x)$. In either case,

$$KATE = f(ISABELLA)$$

gives an alternative definition for *KATE*. □

EXAMPLE 3.12 The process *CHAIN* in Example 2.11 is made up of a number of individual one-place buffers. Each of those buffers behaves in essentially the same way, but on different channels. The behaviour pattern can be captured as the generic one-place buffer *COPY* of type $\mathbb{Z}$, defined first in Example 1.15:

$$COPY = in?x : \mathbb{Z} \to out!x \to COPY$$

This process has $in.\mathbb{Z} \cup out.\mathbb{Z}$ as its interface.

Each process $P_i$ in the chain was defined explicitly as

$$P_i = c_i?x : \mathbb{Z} \to c_{i+1}!x \to P_i$$

Each one could instead be defined using a corresponding event renaming function $f_i$ defined by

$$\begin{aligned} f_i(in.m) &= c_i.m \\ f_i(out.m) &= c_{i+1}.m \end{aligned}$$

The processes $P_i$ could instead have been defined as

$$P_i = f_i(COPY)$$

The interface of each $P_i$ will be the interface of *COPY* transformed by $f_i$, which is $c_i.\mathbb{Z} \cup c_{i+1}.\mathbb{Z}$. □

One special form of one-one event renaming attaches a particular label to all events in a process description. If the label is $l$, then the function $f_l$ to be applied is given by

$$f_l(x) = l.x$$

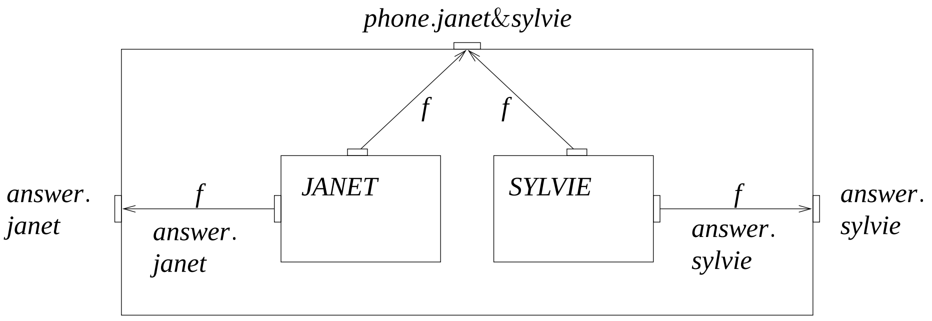


**Fig. 3.3** The interface of the process $f(OFFICE)$

for all $x \in \Sigma$, and $f_l(\checkmark) = \checkmark$. Then the process $f_l(P)$ performs events $l.a$ whenever $P$ would have performed $a$. There is a special form for this kind of event renaming. The process construction $l : P$ is shorthand for $f_l(P)$ where $f_l$ is as defined above.

EXAMPLE 3.13 In Example 2.8, the generic behaviour pattern of a child painting is captured as the process *PAINT*:

$$\begin{array}{l} PAINT \\ \quad = \; get.box \to get.easel \to paint \to drop.box \to drop.easel \to PAINT \\ \qquad \Box \; get.easel \to get.box \to paint \to drop.box \to drop.easel \to PAINT \end{array}$$

Then *KATE* and *ISABELLA* can be described as particular instances of this process, as $kate : PAINT$ and $isabella : PAINT$ respectively. □

All aspects of process behaviour are transformed directly when a process is renamed under a one-one function. A function which maps a number of different events to the same single event can alter some features of the process' behaviour, particularly with regard to choice. If two events of a choice are mapped to the same event, then the environment is no longer able to choose between these two branches of the choice, since they are now both triggered by the same event: if the environment chooses that event, then either branch could be chosen, and no further control over which one is actually chosen is available externally. Hence many-one event renaming may affect the nature of particular choices and may introduce some internal choices where there were previously external ones.

EXAMPLE 3.14 Janet and Sylvie share an office which contains two phones. To converse with either of them it is sufficient to dial their phone number. The possibilities for phoning and answering are described as follows:

$$\begin{array}{rcl} OFFICE & = & JANET \; ||| \; SYLVIE \\ JANET & = & phone.janet \to answer.janet \to JANET \\ SYLVIE & = & phone.sylvie \to answer.sylvie \to SYLVIE \end{array}$$

The two phones are given the same number, required to serve both Janet and Sylvie. The effect of this is to map both *phone.janet* and *phone.sylvie* to a single event *phone.janet&sylvie*. The effect on *OFFICE* is to transform it through the event mapping $f$ given by

$$\begin{aligned} f(phone.janet) &= phone.janet\&sylvie \\ f(phone.sylvie) &= phone.janet\&sylvie \end{aligned}$$

The process $f(OFFICE)$, pictured in Figure 3.3, initially offers the event *phone.janet&sylvie*. The next event could be either *answer.janet* or *answer.sylvie*, and the caller no longer has any control over which of these will occur. The event renaming has altered the nature of the choice available to the caller. □

## Backward renaming

A process may also have its interface changed through renaming under $f^{-1}$ where $f$ is again a total function on $\Sigma^{\checkmark}$. The intention here is that the process $f^{-1}(P)$ can perform $a$ whenever $P$ can perform $f(a)$. Internal events are again unchanged by this form of renaming. Its effect is described by the following transition rules:

$$\frac{P \xrightarrow{f(a)} P'}{f^{-1}(P) \xrightarrow{a} f^{-1}(P')}$$

$$\frac{P \xrightarrow{\tau} P'}{f^{-1}(P) \xrightarrow{\tau} f^{-1}(P')}$$

In the case where $f$ is bijective then $f^{-1}$ is also a bijection and the process $f^{-1}(P)$ can also be treated as a forward renamed process (renamed by the function $f^{-1}$).

When $f$ is many-to-one, then backward renaming is different from any form of forward renaming. In this case, whenever $P$ is able to perform an event $a$, the process $f^{-1}(P)$ is able to perform any of the events that map onto the event $a$. This allows a single event in the description of a process' behaviour pattern to correspond to a choice of alternatives at its interface: all the events that map onto $a$ are available to the environment of $f^{-1}(P)$, and any of them can be chosen. This form of renaming represents interface expansion, where single events in the behaviour pattern can be triggered by a number of different possibilities at its interface.

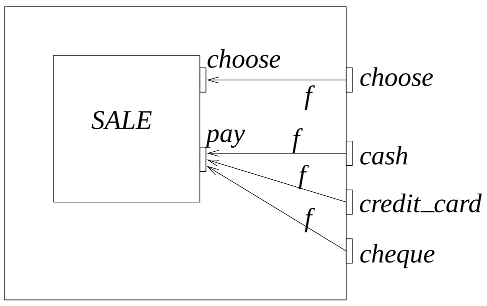


**Fig. 3.4** Interface expansion of *SALE* through backward renaming

EXAMPLE 3.15 The generic pattern of behaviour when a shop makes a sale is that items are chosen, and then paid for. This simple pattern may be described by the process

$$SALE \quad = \quad choose \rightarrow pay \rightarrow SALE$$

The shop offers a number of alternative payment methods: cash, credit card, or cheque. Rather than redefine the process *SALE* it is possible to describe these alternatives by use of the event renaming function

$$\begin{array}{lcl} f(cash) & = & pay \\ f(credit\_card) & = & pay \\ f(cheque) & = & pay \end{array}$$

Then whenever the process *SALE* is ready to accept *pay*, the process $f^{-1}(SALE)$ is prepared to engage in any of those events that map onto *pay*: it offers a choice to the customer between the events *cash*, *credit_card*, and *cheque*. □

## Chaining

Using event renaming, any event in the interface of one process can be connected to any event from a second process' interface so the two processes must synchronize on their performance of these events. The two events are each renamed to the same new event, and then the processes are composed in parallel. Their interfaces are effectively reconfigured so that they must now co-operate on these events.

A special form of this interface reconfiguration is useful for pipe processes, which have exactly two channels: *in* and *out*. When two such processes are connected together, the intention is that the output of the first is connected to the input of the second. This is accomplished by renaming each of these channels to *mid*, composing the resulting processes

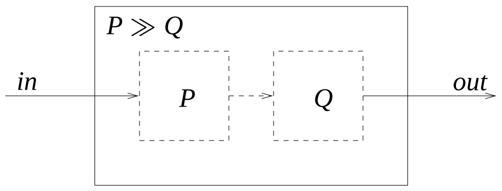


***Fig. 3.5*** A chain of two pipe processes

in parallel, and finally hiding the *mid* channel. The renaming functions required are particular instances of $swap_{c,d}$ for channel names $c$ and $d$, defined as follows:

$$\begin{array}{rcll} swap_{c,d}(c.x) & = & d.x & \\ swap_{c,d}(d.x) & = & c.x & \\ swap_{c,d}(y) & = & y & \text{if } y \neq c.x \text{ and } y \neq d.x \end{array}$$

The *chaining* operator on pipe processes may then be defined in terms of the operators that have already been defined earlier in this chapter:

$$P_1 \gg P_2 \quad \equiv \quad (swap_{out,mid}(P_1) \parallel swap_{in,mid}(P_2)) \setminus mid$$

The resulting process is again a pipe: the only two channels it has are *in* and *out*. The operator is illustrated in Figure 3.5.

EXAMPLE 3.16 Two instances of the *COPY* process chained together produce a pipe which is a buffer of capacity two:

$$2COPY \quad = \quad COPY \gg COPY$$

This process is initially ready for input. After the first data item is input, the left hand *COPY* passes it along the internal channel *mid* to the right hand *COPY*, which makes it available for output. At this point the left hand *COPY* is empty, and is ready to accept a second input, which would lead to the buffer becoming full. Alternatively the right hand *COPY* is ready to output, which would lead to 2*COPY* returning to its original empty state. □

The chaining operator is associative: $P_1 \gg (P_2 \gg P_3)$ has the same executions as $(P_1 \gg P_2) \gg P_3$. This allows a generalized form on sequences of processes. If $P_i$ are processes for $i$ between 1 and $n$, then $\gg_{i=1}^{n} P_i$ is the chaining together of all of those $n$ processes in order:

$$\gg_{i=1}^{n} P_i \quad \equiv \quad P_1 \gg P_2 \gg \ldots \gg P_n$$

If each $P_i$ is actually a copy of the same process $P$, then it is conventional to write $\gg_{i=1}^{n} P$ for the sequence of $n$ copies of $P$ chained together.

EXAMPLE 3.17 Newton's method for approximating square roots of a positive number $n$ states that if $a_i$ is an approximation to $\sqrt{n}$, then $a_{i+1} = \frac{1}{2}(a_i + \frac{n}{a_i})$ is a better approximation. Successive values of $a_i$ are calculated, and the final one in the sequence is taken to be the best approximation.

A pipe which calculates one step of the sequence is defined as follows:

$$NEWTON \quad = \quad in?(n,a) : \mathbb{N} \times \mathbb{R} \rightarrow out!(n,(a + \frac{n}{a})/2) \rightarrow NEWTON$$

If $n$ successive approximations are required, then $n$ copies of *NEWTON* should be chained together. The first copy requires the first approximation as input: this can be provided by a *HEAD* process. The last copy outputs both the original number and the square-root approximation: a *FOOT* process can extract the information required—the final approximation. These bracketing processes are defined as follows:

$$\begin{aligned} HEAD \quad &= \quad in?n : \mathbb{N} \rightarrow out!(n,1) \rightarrow HEAD \\ FOOT \quad &= \quad in?(n,a) : \mathbb{N} \times \mathbb{R} \rightarrow out!a \rightarrow FOOT \end{aligned}$$

The entire square-root extracting process consists of all of these components chained together into one pipe:

$$SQRT \quad = \quad HEAD \gg (\gg_{i=1}^{n} NEWTON) \gg FOOT$$

The initial approximation to the square-root is 1. □

## Renaming recursive calls

Applying an event renaming function to recursive calls allows fresh invocations of the process to offer different events. This allows for the event possibilities to change dynamically as the execution unwinds. In the case where a process is spawned, and the fresh invocation of the process runs in parallel with the original one, it allows different channels on the fresh process to connect to existing channels on the original: altering the interface of a process alters the way in which it synchronizes with other processes.

EXAMPLE 3.18 The hour changer on a 24 hour clock cycles repeatedly through the hours from $0$ to $23$, outputting the value on the channel *hour*, with $hour.0$ as its first output. This may be described using an event renaming function *inc* that increments the hour value by $1$, modulo $24$:

$$inc(hour.n) \quad = \quad hour.((n+1) \bmod 24)$$

The hour changer is then defined as follows:

$$HOUR \quad = \quad hour!0 \rightarrow inc(HOUR)$$

Each time an hour value $h$ is output, the process is recursively invoked under the *inc* function. $\square$

EXAMPLE 3.19 A process which models a set with two operations, *add* and *query*, can be described using new invocations of the process in parallel with existing ones. The operation *add* allows the addition of an element to the set; and the operation *query* permits an enquiry as to whether a particular element is in the set or not. The answer, drawn from $ANS = \{yes, no\}$, is passed on channel *answer*.

The empty set can be defined as follows:

$$\begin{array}{rcl} SET & = & query?x : T \rightarrow answer!no \rightarrow SET \\ & & \square\ add?x : T \rightarrow (NODE(x) \underset{INT}{\parallel} i : SET) \setminus INT \end{array}$$

$$\begin{array}{rcll} NODE(x) & = & query?y : T \rightarrow & answer!yes \rightarrow NODE(x) \quad \text{if } y = x \\ & & & i.query!y \rightarrow i.answer?z : ANS \rightarrow \quad \text{if } y \neq x \\ & & & \quad answer!z \rightarrow NODE(x) \\ & & \square\ add?x : T \rightarrow i.add!x : T \rightarrow NODE(x) & \end{array}$$

When an element $x$ is added to the empty set, it is stored in $NODE(x)$, with a fresh copy of the empty set subordinated, labelled with $i$ and hidden. The internal channels are given by

$$INT = i.add.T \cup i.query.T \cup i.answer.ANS$$

The process $NODE(x)$ communicates with the fresh empty set using the internal channels *i.query*, *i.answer*, and *i.add*. When a query is input, if $NODE(x)$ cannot answer it then it passes the query on to the rest of the set and passes the answer on to the user. When a fresh item is to be added to the set, $NODE(x)$ passes it on to the rest of the set. The state after two items have been added to the set is pictured in Figure 3.6.

This process provides an implementation of the process specified in Example 1.21. $\square$

## 3.3 SEQUENTIAL COMPOSITION

Processes execute when they are invoked, and it is possible that they continue to execute indefinitely, retaining control over execution throughout. It is also possible that control may pass to a second process, either because the first process reaches a particular point in its execution where it is ready to pass control, or because the second process demands it.

The mechanism for transferring control from a terminated process to another process is sequential composition. The process

$$P_1\, ;\, P_2$$

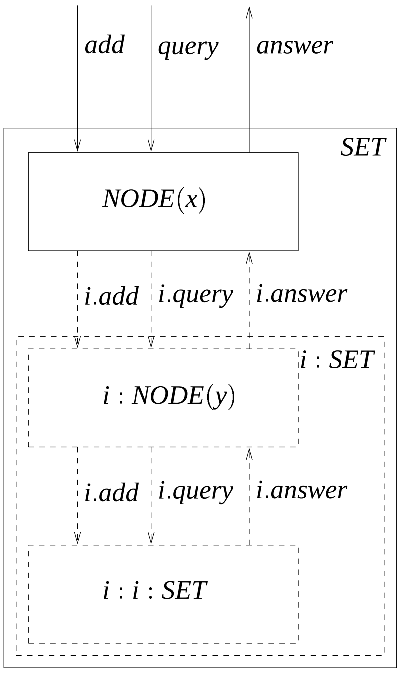


***Fig. 3.6*** The process *SET* after *x* and *y* have been added

executes component $P_1$ until it terminates, as indicated by its performance of a $\checkmark$ event, and then executes component $P_2$. The operational understanding is captured as follows:

$$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1;\ P_2 \xrightarrow{\mu} P_1';\ P_2} \quad [\ \mu \neq \checkmark\ ]$$

$$\frac{P_1 \xrightarrow{\checkmark} P_1'}{P_1;\ P_2 \xrightarrow{\tau} P_2}$$

The sequential composition $P_1;\ P_2$ initially executes as $P_1$. When $P_1$ terminates, its $\checkmark$ event becomes internal to the composition, since $P_1;\ P_2$ should not indicate that it has finished until $P_2$ finally terminates.

Process descriptions may be structured using sequential composition, where processes describe the different phases of the overall process. Specifications and system descriptions may thus be provided in a top down fashion, firstly identifying the phases that the process will pass through, and later providing the detailed description of the behaviour in the individual phases.

EXAMPLE 3.20 A different view of the purchasing process is provided by the description

$$\mathit{PURCHASE} \;=\; \mathit{CHOOSE};\ \mathit{PAY}$$

Each of the components represents a stage of the purchase process. These must be elaborated in order to complete the definition of *PURCHASE*, but its high-level structure is already clear. We may model *CHOOSE* and *PAY* in a number of alternative ways, without affecting the structure. Here we elect to use *CHOOSE* to describe a process where a shopper cannot rest until a suitable item has been found. The process *PAY* describes a variety of payment possibilities.

$$\begin{array}{rcl} \mathit{CHOOSE} & = & \mathit{select} \to (\ \mathit{keep} \to \mathit{SKIP} \\ & & \qquad\qquad \Box\ \mathit{return} \to \mathit{CHOOSE}) \end{array}$$

$$\begin{array}{rcl} \mathit{PAY} & = & \mathit{cash} \to \mathit{receipt} \to \mathit{SKIP} \\ & & \Box\ \mathit{cheque} \to \mathit{receipt} \to \mathit{SKIP} \\ & & \Box\ \mathit{card} \to \mathit{swipe} \to (\ \mathit{sign} \to \mathit{receipt} \to \mathit{SKIP} \\ & & \qquad\qquad\qquad\qquad \Box\ \mathit{reject} \to \mathit{PAY}) \end{array}$$

Repeated execution of the same component or sequence of components can be described by means of a recursive loop. The recursion

$$\mathit{SPENDING} \;=\; \mathit{PURCHASE};\ \mathit{SPENDING}$$

describes recurrent spending. □

EXAMPLE 3.21 A one-time stack accepts data along channel *in*. It continues to do this until the command *produce* occurs, after which all the data are output in reverse order. This can be described by use of a recursive call inside a sequential composition:

$$\begin{array}{rcl} \mathit{STORE} & = & \mathit{in}?x : T \to (\mathit{STORE};\ \mathit{out}!x \to \mathit{SKIP}) \\ & & \Box\ \mathit{produce} \to \mathit{SKIP} \end{array}$$

Each time a recursive call occurs, the message that was last input is stored up awaiting output. For example, after the three inputs $\mathit{in}.5$, $\mathit{in}.3$, $\mathit{in}.8$, the resulting process is

$$\mathit{STORE};\ (\mathit{out}!8 \to \mathit{SKIP});\ (\mathit{out}!3 \to \mathit{SKIP});\ (\mathit{out}!5 \to \mathit{SKIP})$$

While *STORE* continues to input data, the list of outputs can continue to grow. Once the event *produce* occurs then the potential for any further recursive calls is lost, and the sequence is output in the order of last-in-first-out. □

## 3.4 INTERRUPT

Control can also pass from one process $P_1$ to another process $P_2$ by means of an interrupt construction

$$P_1 \bigtriangleup P_2$$

This allows a process $P_1$ to have control removed from it at an arbitrary point of an execution. Unlike sequential composition, the process $P_1$ relinquishing control has no influence over when this occurs. The interrupting process $P_2$ may begin execution at any point throughout the execution of $P_1$: the performance of $P_2$'s first external event is the point at which control passes, and $P_1$ is discarded. The operational rules are as follows:

$$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \bigtriangleup P_2 \xrightarrow{\mu} P_1' \bigtriangleup P_2} \quad [\, \mu \neq \checkmark \,]$$

$$\frac{P_1 \xrightarrow{\checkmark} P_1'}{P_1 \bigtriangleup P_2 \xrightarrow{\checkmark} P_1'}$$

$$\frac{P_2 \xrightarrow{\tau} P_2'}{P_1 \bigtriangleup P_2 \xrightarrow{\tau} P_1 \bigtriangleup P_2'}$$

$$\frac{P_2 \xrightarrow{a} P_2'}{P_1 \bigtriangleup P_2 \xrightarrow{a} P_2'}$$

The process is able to perform any execution of $P_1$. Throughout $P_1$'s execution the interrupting process $P_2$ is also ready to begin, and the interruption occurs on its first external event. If $P_1$ terminates while it is executing then the entire construct is terminated and $P_2$ is discarded.

Nondeterminism could arise if $P_1$ and $P_2$ are both able to perform the same event at any stage, since if that event occurs then the result could be either that the interrupt has occurred, or that it has not. It is pragmatic to ensure where possible that $P_2$ cannot have as its first event any event which $P_1$ can perform.

EXAMPLE 3.22 The process *KATE* of Example 2.8 can be interrupted at any point by *bath* in the following description:

$$KATE \bigtriangleup bath \rightarrow bed \rightarrow SKIP$$

$\Box$

EXAMPLE 3.23 A process which models a variable of type $T$ allows values to be written to it along channel *write*, and the current value it is holding can be read by means of the channel *read*. This may be described as follows:

$$\begin{array}{rcl} VAR & = & write?x : T \rightarrow (VAR(x) \bigtriangleup VAR) \\ VAR(x) & = & read!x \rightarrow VAR(x) \end{array}$$

The process inputs a value and is then prepared to output it repeatedly until interrupted by the next *write*. $\Box$

The particular form $P_1 \bigtriangleup e \rightarrow P_2$ has a single interrupt event $e$, and identifies it explicitly. This may also be written $P_1 \bigtriangleup_e P_2$. The situation where there is a set $A$ of interrupt events available, and each event $a \in A$ is associated with a particular interrupt handler $P(a)$, can be described as follows:

$$P \bigtriangleup (x : A \rightarrow P(x))$$

EXAMPLE 3.24 The main tasks of an office junior are to make tea, to do photocopying, and to do filing. This activity may be temporarily interrupted by the phone ringing, which requires a message to be taken, or by the boss arriving, where help must be provided with the removal and hanging up of a coat. When the task invoked by the interruption has completed, the main tasks are to be resumed. This structure is captured by the description *JUNIOR*.

$$JUNIOR \;=\; TASKS \bigtriangleup x : \{ring, boss\} \rightarrow P(x)$$

The component processes might be defined as follows:

$$\begin{array}{rcl} TASKS & = & tea \rightarrow TASKS \\ & & \Box\; photocopying \rightarrow TASKS \\ & & \Box\; filing \rightarrow TASKS \end{array}$$

$$\begin{array}{rcl} P(ring) & = & message \rightarrow JUNIOR \\ P(boss) & = & remove\_coat \rightarrow hang\_coat \rightarrow JUNIOR \end{array}$$

$\Box$

EXAMPLE 3.25 The office junior has a higher level interrupt of the fire alarm sounding. If it is due to a real fire then work ceases for the day and the junior returns home. Otherwise, it is announced that it is a drill, in which case it is necessary to return to work. The *fire* event interrupts all other activity, even the interrupt handlers $P(ring)$ and $P(boss)$ if they are executing. The complete behaviour is described by $JUNIOR2$:

$$\begin{array}{lcl} JUNIOR2 & = & JUNIOR \triangle fire \to (\ real \to home \to SKIP \\ & & \qquad\qquad\qquad\quad \Box\ drill \to JUNIOR2) \end{array}$$

□

The event *fire* even interrupts the tasks the junior is performing for the boss.

## 3.5 NOTES

### Bibliographic notes

Tony Hoare's original proposal for the language of Communicating Sequential Processes appeared in [45], though that language is quite different to the current version of CSP, presented in this book. In the original language systems have a specific architecture, consisting of a parallel combination of sequential processes which have their own (private) state variables and which communicate via synchronous channels. The language may be considered as the precursor to the OCCAM programming language [51, 62]. Theoretical work on the language in the early 1980's by Brookes, Hoare and Roscoe [100, 12, 13, 14] led to the abstraction and generalization of the language to the current form of CSP, which was presented in Hoare's book [47]. Roscoe introduced a way of handling unbounded nondeterminism [101], subsequently refined by Barrett [6]. Tool support for analysis and verification of CSP processes has been provided in the form of animation [35], model-checking [33] (discussed in Appendix B), and embedding within a proof tool [17, 29, 116].

The CSP language is one of a family of process algebras—languages which focus on the communication patterns between processes, and abstract away from their internal computations. These languages all use synchronization on an atomic event as the foundation for process interaction, and all provide some way of expressing event occurrence, choice, parallel composition, abstraction, and recursion. Other languages for concurrency which developed around the same time as CSP and had an influence on its development include Milner's influential Calculus of Communicating Systems (CCS) [76, 77], and Bergstra and Klop's Algebra of Communicating Processes (ACP) [7, 5], which introduced the term *process algebra*. The ISO standard Language Of Temporal Ordering Specifications (LOTOS) [10, 52, 53] combines elements of CCS and CSP together with a language for data-types. The interface parallel originally appeared in the LOTOS language, as a hybrid of CSP's alphabetized parallel and interleaving operators. More recently the Pi-calculus [78] has been introduced. This is a process algebra in the CCS tradition based around the new concept of mobility. Many of these languages are supported by tools, see for example [19, 31, 36, 119], and the TACAS (Tools and Algorithms for the Construction and Analysis of Systems) and CAV (Computer Aided Verification) conference series.

The approach to presenting operational semantics in terms of inference rules was first introduced by Plotkin [91], and has been used extensively within the CCS tradition for language definition. An operational semantics in this form was presented for CSP in [15]. In this chapter the operational semantics has been used primarily for presentational purposes, to introduce the operators of the CSP language and give an understanding of how they behave. Other approaches use the operational rules as the basis for semantic characterizations. One such approach is that of *bisimulation* [89, 77] which considers processes to be equivalent whenever they can match the states reached by each other's transitions. The other main approach is given by *testing* [28, 43] which considers processes to be equivalent if they give the same result in any testing context; this will be discussed in subsequent chapters. It is a significant result that the denotational models discussed throughout this book yield the same equivalences as the testing approach, and can thus be thought of as characterizing testing equivalence.

## Other dialects of CSP

As well as defining recursive processes equationally in the style of this book, Hoare's [47] and Roscoe's [103] treatment of recursion make use of the CSP fixpoint operator $\mu$, so that $\mu X \bullet F(X)$ is the least fixed point of the function $F(X)$. This approach has the same expressive power as the use of recursive equations to define processes, since

$$\mu X \bullet F(X) \quad = \quad F(\mu X \bullet F(X))$$

or equivalently

$$\mu X \bullet P \quad = \quad P[\mu X \bullet P/X]$$

The $\mu$ operator allows recursive processes to be defined without the need to name them, so the process *LIGHT* of Example 1.14 can be defined simply as $\mu X \bullet on \rightarrow off \rightarrow X$. It also allows nested recursive definitions, such as

$$\mu X \bullet (a \rightarrow (\mu Y \bullet a \rightarrow X \mid b \rightarrow Y))$$

Roscoe uses a more general form of alphabet renaming of processes, by using relations between events rather than functions. If $R$ is a relation on events, then $P[\![R]\!]$ can perform an event $b$ whenever $P$ can perform some event $a$ for which $aRb$. This single operator encompasses both event renaming and inverse event renaming simply by providing the function $f$, or its inverse, as the relation $R$. The relational approach and the functional approaches are equally expressive.

The treatment of termination in this book is marginally different to that presented in [47] and in [103]. The use of the ✓ event requires certain restrictions on its occurrence to ensure

$$
\begin{array}{rcl}
\alpha(STOP) & = & \{\} \\
\alpha(a \rightarrow P) & = & \alpha(P) \cup \{a\} \\
\alpha(x : A \rightarrow P(x)) & = & \bigcup_{a \in A} \alpha(P(a)) \cup A \\
\alpha(c!v \rightarrow P) & = & \alpha(P) \cup c.T \\
\alpha(c?x : T \rightarrow P(x)) & = & \bigcup_{x \in T} \alpha(P(x)) \cup c.T \\
\alpha(SKIP) & = & \{\} \\
\alpha(P_1 \mathbin{\Box} P_2) & = & \alpha(P_1) \cup \alpha(P_2) \\
\alpha(\Box_{i \in I} P_i) & = & \bigcup_{i \in I} \alpha(P_i) \\
\alpha(P_1 \sqcap P_2) & = & \alpha(P_1) \cup \alpha(P_2) \\
\alpha(\bigsqcap_{i \in J} P_i) & = & \bigcup_{i \in J} \alpha(P_i)
\end{array}
$$

***Fig. 3.7*** Default alphabets for sequential processes

that it models termination suitably. For example, parallel combinations are always required to synchronize on ✓. Roscoe's treatment ensures that if a process can possibly terminate, then the process itself can choose to terminate and refuse all other interaction: essentially, termination is under the control of the process and cannot be prevented by its environment simply withholding ✓. Hoare's treatment achieves the same result by imposing a restriction, requiring that ✓ should never be offered as an alternative in a choice. The treatment in this book differs from each of these, in that ✓ may be offered as an alternative of a choice, and termination requires the co-operation of the environment. This allows a cleaner relationship between the untimed and the timed languages, while making little difference in practice.

## A note on alphabets

Hoare's presentation of CSP in [47] required every process definition to be associated explicitly with an alphabet, which might be thought of as its type. A process definition $P$ is not complete until its alphabet $\alpha P$ had been given. This approach makes an alphabetized parallel operator unnecessary, since in a parallel combination the interfaces are already associated with the component processes, and there is no need for the operator also to supply them.

More recent presentations of CSP have relaxed the requirement to provide alphabets with process definitions, and instead include the interface information with the parallel operator whenever it is used. The two approaches are equally expressive. In this book, the alphabet $\alpha P$ of a process $P$ is used in a less formal way, to mean the interface consisting of all of the events mentioned in the definition of the process $P$. The alphabet operator is defined in Figures 3.7 and 3.8.

Since chaining is a derived operator, its alphabet can be deduced in the general case from its definition. However, it is good practice to ensure that only processes with alphabets

$$\begin{array}{rcl}
\alpha(P_1 \; {}_A\|_B \; P_2) & = & A \cup B \\
\alpha(P_1 \parallel P_2) & = & \alpha(P_1) \cup \alpha(P_2) \\
\alpha(\|_{A_i}^{i \in I} P_i) & = & \bigcup_{i \in I} A_i \\
\alpha(\|^{i \in I} P_i) & = & \bigcup_{i \in I} \alpha(P_i) \\
\alpha(P_1 \mid\mid\mid P_2) & = & \alpha(P_1) \cup \alpha(P_2) \\
\alpha(\mid\mid\mid_{i \in I} P_i) & = & \bigcup_{i \in I} \alpha(P_i) \\
\alpha(P_1 \underset{A}{\parallel} P_2) & = & \alpha(P_1) \cup \alpha(P_2) \\
\alpha(P \setminus A) & = & \alpha(P) \setminus A \\
\alpha(f(P)) & = & f(\alpha(P)) \\
\alpha(l : P) & = & f_l(\alpha(P)) \\
\alpha(f^{-1}(P)) & = & f^{-1}(\alpha(P)) \\
\alpha(P_1 ;\; P_2) & = & \alpha(P_1) \cup \alpha(P_2) \\
\alpha(P_1 \bigtriangleup P_2) & = & \alpha(P_1) \cup \alpha(P_2)
\end{array}$$

***Fig. 3.8*** Further default alphabets

$in.T \cup out.T$ should be chained together; if this is indeed both $\alpha P_1$ and $\alpha P_2$, then it is also $\alpha(P_1 \gg P_2)$. Similarly, if it is the alphabet of all of the $P_i$, then it will also be the alphabet of the indexed chain of processes $\gg_{i-1}^{n} P_i$.

In a recursive definition $N = P$, the alphabet $\alpha(N)$ is defined to be the smallest set which makes the equation $\alpha(N) = \alpha(P)$ true.

## Exercises

EXERCISE 3.1 Give the transition graphs of the following processes

1. The process $2COPY$ of Example 3.16
2. The process $f(OFFICE)$ of Example 3.14
3. The process $f^{-1}(SALE)$ of Example 3.15

EXERCISE 3.2 Give a process which captures the generic behaviour of each cell $C_{i,j}$ in the systolic array $SORTER$, given in Example 2.12. What are the appropriate event renaming functions for instantiating the generic process to each cell? Give an alternative definition of *SORTER* as a parallel combination of these renamed components.

EXERCISE 3.3 If $A \cap B = \{\}$, then does

$$(((P_1 \underset{A}{\|} P_2) \setminus A) \underset{B}{\|} P_3) \setminus B \quad = \quad ((P_1 \underset{A}{\|} P_2) \underset{B}{\|} P_3) \setminus (A \cup B)$$

How about if $A \cap B \neq \{\}$ ?

EXERCISE 3.4 If $P$ can never reach a deadlock state, does it follow that $P \setminus A$ can never reach one?

EXERCISE 3.5 A stack with operations *push* and *pop* can be defined in the style of the process *SET* of Example 3.19, so that

$$\begin{aligned} STACK \quad = \quad & push?x : T \rightarrow (NODE(x) \| i : STACK) \setminus (i.push.T \cup i.pop.T) \\ & \Box\ pop!empty \rightarrow STACK \end{aligned}$$

Give a suitable definition of process $NODE(x)$.

EXERCISE 3.6 Give the transition graph of the process *PURCHASE* of Example 3.20

EXERCISE 3.7 Give the transition graph of the process *JUNIOR* of Example 3.24

EXERCISE 3.8 Does $P_1 \bigtriangleup (P_2 \bigtriangleup P_3)$ have the same behaviour as $(P_1 \bigtriangleup P_2) \bigtriangleup P_3$ (i.e. is the interrupt operator associative) ?

EXERCISE 3.9 A library allows readers to register, and then repeatedly to borrow and return books until deregistration is requested.

$$\begin{aligned} NONREADER \quad &= \quad register \rightarrow READER \\ READER \quad &= \quad borrow \rightarrow READER \\ & \qquad \Box\ return \rightarrow READER \\ & \qquad \Box\ deregister \rightarrow STOP \end{aligned}$$

Introduce the following constraints, in each case by means of a fresh parallel component

1. Readers cannot return more books than they have borrowed.
2. Readers are not permitted to deregister if there are any book loans outstanding.
3. Readers can borrow a maximum of three books.

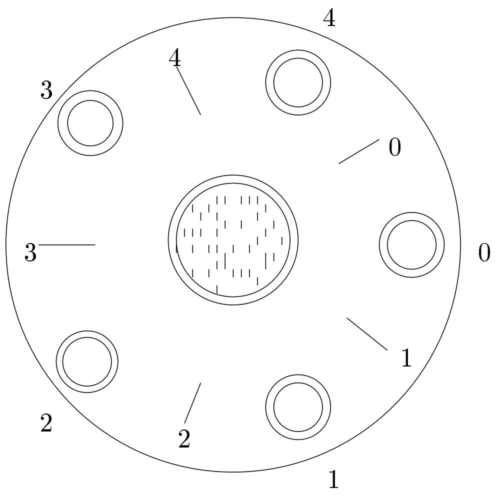


***Fig. 3.9*** The dining philosophers

4. Readers cannot deregister.

5. Readers can borrow books only when the library is open. Introduce a new component with extra events *open* and *close*.

EXERCISE $3.10$ [Dijkstra/Hoare] A college consists of five philosophers who think and eat. They eat at a circular dining table. When they need to eat, they enter the dining hall, pick up the chop-sticks on either side of their plate, eat, replace the chop-sticks, and then leave.

For convenience, the philosophers are labelled $0$ to $4$. Each philosopher picks up two chop-sticks, also labelled $0$ to $4$. Their relative positions are illustrated in Figure 3.9. The process describing the behaviour of philosopher *i* has the following interface of events:

| | |
|---|---|
| $\{$ $enter.i$ | Philosopher *i* enters the dining room |
| $eat.i$ | Philosopher *i* eats |
| $leave.i$ | Philosopher *i* leaves the dining room |
| $pick.i.i$ | Philosopher *i* picks up right-hand chop-stick |
| $pick.i.(i + 1 \mathsf{\ mod\ } 5)$ | Philosopher *i* picks up left-hand chop-stick |
| $put.i.i$ | Philosopher *i* replaces right-hand chop-stick |
| $put.i.(i + 1 \mathsf{\ mod\ } 5)$ $\}$ | Philosopher *i* replaces left-hand chop-stick *j* |

The possible events for $PHIL_i$ are described in the following recursive definition:

$$\begin{aligned} PHIL_i \quad = \quad & enter.i \rightarrow \\ & \quad ((pick.i.i \rightarrow pick.i.((i+1) \bmod 5) \rightarrow eat.i \\ & \qquad \rightarrow put.i.i \rightarrow put.i.((i+1) \bmod 5) \rightarrow leave.i \rightarrow PHIL_i) \\ & \quad \Box \\ & \quad (pick.i.((i+1) \bmod 5) \rightarrow pick.i.i \rightarrow eat.i \\ & \qquad \rightarrow put.i.((i+1) \bmod 5) \rightarrow put.i.i \rightarrow leave.i \rightarrow PHIL_i)) \end{aligned}$$

The philosophers do not synchronize on any events. Their combination can therefore be described as

$$PHILS \quad = \quad |||_{i=0}^{4} PHIL_i$$

Each chop-stick can be obtained by either of its neighbouring philosophers. The interface of a chop-stick $j$ is

$$\{pick.i.j \mid 0 \leqslant i \leqslant 4\} \cup \{put.i.j \mid 0 \leqslant i \leqslant 4\}$$

Its description is given by the following recursive definition:

$$\begin{aligned} CHOP_j \quad = \quad & pick.j.j \rightarrow put.j.j \rightarrow CHOP_j \\ & \Box\ pick.((j-1) \bmod 5).j \rightarrow put.((j-1) \bmod 5).j \rightarrow CHOP_j \end{aligned}$$

The chop-sticks do not synchronize on any event, so their combination can be described as

$$CHOPSTICKS \quad = \quad |||_{j=0}^{4} CHOP_j$$

The combination of all the components is then described by the process *COLLEGE*:

$$COLLEGE \quad = \quad PHILS \parallel CHOPSTICKS$$

1. Possession of a chop-stick by a philosopher blocks a neighbouring philosopher from acquiring that chop-stick—this provides a potential for philosophers to block other philosophers by denying them chop-sticks. How can the combination *COLLEGE* reach a deadlocked state?
2. Which of the following alterations to *COLLEGE* remove the possibility of deadlock? In each case, describe the amended system in CSP.
   (a) Requiring all philosophers to lift their left chop-stick first.

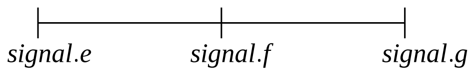


***Fig. 3.10*** Signals in a pair of segments

(b) Requiring at least one philosopher to lift his or her right chop-stick first and at least one to lift his or her left chop-stick first.

(c) Introducing a footman who allows only one philosopher to be seated at any time.

(d) Introducing a butler who prevents all from being seated simultaneously.

(e) Allowing philosophers to release the chop-stick if they hold only one.

3. Which of these guarantee that any philosopher who sits down will eventually receive something to eat?

4. Which of these guarantee that at least one seated philosopher will eventually receive something to eat?

EXERCISE 3.11 [Roscoe] A railway network imposes the safety constraint that no two trains should ever be on adjacent segments of track. A train moves from one segment of track to the next by passing through a signal. The constraint is imposed by controlling the signals so that they allow trains to pass only when it is safe to do so. An event *signal.i* will be used to describe the event of a train moving past the particular signal *i*.

For each pair of adjacent segments the signals are controlled so that trains may only enter the first segment when both segments are empty. A pair of segments will have three signals: *e* corresponding to a train entering the first segment, *f* corresponding to the train moving from the first to the second, and *g* corresponding to the train leaving the second. This is pictured in Figure 3.10.

If both segments are empty, then a train is allowed to enter the first segment, modelled by the possibility of the event *signal.e*. If the first segment is occupied, then *signal.e* is blocked, but *signal.f* can occur. If the second segment is occupied, then *signal.e* is again blocked, but *signal.g* can occur. These states are interrelated as follows:

$$\begin{aligned} \mathit{EMPTY} &= \mathit{signal.e} \to \mathit{FIRST} \\ \mathit{FIRST} &= \mathit{signal.f} \to \mathit{SECOND} \\ \mathit{SECOND} &= \mathit{signal.g} \to \mathit{EMPTY} \end{aligned}$$

The safety constraint is imposed by the process *CON* which participates in the control of these three signals by moving between these three states. If the segments are initially empty, then the signal controller *CON* is defined by $\mathit{CON} = \mathit{EMPTY}$. If there is a train initially on the first segment, then $\mathit{CON} = \mathit{FIRST}$. If there is a train initially on the second segment, then $\mathit{CON} = \mathit{SECOND}$. The process *CON* follows the same cycle in all cases, but the point where it starts depends on the presence or otherwise of a train.

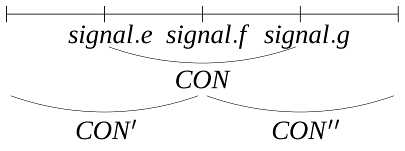


***Fig. 3.11*** Three pairs of adjacent segments

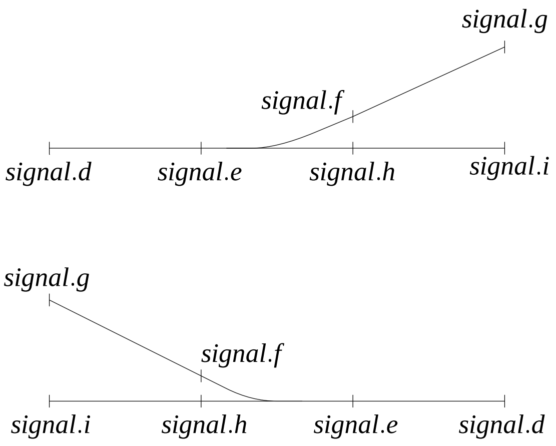


***Fig. 3.12*** Points sections of track

A pair of segments is part of a larger system in which each segment is half of another pair, as illustrated in Figure 3.11. The safety property must also hold for these pairs, and will be imposed for them by their own controller processes. Each of them is also involved in the event *signal.f*, since it corresponds to a train leaving the left-hand pair, and entering the right-hand pair. The event *signal.f* therefore requires the participation of three processes.

1. The constraints on the track may be described as a parallel combination of renamed copies of the generic process *CON*. Describe the conjunction of all the constraints on a circular track of 100 segments, with signals numbered from 0 to 99, and 10 trains initially spaced evenly around the track (traveling in the same direction).

2. Is the system you have described free from any potential deadlock?

3. What is the maximum number of trains for which the system is deadlock-free?

4. How does the CSP system behave if two trains are initially on adjacent segments? Can you improve on the description of *CON* so that it handles this case more satisfactorily?

5. Describe the constraints required to deal with points segments of the form pictured in Figure 3.12.

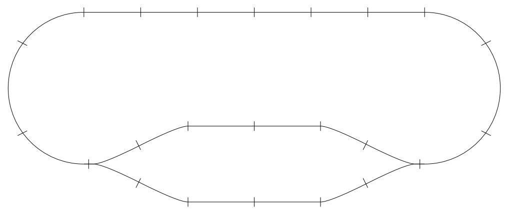


***Fig. 3.13*** A circuit with junctions

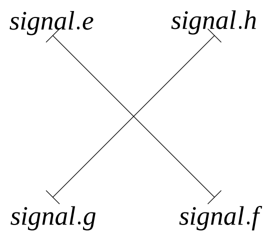


***Fig. 3.14*** A crossover segment

6. Describe the network pictured in Figure 3.13. Is it deadlock-free if two trains run on it?

7. Describe the constraints required to deal with crossover segments of Figure 3.14.

8. Alter the description of constraints to deal with (single track) bi-directional segments which trains can traverse in either direction (though trains cannot reverse their direction of travel). Also do this for points and crossover segments.

9. Describe the network (with bi-directional segments) pictured in Figure 3.15, with the trains initially moving in the same direction. Is it deadlock-free if two trains run on it?

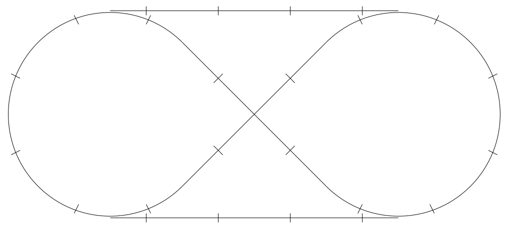


***Fig. 3.15*** A bi-directional track, with junctions and a crossover

*Part II*

# *Analyzing Processes*

# 4 Traces

At the level of abstraction provided by CSP, processes interact with their environment through the performance of events in their interface. Their environment, whether it is another process, a user, or a combination of these, has no direct access to the internal state of the process or to the internal events that it performs. Two processes which are indistinguishable at their interfaces should be equally appropriate for any particular purpose; the way they are implemented cannot have any influence on their respective suitability.

There are a number of ways in which interface behaviour can be analyzed, but they all concentrate exclusively on the external activity of the process. One important aspect of process behaviour concerns the occurrence of events in the right order, and that events do not occur at inappropriate points. The kind of sequence which is acceptable will be given by the requirements on the system. Such requirements will describe constraints on when particular events can occur. The environment of the process cannot know precisely which internal state the process has reached at any particular point, since it has access only to the projection of the execution onto the interface.

EXAMPLE 4.1 A safety requirement on the railway crossing controller in Example 1.16 might specify that the gate should not rise between the train entering the crossing, and the train leaving the crossing. This can be expressed in terms of the events in the interface: if $train.enter$ is the most recently observed of the two events $train.enter$ and $train.leave$, then the event $gate.raise$ should not occur. Its occurrence at any such point is undesirable in any execution. □

To analyze processes with respect to these requirements, it is necessary to consider those sequences of events that can be observed at the interface of the process. These observations are called *traces*, and the set of all possible traces of a process $P$ is denoted '$\mathsf{traces}(P)$'.

Trace information is concerned with those events that could possibly occur in a process execution. One might imagine an observer watching a process execute, and recording all events in sequence as they are observed. A trace is simply a record of events in the order they occur. The set of traces of a process is the set of all sequences that might possibly be recorded.

To be sure that a process does not violate a trace specification, it is necessary to examine all of its traces and check that each is acceptable.

## Notation for sequences

Sequences may be described explicitly, by listing their elements in order between angled brackets. The empty sequence is thus denoted $\langle\rangle$. If $A$ is a set, then $A^*$ is the set of all finite sequences of elements of $A$. For example, $\langle a, b, a\rangle \in \{a, b, c\}^*$.

If $seq_1$ and $seq_2$ are both sequences, then their *concatenation* described by $seq_1 \frown seq_2$ is the sequence of elements in $seq_1$ followed by those in $seq_2$. The concatenation operation is associative. The notation $seq^n$ describes $n$ copies of the finite sequence $seq$ concatenated together, and so $seq^0$ is always the empty sequence $\langle\rangle$.

If $seq$ is a non-empty sequence, then it may be written $\langle a\rangle \frown seq'$ where $a$ is the first element of $seq$, and $seq'$ is the remainder of the sequence. In this case, two functions on $seq$ are defined: $head(seq) = a$ and $tail(seq) = seq'$. Similarly, if $seq$ is non-empty and finite, then it may also be written as $seq'' \frown \langle b\rangle$, where $b$ is the final element of the sequence $seq$ and $seq''$ is the sequence of all the elements before it. Two further functions are defined: $foot(seq) = b$ and $init(seq) = seq''$.

The length $\#seq$ of a sequence is the number of elements it contains. For example, $\#\langle a, b, a\rangle = 3$.

The notation $a$ **in** $seq$ means that the element $a$ appears in the sequence $seq$, and $\sigma(seq)$ is the set of all elements that appear in $seq$.

Relationships between sequences are easily expressed. If there is some sequence $seq_2$ such that $seq \frown seq_2 = seq_1$, then $seq$ is a prefix of $seq_1$, written $seq \leqslant seq_1$. Furthermore, $seq \leqslant_n seq_1$ means that $seq \leqslant seq_1$ and their lengths differ by no more than $n$. If $seq \neq seq_1$ then $seq$ is a strict prefix of $seq_1$, written $seq < seq_1$. The notation $seq \preccurlyeq seq_1$ means that $seq$ is a (not necessarily contiguous) subsequence of $seq_1$.

For example,

$$\begin{array}{rcl}
\langle a, b, d\rangle & \preccurlyeq & \langle a, c, b, a, d\rangle \\
\langle b, c, d\rangle & \not\preccurlyeq & \langle a, c, b, a, d\rangle \\
\langle a, c, b, a\rangle & \leqslant & \langle a, c, b, a, d\rangle
\end{array}$$

The projection of a sequence $seq$ onto elements of a set $A$ is written $seq \upharpoonright A$: it is the subsequence of all elements of $seq$ that are in the set $A$. Conversely, the notation $seq \setminus A$ is the subsequence of *seq* whose elements are not in $A$. For example, $\langle a, b, c, a\rangle \upharpoonright \{a, b\} = \langle a, b, a\rangle$, and $\langle a, b, c, a\rangle \setminus \{a, b\} = \langle c\rangle$. If $f$ is a mapping on elements, then $f(seq)$ is the sequence obtained by applying $f$ to each element of *seq* in turn.

These functions can be used to extract information from sequences. For instance, the value of $\#(seq \upharpoonright A)$ gives the number of occurrences of events from $A$ in *seq*. This will be abbreviated $seq \downarrow A$. In the case where $A$ is a singleton set $\{a\}$, the set brackets will be elided and $seq \downarrow a$ will abbreviate $seq \downarrow \{a\}$. Similarly, $seq \upharpoonright a$ will abbreviate $seq \upharpoonright \{a\}$.

## Notation for traces

Traces are simply a particular class of finite sequences of events drawn from $\Sigma^{\checkmark}$ which represent executions. Since events in a process' execution cannot occur after termination, any termination event $\checkmark$ occurring in a trace must appear at the end. The set of all such traces is defined as *TRACE*.

$$TRACE \quad = \quad \{tr \mid \ \sigma(tr) \subseteq \Sigma^{\checkmark} \wedge \#tr \in \mathbb{N} \wedge \checkmark \notin \sigma(init(tr))\}$$

The sequence $\langle a, c, b, a, d\rangle$ is a record of an execution where the events $a$, $c$, $b$, $a$, $d$ occurred in that order. The empty sequence $\langle\rangle$ corresponds to an execution in which no events were observed.

Since all traces are sequences, they inherit all of the sequence operators. These all yield a trace when applied to traces, apart from sequence concatenation (and hence repeated concatenation) and mapping through a function.

However, sequence concatenation does map traces $tr_1$ and $tr_2$ to a trace $tr_1 {}^\frown tr_2$ provided $\checkmark \notin \sigma(tr_1)$. Thus $tr^n$ will be a trace if $\checkmark \notin \sigma(tr)$.

Furthermore, if a function $f$ maps $\Sigma$ into $\Sigma$ and $f(\checkmark) = \checkmark$, then $f(tr)$ will always be a trace.

Events appearing in traces will often be of the form $c.v$ corresponding to a communication of a value $v$ along channel $c$. In this case the following projections may be defined, where $c$ is not of the form $x.y$ for any $x$ and $y$:

$$\begin{aligned} \mathsf{channel}(c.v) &= c \\ \mathsf{value}(c.v) &= v \end{aligned}$$

The channels appearing in a trace $tr$ can then be extracted:

$$\mathsf{channels}(tr) \quad = \quad \{\mathsf{channel}(x) \mid x \ \mathbf{in} \ tr\}$$

The sequence of values appearing on a channel $c$ in a trace $tr$ can also be extracted:

$$tr \Downarrow c \quad = \quad \langle \mathsf{value}(x) \mid x \leftarrow tr, \mathsf{channel}(x) = c\rangle$$

This sequence comprehension describes the sequence of values of items appearing on channel $c$. It also generalizes to sets of channels $C$:

$$tr \Downarrow C \quad = \quad \langle \mathsf{value}(x) \mid x \leftarrow tr, \mathsf{channel}(x) \in C \rangle$$

For example, if $tr = \langle in.3, in.6, out.3, in.7, in.9, out.6 \rangle$ then $tr \Downarrow in = \langle 3, 6, 7, 9 \rangle$, and $\mathsf{channels}(tr) = \{in, out\}$.

## Traces and executions

The transition rules for CSP define those executions that are possible for processes. Trace information can be extracted from these executions by ignoring the intermediate states and internal transitions, and considering only the visible transitions. A trace is a record of the visible events of an execution.

The notation $P \overset{tr}{\Longrightarrow} P'$ means that there is a sequence of transitions whose initial process is $P$ and whose final process is $P'$, and whose visible transitions constitute the sequence $tr$. Since termination can occur only at the end of an execution, if $\checkmark$ occurs in $tr$ then it must be at the end. In this case, $P'$ will have no transitions.

The final process may be dropped in cases where it is not required: the notation $P \overset{tr}{\Longrightarrow}$ is used as shorthand for $\exists P' \bullet P \overset{tr}{\Longrightarrow} P'$.

The *traces* of a process may then be defined in terms of the sequences of events that may be exhibited by that process:

$$\mathsf{traces}(P) \quad = \quad \{tr \mid P \overset{tr}{\Longrightarrow} \}$$

EXAMPLE 4.2 The process $a \rightarrow ((b \rightarrow STOP) \sqcap (c \rightarrow d \rightarrow STOP))$ has the trace $\langle a, c \rangle$ as one of its traces. This may be extracted from the following execution:

$$\begin{array}{l}
a \rightarrow ((b \rightarrow STOP) \sqcap (c \rightarrow d \rightarrow STOP)) \\
\quad \downarrow a \\
((b \rightarrow STOP) \sqcap (c \rightarrow d \rightarrow STOP)) \\
\quad \downarrow \tau \\
(c \rightarrow d \rightarrow STOP) \\
\quad \downarrow c \\
d \rightarrow STOP
\end{array}$$

□

EXAMPLE 4.3 The process $STOP$ has no transitions, and hence only one execution, in which it forever remains in the same state. The trace corresponding to this trace will be the empty trace. □

## 4.1 TRACE SEMANTICS

The extraction of trace information from the process transition rules provides an explanation of the relationship between the executions of a process and its traces. However, the operational characterization is too low level for reasoning about processes, since the level of abstraction remains that of process executions, with the set of traces supervenient. The *traces model* for CSP considers processes directly in terms of their traces, and lifts the entire analysis of CSP processes to this more abstract level. All of the operators of the language can be understood at this level: the traces of a composite process are dependent only on the traces of its components. This allows a *compositional* semantic model, where all processes are considered only in terms of their sets of traces, and at no stage do the underlying executions need to be considered explicitly.

In the traces model, each CSP process is associated with a set of traces—the set of all possible sequences of events that may be observed of some execution. Processes will be *trace equivalent* when they have exactly the same set of possible traces. This particular form of equality will be denoted $=_T$, and its definition is that

$$P_1 =_T P_2 \quad = \quad \mathsf{traces}(P_1) = \mathsf{traces}(P_2)$$

In the traces model, processes are equal when they have exactly the same traces. Traces equality gives rise to algebraic laws for individual operators, and also concerning the relationships between various operators. These laws allow manipulation of CSP process descriptions from one form to another while keeping the associated set of traces unchanged. Many laws are concerned with general algebraic properties such as associativity and commutativity of operators (which allow components to be composed in any order), idempotence, and the identification of units and zeros for particular operators (which may allow process descriptions to be simplified). Other laws are concerned with the relationships between different operators, which allow for example the expansion of a parallel combination into a prefix choice process.

In Chapters 6, 8 and 11 more detailed views of process executions will be used to characterize processes in different ways. Some process laws may be concerned only with traces, but others may be true in any of these models. If a law holds in any of these models, as in fact most of those given in this chapter will, then the subscript will be dropped from the equality. Hence $P_1 = P_2$ means not only that $P_1 =_T P_2$, but also that $P_1 =_{SF} P_2$, $P_1 =_{FDI} P_2$ and $P_1 =_{TF} P_2$, corresponding to the equalities that will be defined later, under the more detailed views (stable failures, failures/divergences/infinite traces and timed failures) given in Chapters 6, 8 and 11 respectively. If a law is valid in all of the untimed models, then the equality symbol will be subscripted with a $U$. For example, the associativity of external choice is true in all models, since the executions of $P_1 \Box (P_2 \Box P_3)$ match those of $(P_1 \Box P_2) \Box P_3$, so all views of executions of these processes, no matter how detailed, will not distinguish them. The fact that it will be true in any of these models is indicated by the lack of a subscript on the equality symbol.

$$P_1 \Box (P_2 \Box P_3) \quad = \quad (P_1 \Box P_2) \Box P_3$$

On the other hand, although the traces of $P_1 \mathbin{\Box} P_2$ and $P_1 \sqcap P_2$ will be the same, a more sophisticated view of process executions will distinguish them. This law will be written as

$$P_1 \mathbin{\Box} P_2 \quad =_T \quad P_1 \sqcap P_2$$

since it is true only in the traces model.

Any set of traces $S$ associated with some process must contain the empty trace: any process can be observed to do nothing. It will also be prefix closed: if a process can perform a sequence of events, then it can also be observed to perform any prefix of that sequence. These properties are formalized as $T1$ and $T2$ on set $S$:

$$\begin{array}{ll} T1 & \langle\rangle \in S \\ T2 & \forall\, tr_1, tr_2 : \mathit{TRACE} \bullet (tr_1 \leqslant tr_2 \wedge tr_2 \in S \Rightarrow tr_1 \in S) \end{array}$$

## *STOP*

There is only one trace associated with the process *STOP*, and that is the empty trace. The semantics of *STOP* is given directly as

$$\mathsf{traces}(\mathit{STOP}) \quad = \quad \{\langle\rangle\}$$

### Prefixing

In an observation of the process $a \rightarrow P$, there are two possibilities: either the event $a$ has not occurred, in which case the observation must be $\langle\rangle$, or else the event $a$ has occurred and the rest of the trace derives from process $P$.

$$\begin{array}{lll} \mathsf{traces}(a \rightarrow P) & = & \{\langle\rangle\} \\ & & \cup \\ & & \{\langle a\rangle \frown tr \mid tr \in \mathsf{traces}(P)\} \end{array}$$

### Prefix choice

An observation of the process $x : A \rightarrow P(x)$ is again one of two possibilities. Either no event has yet occurred, or else an event $a$ in $A$ has occurred, and the subsequent behaviour is that of the corresponding process $P(a)$.

$$\begin{array}{lll} \mathsf{traces}(x : A \rightarrow P(x)) & = & \{\langle\rangle\} \\ & & \cup \\ & & \{\langle a\rangle \frown tr \mid a \in A \wedge tr \in \mathsf{traces}(P(a))\} \end{array}$$

| | |
|---|---|
| $x : \{\} \to P(x) = STOP$ | $\langle STOP\text{-step}\rangle$ |
| $x : \{b\} \to P(x) = b \to P(b)$ | $\langle \text{prefix} \rangle$ |

***Fig. 4.1*** Laws for prefix choice

EXAMPLE 4.4 The process *BUS_1* of Example 1.24 is described as follows:

$$\begin{array}{lcl} BUS\_1 & = & board.A \to (\ pay.90 \to alight.B \to STOP \\ & & \qquad\qquad\quad \mid alight.A \to STOP) \end{array}$$

This process has the following traces:

$$\begin{array}{lcl} \text{traces}(BUS\_1) & = & \{\ \langle\rangle, \\ & & \ \ \langle board.A \rangle, \\ & & \ \ \langle board.A, pay.90 \rangle, \\ & & \ \ \langle board.A, pay.90, alight.B \rangle, \\ & & \ \ \langle board.A, alight.A \rangle \} \end{array}$$

It initially allows *board*.*A*, after which either the fare is paid and the journey made, or else the journey is not made and the passenger alights again. □

The definition of $\text{traces}(x : A \to P(x))$ has two special cases: where $A$ contains no elements ($A = \{\}$) and where $A$ contains but a single element ($A = \{b\}$).

In the case where $A = \{\}$, the second clause of the definition cannot be met, since there is no event $a$ for which $a \in A$. The semantics is thus equal to $\{\langle\rangle\}$, which is the semantics of *STOP*. In the case where $A = \{b\}$, the second clause of the definition is equivalent to

$$\{\langle b \rangle \frown tr \mid tr \in \text{traces}(P(b))\}$$

which is the second clause of the event prefix definition for $b \to P(b)$. These observations support two *laws* concerning equality of process expressions, given in Figure 4.1.

## Output and input

The output and input constructors are special cases of the prefix and prefix choice operators. The definition of their trace semantics follows the same pattern.

$$\begin{array}{lcl} \text{traces}(c!v \to P) & = & \{\langle\rangle\} \\ & & \cup\{\langle c.v \rangle \frown tr \mid tr \in \text{traces}(P)\} \end{array}$$

$$\begin{array}{rcl}\mathsf{traces}(c?m : T \to P(m)) & = & \{\langle\rangle\} \\ & & \cup\{\langle c.v\rangle \frown tr \mid v \in T \wedge tr \in \mathsf{traces}(P(v))\}\end{array}$$

EXAMPLE 4.5 The traces of the process $in?x : \mathbb{Z} \to out!x \to STOP$ are given as follows:

$$\begin{array}{rcl}\mathsf{traces}(in?x : \mathbb{Z} \to out!x \to STOP) & = & \{\langle\rangle\} \\ & & \cup \{\langle in.v\rangle \mid v \in \mathbb{Z}\} \\ & & \cup \{\langle in.v, out.v\rangle \mid v \in \mathbb{Z}\}\end{array}$$

An observation of this process might contain no events, or a single input, or an input of a particular value followed by output of that same value. $\Box$

## *SKIP*

The atomic process *SKIP* is used to denote successful termination, and it signals this by means of the termination event ✓, the only event it can perform. The only traces it exhibits are the empty trace and the singleton trace containing ✓.

$$\mathsf{traces}(SKIP) = \{\langle\rangle, \langle\checkmark\rangle\}$$

## *RUN*

The CSP operators describing choice and concurrency exhibit a number of useful laws on processes. A particular process which interacts well with them is the process *RUN*, defined to be the process which can do any sequence of events. It may be defined directly in the traces model as follows:

$$\mathsf{traces}(RUN) = \{tr \mid tr \in TRACE\}$$

The semantics of this process consists of all possible traces. It is the most obliging process, always willing to perform any event. It may also be recursively defined using the existing operators of the language, as follows:

$$RUN = (x : \Sigma \to RUN) \Box SKIP$$

This process may also be defined with a particular interface $A \subseteq \Sigma$. The process $RUN_A$ is defined to be the process with interface $A$ that can always perform any event in its interface. Its trace set is given as follows:

$$\mathsf{traces}(RUN_A) = \{tr \mid tr \in TRACE \wedge \sigma(tr) \subseteq A\}$$

$$
\begin{array}{ll}
P \mathbin{\Box} P = P & \langle \Box\text{-idem} \rangle \\
P_1 \mathbin{\Box} (P_2 \mathbin{\Box} P_3) = (P_1 \mathbin{\Box} P_2) \mathbin{\Box} P_3 & \langle \Box\text{-assoc} \rangle \\
P_1 \mathbin{\Box} P_2 = P_2 \mathbin{\Box} P_1 & \langle \Box\text{-sym} \rangle \\
P \mathbin{\Box} STOP = P & \langle \Box\text{-unit} \rangle \\
P \mathbin{\Box} RUN =_T RUN & \langle \Box\text{-zero}_T \rangle \\
x : A \to P_1(x) \mathbin{\Box} y : B \to P_2(y) & \langle \Box\text{-step} \rangle \\
\quad = \; z : A \cup B \to R(z) \text{where} & \\
\qquad \begin{array}{lll} R(c) & = P_1(c) & \text{if } c \in A \setminus B \\ & = P_2(c) & \text{if } c \in B \setminus A \\ & = P_1(c) \sqcap P_2(c) & \text{if } c \in A \cap B \end{array} &
\end{array}
$$

***Fig. 4.2*** Laws for external choice

or it might alternatively be defined recursively using choice constructs:

$$RUN_A \quad = \quad x : A \to RUN_A$$

$RUN_{A\checkmark}$ behaves as $RUN_A$ but it can also terminate:

$$RUN_{A\checkmark} \quad = \quad (x : A \to RUN_A) \mathbin{\Box} SKIP$$

The process $RUN$ defined above is equivalent to $RUN_{\Sigma\checkmark}$.

## External choice

An observer of the choice construct $P_1 \mathbin{\Box} P_2$ might observe an execution of $P_1$, or of $P_2$; there are no other possibilities. The possible traces of the choice consists of the union of the two sets of traces:

$$\text{traces}(P_1 \mathbin{\Box} P_2) \quad = \quad \text{traces}(P_1) \cup \text{traces}(P_2)$$

The treatment of external choice as the union of trace sets means that the operator inherits the properties of the union operator, in particular idempotence, associativity, and commutativity, as given in the first three laws of Figure 4.2.

The first of these laws, $\Box$-idem, states that offering a choice between two copies of the same process is not actually offering a choice at all. The second and third laws allow larger sets of choices to be rearranged without altering the trace possibilities. These are the laws that guarantee that the definition of the indexed choice operator is well-defined. They allow a choice of processes to be defined purely in terms of the set of choices.

Law $\Box$-unit states that external choice gives any process $P$ precedence over $STOP$, which can never resolve a choice in its favour. Law $\Box$-zero$_T$ states that external choice allows any process $P$ to be masked by $RUN$: in a choice with $RUN$, if the choice does happen to be resolved in favour of $P$, then any trace corresponding to such an execution of $P$ is also possible for $RUN$. Thus every trace of $P \ \Box \ RUN$ is a trace of $RUN$, and the presence of $RUN$ as an alternative masks the executions of $P$. In algebraic terms, $STOP$ is a unit of external choice, and $RUN$ is a zero.

Finally, an external choice of two menu choices may be rewritten as a single menu choice. Law $\Box$-step gives the correspondence. The events that are on offer in the menu choice must consist of all events that are on offer in one or other of the two component menu choices. If an event is chosen which was offered only by one component, then the subsequent behaviour must be determined by that component. If the chosen event was actually offered by both components, then the choice as to which one is subsequently executed is made internally—the environment could choose the event, but cannot choose which subsequent behaviour will arise.

EXAMPLE 4.6 The choice between the two buses given in Example 1.24 is the choice $BUS\_1 \ \Box \ BUS\_2$ where

$$\begin{array}{lcl} BUS\_1 & = & board.A \rightarrow (\ pay.90 \rightarrow alight.B \rightarrow STOP \\ & & \qquad\qquad\quad \mid alight.A \rightarrow STOP) \end{array}$$

$$\begin{array}{lcl} BUS\_2 & = & board.A \rightarrow (pay.70 \rightarrow alight.B \rightarrow STOP \\ & & \qquad\qquad\quad \mid alight.A \rightarrow STOP) \end{array}$$

Law $\Box$-step can be applied to this choice of processes. Since both components have the same single first event, the choice reduces as follows:

$$\begin{array}{lcll} BUS\_1 \ \Box \ BUS\_2 & =_T & board.A \rightarrow & ((pay.90 \rightarrow alight.B \rightarrow STOP \\ & & & \mid alight.A \rightarrow STOP) \\ & & & \sqcap \\ & & & (pay.70 \rightarrow alight.B \rightarrow STOP \\ & & & \mid alight.A \rightarrow STOP)) \end{array}$$

This equivalence reflects the fact that the environment has no control over which of two copies of the same event is actually chosen; the choice is instead resolved internally when the event is chosen. $\Box$

EXAMPLE 4.7 Process $P_1$ offers a choice between events $a$ and $b$, and $P_2$ offers a choice between $b$ and $c$, as follows:

$$\begin{array}{lcl} P_1 & = & a \rightarrow d \rightarrow STOP \\ & & \mid b \rightarrow e \rightarrow STOP \end{array}$$

$$\begin{array}{lcl} P_2 & = & b \rightarrow f \rightarrow STOP \\ & & \mid c \rightarrow g \rightarrow STOP \end{array}$$

$$\Box_{i \in \{\}} P_i = STOP \qquad \langle \Box\text{-unit} \rangle$$

$$\Box_{i \in I}(x : A_i \to P_i(x)) = x : (\textstyle\bigcup_{i \in I} A_i) \to \sqcap_{\{i \mid x \in A_i\}} P_i(x) \qquad \langle \Box\text{-step} \rangle$$

***Fig. 4.3*** Laws for indexed external choice

The external choice $P_1 \Box P_2$ between $P_1$ and $P_2$ offers a choice between the events $a$, $b$, and $c$. The environment may choose between these events, but this is the extent of its control over subsequent behaviour.

$$\begin{aligned} P_1 \Box P_2 \quad =_T \quad & a \to d \to STOP \\ & \mid b \to (\ e \to STOP \\ & \qquad\quad \sqcap f \to STOP) \\ & \mid c \to g \to STOP \end{aligned}$$

If $b$ is chosen, then the next event could be either $e$ or $f$, and the choice between them will be made internally by the process $P_1 \Box P_2$. □

The executions of the indexed external choice $\Box_{i \in I} P_i$ are the executions of all of its components. Its traces are given as follows:

$$\text{traces}(\Box_{i \in I} P_i) \quad = \quad \textstyle\bigcup_{i \in I} \text{traces}(P_i) \cup \{\langle\rangle\}$$

The explicit inclusion of the empty trace $\langle\rangle$ is required in the case when $I$ is the empty set. When the set $I$ is non-empty, then inclusion of $\langle\rangle$ is redundant since it will be included in any of the trace sets $\text{traces}(P_i)$.

There are two laws particular to indexed external choice. They are given in Figure 4.3. The first law states that an empty indexed choice is a process that can do nothing. The second law is a generalization of Law $\Box$-step. It states that a indexed external choice of prefix choices is equivalent to a single prefix choice over all the possible first events (i.e. the union of all the component choice sets). The process subsequent to a given $x$ is any of the corresponding processes $P_i(x)$ from one of the prefix choices which offered $x$ (i.e. for which $x \in A_i$).

## Internal choice

The internal choice $P_1 \sqcap P_2$ behaves either as $P_1$ or as $P_2$, and its environment exercises no control over which. A recorder of traces is concerned only with the executions that are possible, and these are the executions of $P_1$ and of $P_2$. The traces of $P_1 \sqcap P_2$ are therefore

$$\text{traces}(P_1 \sqcap P_2) \quad = \quad \text{traces}(P_1) \cup \text{traces}(P_2)$$

| | |
|---|---|
| $P \sqcap P = P$ | $\langle\sqcap\text{-idem}\rangle$ |
| $P_1 \sqcap (P_2 \sqcap P_3) = (P_1 \sqcap P_2) \sqcap P_3$ | $\langle\sqcap\text{-assoc}\rangle$ |
| $P_1 \sqcap P_2 = P_2 \sqcap P_1$ | $\langle\sqcap\text{-sym}\rangle$ |
| $P_1 \sqcap P_2 =_T P_1 \Box P_2$ | $\langle\text{choice-equiv}_T\rangle$ |

***Fig. 4.4*** Laws for internal choice

This form of choice has different executions to the external choice $P_1 \Box P_2$, since the choice is first resolved by an internal $\tau$ transition before the appropriate choice begins execution. However, this internal transition is not recorded in any trace, and a trace observer is not concerned with identifying where responsibility lies for particular choices, but only with the possible sequences of events. Under these circumstances, the internal and external choice constructs are not distinguished. Both exhibit precisely the same possible sequences of visible events, and their trace semantics are identical. Examination of a process' set of traces is not adequate for detecting the presence or absence of nondeterminism. More detailed observations are required to distinguish internal from external choice, and these will be introduced in Chapter 6.

Since they are currently treated the same way, the internal choice operator satisfies the same laws as the external choice operator, though only in the traces model. The useful laws are given in Figure 4.4.

The indexed internal choice $\bigsqcap_{i \in J} P_i$ (where the indexing set $J$ must be non-empty) is able to behave as any of its component processes. Its traces will therefore be the indexed union of the traces of all of its constituents:

$$\text{traces}(\bigsqcap_{i \in J} P_i) \quad = \quad \bigcup_{i \in J} \text{traces}(P_i)$$

## Alphabetized parallel

A parallel combination $P_1 \;{}_A\|_B\; P_2$ consists of $P_1$ performing events in $A$, and $P_2$ performing events in $B$. Processes $P_1$ and $P_2$ synchronize on events in $A \cap B$, and perform their other events independently.

Since $P_1$ is involved in the performance of all events from $A$, any execution of the parallel combination projected onto $A$ must be an execution of $P_1$. Similarly, any execution projected onto $B$ must be an execution of $P_2$. The traces of $P_1 \;{}_A\|_B\; P_2$ are those sequences of events which are consistent with both $P_1$ and $P_2$. Only events in $A$ or $B$, or termination, can be performed, so the set of events in the trace must be contained in $(A \cup B)^{\checkmark}$:

$$\begin{array}{rcll} \mathsf{traces}(P_1 \;{}_A\|_B\; P_2) & = & \{tr \in TRACE \mid & tr \upharpoonright A^{\checkmark} \in \mathsf{traces}(P_1) \\ & & & \wedge\; tr \upharpoonright B^{\checkmark} \in \mathsf{traces}(P_2) \\ & & & \wedge\; \sigma(tr) \subseteq (A \cup B)^{\checkmark} \quad \} \end{array}$$

EXAMPLE 4.8 The traces of $P_1 = a \to STOP$ are given by $\mathsf{traces}(P_1) = \{\langle\rangle, \langle a\rangle\}$. Similarly, the traces of $P_2 = b \to STOP$ are given by $\mathsf{traces}(P_2) = \{\langle\rangle, \langle b\rangle\}$. A trace of $P_1 \;{}_{\{a\}}\|_{\{b\}}\; P_2$ must be a sequence of events from $\{a, b, \checkmark\}$ whose projection to $\{a, \checkmark\}$ is either $\langle\rangle$ or $\langle a\rangle$, and whose projection to $\{b, \checkmark\}$ is either $\langle\rangle$ or $\langle b\rangle$. There are five such traces:

$$\{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a, b\rangle, \langle b, a\rangle\}$$

and so this set is $\mathsf{traces}(P_1 \;{}_{\{a\}}\|_{\{b\}}\; P_2)$. □

EXAMPLE 4.9 The traces of $P_1 = a \to b \to STOP$ are given as $\mathsf{traces}(P_1) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$. Similarly, the traces of $P_2 = b \to c \to STOP$ are given by $\mathsf{traces}(P_2) = \{\langle\rangle, \langle b\rangle, \langle b, c\rangle\}$. A trace of $P_1 \;{}_{\{a,b\}}\|_{\{b,c\}}\; P_2$ must be a sequence of events from $\{a, b, c, \checkmark\}$ whose projection to $\{a, b, \checkmark\}$ is either $\langle\rangle$, $\langle a\rangle$, or $\langle a, b\rangle$, and whose projection to $\{b, c, \checkmark\}$ is either $\langle\rangle$, $\langle b\rangle$, or $\langle b, c\rangle$. The existence of the event $b$ in the interfaces of both $P_1$ and $P_2$ means that both processes have control over its occurrence, and its appearance in any trace must be consistent with both components. The process $P_1$ forces event $b$ to occur after the event $a$, and $P_2$ forces $b$ to occur before $c$. The set of traces consistent with both processes is therefore

$$\{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle\}$$

and so this set is $\mathsf{traces}(P_1 \;{}_{\{a,b\}}\|_{\{b,c\}}\; P_2)$. □

There are a number of trace laws concerning the parallel operator. These are listed in Figure 4.5.

Law $\|$-idem is a form of idempotence: if the interface $A$ provided for $P$ allows all of its possible events—$\alpha(P) \subseteq A$—then the traces of $P$ are the same as the traces of two copies of $P$ running together. Any execution of $P$ can be performed by both copies of $P$ executing together synchronizing on every event. Laws $\|$-assoc and $\|$-sym are the associativity and commutativity laws for the parallel operator. The intermediate interfaces in law $\|$-assoc depend on the order in which components are composed together, but the resulting process is the same in each case. Law $\|$-unit provides a unit for the parallel operator: the process $RUN_{(A \cap B)^{\checkmark}}$ which is always prepared to perform any event in the common interface, and hence places no restriction on $P$'s performance of those events. The construction of the interfaces means that the process $P$ is not prevented from performing events in $A \setminus B$ either, so $P$ in $P \;{}_A\|_B\; RUN_{(A \cap B)^{\checkmark}}$ is able to perform any of its executions, and the resulting process behaves exactly as $P$.

$P \;{}_A\|_A\; P =_T P \quad \text{if } \alpha(P) \subseteq A$ ⟨‖-idem$_T$⟩

$P_1 \;{}_A\|_B\; P_2 = P_2 \;{}_B\|_A\; P_1$ ⟨‖-sym⟩

$P_1 \;{}_A\|_{B\cup C}\; (P_2 \;{}_B\|_C\; P_3) = (P_1 \;{}_A\|_B\; P_2) \;{}_{A\cup B}\|_C\; P_3$ ⟨‖-assoc⟩

$P \;{}_A\|_B\; RUN_{(A\cap B)^{\checkmark}} = P \quad \text{if } \alpha(P) \subseteq A$ ⟨‖-unit⟩

$C \subseteq A \wedge D \subseteq B \Rightarrow$ ⟨‖-step⟩

$(x : C \to P_1(x)) \;{}_A\|_B\; (y : D \to P_2(y))$

$\quad = z : ((C \setminus B) \cup (D \setminus A) \cup (C \cap D)) \to R(z)$

where

$$\begin{aligned} R(c) &= P_1(c) \;{}_A\|_B\; (y : D \to P_2(y)) && \text{if } c \in C \setminus B \\ &= (x : C \to P_1(x)) \;{}_A\|_B\; P_2(c) && \text{if } c \in D \setminus A \\ &= P_1(c) \;{}_A\|_B\; P_2(c) && \text{if } c \in C \cap D \end{aligned}$$

$SKIP \;{}_A\|_B\; SKIP = SKIP$ ⟨‖-term 1⟩

$(x : C \to P(x)) \;{}_A\|_B\; SKIP = x : C \cap (A \setminus B) \to (P(x) \;{}_A\|_B\; SKIP)$ ⟨‖-term 2⟩

***Fig. 4.5*** Laws for alphabetized parallel

Law ‖-step shows how to reduce a parallel combination of prefix choices to a single prefix choice. The events that are initially possible are those that either side can perform without the co-operation of the other, together with those that both are initially ready to perform. The events that are blocked are those that only one side is ready to perform but where the co-operation of both is required. Figure 4.6 illustrates the situation, where process $P_1$ with interface $A$ is initially able to perform events in $C$, and $P_2$ with interface $B$ is initially able to perform events in $D$: the events that can initially be performed are those in the shaded regions.

Laws ‖-term 1 and ‖-term 2 are concerned with termination of a parallel combination. If both components are ready to terminate, then termination occurs. If only one component is ready for termination, then only the possibilities of the other side are initially available.

EXAMPLE 4.10 The parallel combination

$$\begin{array}{lcl} (a \to STOP & {}_{\{a,b,c\}}\|_{\{b,c,d,e\}} & (b \to STOP \\ \mid c \to STOP) & & \mid c \to STOP \\ & & \mid d \to STOP) \end{array}$$

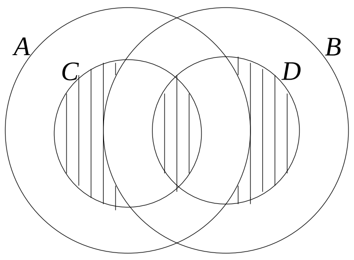


***Fig. 4.6*** Initial offers of a parallel combination

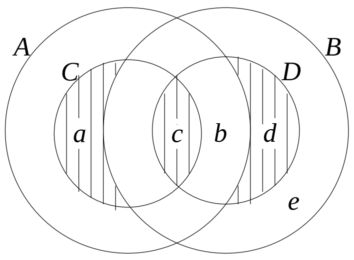


***Fig. 4.7*** Initial offers of the parallel combination of Example 4.10

can be reduced to a single prefix choice using law $\|$-step. The interface sets are $A = \{a, b, c\}$ and $B = \{b, c, d, e\}$, and the initial choice sets are $C = \{a, c\}$ and $D = \{b, c, d\}$. In all cases, the subsequent processes $P_1(x)$ and $P_2(y)$ are *STOP*.

The initial choice is given by $(C \setminus B) \cup (D \setminus A) \cup (B \cap C)$. The set $C \setminus B = \{a\}$ is the set of events that can be performed initially by the left-hand process independently of the right-hand one. The set $D \setminus A = \{d\}$ is the set of events that can be performed initially be the right-hand process independently of the left. Finally, the set $B \cap C = \{c\}$ is the set of events that both processes can initially synchronize on. The combined set of events that are initially on offer is the union of these possibilities: the set $\{a, c, d\}$. Event $b$ is blocked by the left-hand side, and event $e$ is not offered by the right-hand side. This situation is illustrated in Figure 4.7.

Law $\|$-step also describes the processes subsequent to each of these events:

$$\begin{array}{l} a \rightarrow (STOP \;{}_{\{a,b,c\}}\|_{\{b,c,d,e\}} \;\; \begin{array}[t]{l} b \rightarrow STOP \\ \mid c \rightarrow STOP \\ \mid d \rightarrow STOP) \end{array} \\ \mid d \rightarrow (a \rightarrow STOP \mid c \rightarrow STOP \;{}_{\{a,b,c\}}\|_{\{b,c,d,e\}}\; STOP) \\ \mid c \rightarrow (STOP \;{}_{\{a,b,c\}}\|_{\{b,c,d,e\}}\; STOP) \end{array}$$

Law $\|$-step is applicable to each subsequent behaviour. The process following the performance of $a$ is given by

$$\begin{array}{lll} & STOP\ {}_{\{a,b,c\}}\|_{\{b,c,d,e\}}\ \begin{array}{l}(b \rightarrow STOP\quad)\\ \mid c \rightarrow STOP\\ \mid d \rightarrow STOP)\end{array} & \\ = & d \rightarrow (STOP\ {}_{\{a,b,c\}}\|_{\{b,c,d,e\}}\ STOP) & \|\text{-step} \\ = & d \rightarrow STOP & STOP\text{-step} \end{array}$$

The other branches of the initial choice reduce in a similar way, resulting in the following description which is given entirely in terms of prefix and choice:

$$\begin{array}{l} a \rightarrow d \rightarrow STOP \\ \mid d \rightarrow a \rightarrow STOP \\ \mid c \rightarrow STOP \end{array}$$

Components either perform $a$ and $d$ independently, or synchronize on $c$. □

EXAMPLE 4.11 Two processes $P_1 = a \rightarrow b \rightarrow STOP$ and $P_2 = b \rightarrow c \rightarrow STOP$ are required to synchronize on $a$, $b$, and $c$.

$$(a \rightarrow b \rightarrow STOP)\ {}_{\{a,b,c\}}\|_{\{a,b,c\}}\ (b \rightarrow c \rightarrow STOP)$$

The operational semantics for this process has no transitions, so it behaves the same way as $STOP$. The laws for parallel composition allow this conclusion to be reached by reasoning at the level of trace equivalences.

The interface sets $A$ and $B$ are both $\{a, b, c\}$, the initial set for $P_1$ is $C = \{a\}$, and the initial set for $P_2$ is $D = \{b\}$. The events initially on offer are given by the union of $C \setminus B$ (the events that $P_1$ can perform independently of $P_2$), $D \setminus A$ (the events that $P_2$ can perform independently of $P_1$), and $C \cap D$ (the events on which they can initially synchronize). Each of these sets is the empty set $\{\}$, so law $\|$-step states that

$$\begin{array}{lll} P_1\ {}_A\|_B\ P_2 & = & x : \{\} \rightarrow P(x) \\ & = & STOP \qquad\qquad \text{by } STOP\text{-step} \end{array}$$

The parallel combination deadlocks immediately—no events can initially be performed.

□

EXAMPLE 4.12 The parallel combination

$$a \rightarrow b \rightarrow STOP\ {}_{\{a,b\}}\|_{\{b,c\}}\ b \rightarrow c \rightarrow STOP$$

can be rewritten to remove the parallel operator using the laws for parallel. The laws *STOP*-step and prefix are used implicitly to treat event prefixes and *STOP* as prefix choices.

$$\begin{array}{cll} & a \to b \to STOP\ {}_{\{a,b\}}\|_{\{b,c\}}\ b \to c \to STOP & \\ = & a \to ((b \to STOP)\ {}_{\{a,b\}}\|_{\{b,c\}}\ b \to c \to STOP) & \text{by } \|\text{-step} \\ = & a \to b \to (STOP\ {}_{\{a,b\}}\|_{\{b,c\}}\ c \to STOP) & \text{by } \|\text{-step} \\ = & a \to b \to c \to STOP & \text{by } \|\text{-step} \end{array}$$

At the first step the event *b* is blocked by the left-hand process, and only event *a* is possible. This is reflected in the application of the law, which does not make *b* available at the first step because it is not a choice offered by both sides. □

EXAMPLE 4.13 Two processes

$$\begin{array}{lcl} P_1 & = & a \to x \to STOP \mid b \to y \to STOP \\ P_2 & = & c \to x \to STOP \mid d \to y \to STOP \end{array}$$

have respective interfaces

$$\begin{array}{lcl} A & = & \{a,b,x,y\} \\ B & = & \{c,d,x,y\} \end{array}$$

They are intended to operate independently on the events *a*, *b*, *c*, and *d*, but to synchronize on the events *x* and *y*. Their combination is described as

$$P_1\ {}_A\|_B\ P_2$$

$$P \;{}_{\Sigma}\|_{\Sigma}\; RUN =_U P \qquad \langle\|\text{-unit}\rangle$$

$$P \;{}_{A}\|_{\Sigma}\; STOP = STOP \qquad \langle\|\text{-zero}\rangle$$

***Fig. 4.8*** Further laws for parallel

All the first events of each process are independent of the other process, and so all are initially available in the parallel combination. An application of law ||-step gives

$$\begin{array}{cll}
 & P_1 \;{}_A\|_B\; P_2 & \\
= & a \rightarrow ((x \rightarrow STOP) \;{}_A\|_B\; P_2) & \text{by } \|\text{-step} \\
 & \mid b \rightarrow ((y \rightarrow STOP) \;{}_A\|_B\; P_2) & \\
 & \mid c \rightarrow (P_1 \;{}_A\|_B\; x \rightarrow STOP) & \\
 & \mid d \rightarrow (P_1 \;{}_A\|_B\; y \rightarrow STOP) & \\
= & a \rightarrow (\;\; c \rightarrow ((x \rightarrow STOP) \;{}_A\|_B\; x \rightarrow STOP) & \text{by } \|\text{-step} \\
 & \qquad\quad \mid d \rightarrow ((x \rightarrow STOP) \;{}_A\|_B\; y \rightarrow STOP)) & \\
 & \mid b \rightarrow (\;\; c \rightarrow ((y \rightarrow STOP) \;{}_A\|_B\; x \rightarrow STOP) & \\
 & \qquad\quad \mid d \rightarrow ((y \rightarrow STOP) \;{}_A\|_B\; y \rightarrow STOP)) & \\
 & \mid c \rightarrow (\;\; a \rightarrow ((x \rightarrow STOP) \;{}_A\|_B\; x \rightarrow STOP) & \\
 & \qquad\quad \mid b \rightarrow ((y \rightarrow STOP) \;{}_A\|_B\; x \rightarrow STOP)) & \\
 & \mid d \rightarrow (\;\; a \rightarrow ((x \rightarrow STOP) \;{}_A\|_B\; y \rightarrow STOP) & \\
 & \qquad\quad \mid b \rightarrow ((y \rightarrow STOP) \;{}_A\|_B\; y \rightarrow STOP)) & \\
= & a \rightarrow (\;\; c \rightarrow x \rightarrow STOP & \text{by } \|\text{-step} \\
 & \qquad\quad \mid d \rightarrow STOP) & \\
 & \mid b \rightarrow (\;\; c \rightarrow STOP & \\
 & \qquad\quad \mid d \rightarrow y \rightarrow STOP) & \\
 & \mid c \rightarrow (\;\; a \rightarrow x \rightarrow STOP & \\
 & \qquad\quad \mid b \rightarrow STOP) & \\
 & \mid d \rightarrow (\;\; a \rightarrow STOP & \\
 & \qquad\quad \mid b \rightarrow y \rightarrow STOP) &
\end{array}$$

In order for the two components to synchronize on an *x* or *y* event, they must independently follow paths that lead to the same event. □

The parallel operator has process *STOP* as a zero, and *RUN* as a unit, as given in Figure 4.8.

When applying the laws of parallel to expand a parallel composition $P_1 \parallel P_2$, the implicit interface sets $\alpha(P_1)$ and $\alpha(P_2)$ must first be made explicit. The process $P_1 \parallel P_2$ is an abbreviation for $P_1 \;{}_{\alpha(P_1)}\|_{\alpha(P_2)}\; P_2$.

## Interleaving

An interleaving of two processes $P_1 \mid\mid\mid P_2$ executes each component entirely independently of the other, until termination. Traces of the combination will therefore appear as *interleavings* of traces of the two component processes.

A trace $tr$ is an interleaving of two others $tr_1$ and $tr_2$ if each occurrence of each event from $tr_1$ and $tr_2$ appears exactly once in $tr$, and events from $tr_1$ and $tr_2$ occur in the same order. They must also agree on termination. This is denoted $tr$ interleaves $tr_1, tr_2$. For example,

$$\langle a, c, b\rangle \text{ interleaves } \langle a, b\rangle, \langle c\rangle$$

$$\langle a, d\rangle \text{ interleaves } \langle\rangle, \langle a, d\rangle$$

This may be formally defined by a structural induction on sequences:

$$\begin{array}{rcl}
\langle\rangle \text{ interleaves } tr_1, tr_2 & \Leftrightarrow & tr_1 = tr_2 = \langle\rangle \\
\langle\checkmark\rangle \text{ interleaves } tr_1, tr_2 & \Leftrightarrow & tr_1 = tr_2 = \langle\checkmark\rangle
\end{array}$$

$$\begin{array}{lcl}
\langle a\rangle \frown tr \neq \langle\checkmark\rangle \Rightarrow & & \\
\langle a\rangle \frown tr \text{ interleaves } tr_1, tr_2 & \Leftrightarrow & head(tr_1) = a \wedge tr \text{ interleaves } tail(tr_1), tr_2 \\
 & & \vee\ head(tr_2) = a \wedge tr \text{ interleaves } tr_1, tail(tr_2)
\end{array}$$

If a trace beginning with $a$ interleaves two others, then one of those two must begin with $a$, and the subsequent trace must be an interleaving of the subsequent two traces.

Any trace of the interleaved process $P_1 \mid\mid\mid P_2$ will be an interleaving of a trace from $P_1$ and a trace from $P_2$. The traces of $P_1 \mid\mid\mid P_2$ are given as follows:

$$\begin{array}{rcl}
\text{traces}\,(P_1 \mid\mid\mid P_2) & = & \{tr \in TRACE \mid \exists\, tr_1, tr_2 \bullet\ tr_1 \in \text{traces}(P_1) \wedge \\
 & & \qquad tr_2 \in \text{traces}(P_2) \wedge \\
 & & \qquad tr \text{ interleaves } tr_1, tr_2\}
\end{array}$$

EXAMPLE 4.14 The traces of $(a \to b \to STOP) \mid\mid\mid (c \to STOP)$ are calculated from the trace sets of the two component processes:

$$\begin{array}{rcl}
\text{traces}(a \to b \to STOP) & = & \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\} \\
\text{traces}(c \to STOP) & = & \{\langle\rangle, \langle c\rangle\}
\end{array}$$

The traces of the combined process is made up of all possible interleavings of pairs of traces:

$$\begin{array}{l}
\text{traces}((a \to b \to STOP) \mid\mid\mid (c \to STOP)) \\
\quad = \ \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle c\rangle, \langle a, c\rangle, \langle c, a\rangle, \langle a, b, c\rangle, \langle a, c, b\rangle, \langle c, a, b\rangle\}
\end{array}$$

$P_1 \mid\mid\mid P_2 = P_2 \mid\mid\mid P_1$ ⟨|||-sym⟩

$P_1 \mid\mid\mid (P_2 \mid\mid\mid P_3) = (P_1 \mid\mid\mid P_2) \mid\mid\mid P_3$ ⟨|||-assoc⟩

$P \mid\mid\mid SKIP = P$ ⟨|||-unit⟩

$P \mid\mid\mid RUN_\Sigma = RUN_\Sigma$ ⟨|||-zero⟩

$(x : C \to P_1(x)) \mid\mid\mid (y : D \to P_2(y)) = z : (C \cup D) \to R(z)$ ⟨|||-step⟩

where

$$\begin{array}{rcll} R(c) & = & P_1(c) \mid\mid\mid (y : D \to P_2(y)) & \text{if } c \in C \setminus D \\ & = & (x : C \to P_1(x)) \mid\mid\mid P_2(c) & \text{if } c \in D \setminus C \\ & = & P_1(c) \mid\mid\mid (y : D \to P_2(y)) & \text{if } c \in C \cap D \\ & & \sqcap\ (x : C \to P_1(x)) \mid\mid\mid P_2(c) & \end{array}$$

$SKIP \mid\mid\mid SKIP = SKIP$ ⟨|||-term 1⟩

$(x : C \to P(x)) \mid\mid\mid SKIP = (x : C \to (P(x) \mid\mid\mid SKIP))$ ⟨|||-term 2⟩

***Fig. 4.9*** Laws for interleaving

The *a* must occur before the *b*, but the *c* can occur anywhere with respect to these two events. □

There are a number of trace laws concerning interleaving. These are listed in Figure 4.9. The first two laws state simply that the interleaving operator is commutative and associative. The next two laws give a unit and a zero for the operator. The fifth law gives a way of expanding an interleaving of two choices into a single prefix choice. It states that such an interleaving offers the choice of any of the first events of either of its components.

Interleaving parallel allows its two component processes independent control over termination. The entire combination will terminate when either of its component processes does so. This is reflected in Laws |||-term 1 and |||-term 2. If both sides are ready to terminate, then only termination can occur. Alternatively, if one side is ready to terminate but the other side is able to progress, then progress occurs in accordance with the non-terminating component.

EXAMPLE 4.15 The process $(a \to b \to STOP) \mid\mid\mid (c \to STOP)$ may be rewritten using Law |||-step as follows:

$$\begin{array}{ll} (a \to b \to STOP) \mid\mid\mid (c \to STOP) & \\ \quad =_T & a \to ((b \to STOP) \mid\mid\mid (c \to STOP)) \\ & \mid c \to ((a \to b \to STOP) \mid\mid\mid STOP) \end{array}$$

The left-hand component is initially able to perform $a$, and the right-hand component is initially able to perform $c$. The combination therefore offers a choice between $a$ and $c$. Further applications of $|||$-step and $|||$-unit reduce the process to

$$\begin{array}{l} a \rightarrow (\ \ b \rightarrow c \rightarrow STOP \\ \qquad\quad \mid c \rightarrow b \rightarrow STOP) \\ \mid c \rightarrow a \rightarrow b \rightarrow STOP \end{array}$$

□

## Interface parallel

The process $P_1 \underset{A}{\|} P_2$ is a blend of both the parallel operator and the interleaving operator. Its traces will consist of combinations of traces of $P_1$ and $P_2$ which match on all occurrences of events in $A^{\checkmark}$, and which interleave on events not in $A^{\checkmark}$.

Traces $tr_1$ of $P_1$ and $tr_2$ of $P_2$ may combine in a number of ways in the combination $P_1 \underset{A}{\|} P_2$, provided they agree on events from $A$. The relation $tr\ \mathsf{synch}_A\ tr_1, tr_2$ states that $tr$ describes one way in which $tr_1$ and $tr_2$ can combine. It is defined as follows:

$$\begin{array}{rcl} \langle\rangle\ \mathsf{synch}_A\ tr_1, tr_2 & \Leftrightarrow & tr_1 = tr_2 = \langle\rangle \\ \langle\checkmark\rangle\ \mathsf{synch}_A\ tr_1, tr_2 & \Leftrightarrow & tr_1 = tr_2 = \langle\checkmark\rangle \end{array}$$

$$\begin{array}{l} \langle a\rangle ^\frown tr \neq \langle\checkmark\rangle \Rightarrow \\ \quad \langle a\rangle ^\frown tr\ \mathsf{synch}_A\ tr_1, tr_2 \quad \Leftrightarrow \quad (a \in A\ \wedge head(tr_1) = head(tr_2) = a \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ tr\ \mathsf{synch}_A\ tail(tr_1), tail(tr_2)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \vee\ \ a \notin A \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad (head(tr_1) = a \wedge tr\ \mathsf{synch}_A\ tail(tr_1), tr_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee\ head(tr_2) = a \wedge tr\ \mathsf{synch}_A\ tr_1, tail(tr_2)) \end{array}$$

The constraint that the traces must agree on $A$ means that some traces $tr_1$ and $tr_2$ are not consistent. In this case, there will be no $tr$ which relates to the pair of them. For example, $\langle a, b, a\rangle$ and $\langle a, c\rangle$ cannot agree on the set $\{a\}$.

Any trace of the parallel process $P_1 \underset{A}{\|} P_2$ will be a combination of a trace from $P_1$ and a trace from $P_2$. The traces of $P_1 \underset{A}{\|} P_2$ are given as follows:

$$\begin{array}{rcll} \mathsf{traces}(P_1 \underset{A}{\|} P_2) & = & \{tr \in TRACE \mid \exists\, tr_1, tr_2 \bullet & tr_1 \in \mathsf{traces}(P_1) \wedge \\ & & & tr_2 \in \mathsf{traces}(P_2) \wedge \\ & & & tr\ \mathsf{synch}_A\ tr_1, tr_2 \quad \} \end{array}$$

EXAMPLE 4.16

$$\begin{array}{l} \mathsf{traces}\,((a \rightarrow b \rightarrow a \rightarrow STOP) \underset{\{a\}}{\|} (a \rightarrow c \rightarrow a \rightarrow STOP)) \\ \quad = \ \ \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, c\rangle, \langle a, b, c\rangle, \langle a, c, b\rangle, \langle a, b, c, a\rangle, \langle a, c, b, a\rangle\} \end{array}$$

$$P_1 \underset{A}{\|} P_2 = P_2 \underset{A}{\|} P_1 \qquad \langle \underset{A}{\|}\text{-sym}\rangle$$

$$P_1 \underset{A}{\|} (P_2 \underset{A}{\|} P_3) = (P_1 \underset{A}{\|} P_2) \underset{A}{\|} P_3 \qquad \langle \underset{A}{\|}\text{-assoc}\rangle$$

$$P \underset{A}{\|} RUN_{A^{\checkmark}} =_U P \qquad \langle \underset{A}{\|}\text{-unit}\rangle$$

$$P \underset{A}{\|} RUN_{\Sigma \setminus A} =_U RUN_{\Sigma \setminus A} \qquad \langle \underset{A}{\|}\text{-zero}\rangle$$

$$(x : C \to P_1(x)) \underset{A}{\|} (y : D \to P_2(y)) \langle \underset{A}{\|}\text{-step}\rangle$$

$$= z : (((C \cup D) \setminus A) \cup (C \cap D \cap A)) \to R(z)$$

where

$$\begin{aligned} R(c) &= P_1(c) \underset{A}{\|} (y : D \to P_2(y)) && \text{if } c \in C \setminus (A \cup D) \\ &= (x : C \to P_1(x)) \underset{A}{\|} P_2(c) && \text{if } c \in D \setminus (A \cup C) \\ &= P_1(c) \underset{A}{\|} (y : D \to P_2(y)) && \text{if } c \in (C \cap D) \setminus A \\ &\quad \sqcap (x : C \to P_1(x)) \underset{A}{\|} P_2(c) && \\ &= P_1(c) \underset{A}{\|} P_2(c)) && \text{if } c \in C \cap D \cap A \end{aligned}$$

$$SKIP \underset{A}{\|} SKIP = SKIP \qquad \langle \underset{A}{\|}\text{-term 1}\rangle$$

$$(x : C \to P(x)) \underset{A}{\|} SKIP \qquad \langle \underset{A}{\|}\text{-term 2}\rangle$$

$$= x : (C \setminus A) \to (P(x) \underset{A}{\|} SKIP)$$

***Fig. 4.10*** Laws for interface parallel

The traces of the component processes must agree on occurrences of $a$, but are otherwise independent. □

There are a number of trace laws concerning interface parallel. These are listed in Figure 4.10.

The first two laws are concerned with commutativity and associativity of the interface parallel operator. Associativity applies in the case that both instances $A$ of the interface set are the same. Law $\underset{A}{\|}$-unit gives a unit for the operator: $RUN_A$ allows $P$ to perform any event in $A$, and the common interface $A$ means that $P$ can independently perform events not in $A$; so the process $P \underset{A}{\|} RUN_{A^{\checkmark}}$ has exactly the same traces as $P$. A zero which blocks all events in $A$ and masks all other events, is given by Law $\underset{A}{\|}$-zero. Law $\underset{A}{\|}$-step allows a parallel combination of choices to be expanded to a prefix choice of processes. The events offered by the prefix choice

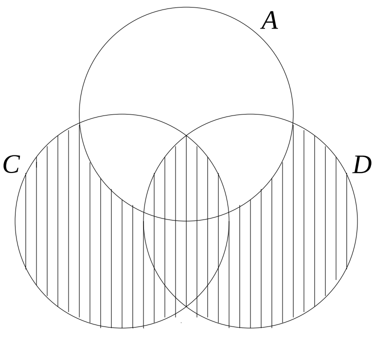


***Fig. 4.11*** Initial possibilities for a interface parallel combination

are those in $A$ which are offered by both components, together with those not in $A$ offered by either component. These possibilities are illustrated in Figure 4.11.

The behaviour of interface parallel with respect to termination is given by the last two laws. Law $\underset{A}{\|}$-term 1 states that if both components are ready to terminate then termination must occur. Law $\underset{A}{\|}$-term 2 is concerned with the case where one side is ready to terminate but the other is not; termination is not a possibility. The other process may progress on any event that it is able to perform independently—any event not in the common interface $A$.

The relationship between interface parallel and the other two forms of parallel is made explicit in the following two laws:

$$P_1 \;{}_A\|_B\; P_2 = P_1 \underset{(A\cap B)}{\|} P_2 \quad \text{if} \quad \begin{array}{l}\alpha(P_1) \subseteq A \\ \wedge\ \alpha(P_2) \subseteq B\end{array} \qquad \langle \underset{A}{\|}\text{-equiv 1}\rangle$$

$$P_1 \;|||\; P_2 = P_1 \underset{\{\}}{\|} P_2 \qquad \langle \underset{A}{\|}\text{-equiv 2}\rangle$$

Law $\underset{A}{\|}$-equiv 1 covers the case where $P_1$ and $P_2$ must synchronize on all events that are in $A \cap B$, and can perform independently only those events which are in one alphabet and outside $A \cap B$. This is naturally written using the alphabetized parallel operator, but the effect is that $A \cap B$ is a common interface, and it can equally be written with the interface parallel operator.

Law $\underset{A}{\|}$-equiv 2 states simply that process interleaving is equivalent to an empty interface parallel combination.

| | |
|---|---|
| $(P \setminus A) \setminus B = P \setminus (A \cup B)$ | ⟨hide-combine⟩ |
| $(a \rightarrow P) \setminus A = \begin{cases} a \rightarrow (P \setminus A) & \text{if } a \notin A \\ P \setminus A & \text{if } a \in A \end{cases}$ | ⟨hide-step 1⟩ |
| $(\sqcap_{i \in I} P_i) \setminus A = \sqcap_{i \in I}(P_i \setminus A)$ | ⟨$\sqcap$-dist⟩ |
| $STOP \setminus A = STOP$ | ⟨hide-*STOP*⟩ |
| $(x : C \rightarrow P(x)) \setminus A = x : C \rightarrow (P(x) \setminus A) \quad \text{if } A \cap C = \{\}$ | ⟨hide-step 2⟩ |
| $(x : C \rightarrow P(x)) \setminus A = \sqcap_{x \in C}(P(x) \setminus A) \quad \text{if } C \subseteq A$ | ⟨hide-step 3⟩ |
| $SKIP \setminus A = SKIP$ | ⟨hide-term⟩ |

***Fig. 4.12*** Laws for hiding

## Hiding

The process $P \setminus A$ for $A \subseteq \Sigma$ has the same executions as $P$, except that at any point where $P$ performs a visible event from $A$, the process $P \setminus A$ performs the same event internally. All events from $A$ become internal events in $P \setminus A$, and do not appear in its traces. Any trace $tr$ of $P$ gives rise to a trace $tr \setminus A$ of $P \setminus A$; and conversely, any trace of $P \setminus A$ must be derived from a trace of $P$ with the events from $A$ made internal.

$$\text{traces}(P \setminus A) \quad = \quad \{tr \setminus A \mid tr \in \text{traces}(P)\}$$

For instance

$$\text{traces}(a \rightarrow b \rightarrow a \rightarrow STOP) \quad = \quad \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, a\rangle\}$$

and so

$$\text{traces}((a \rightarrow b \rightarrow a \rightarrow STOP) \setminus \{a\}) \quad = \quad \{\langle\rangle, \langle b\rangle\}$$

There are a number of laws concerning hiding. These are given in Figure 4.12. The first law states that hiding successive sets of events obtains the same process as hiding all the sets of events at once. It follows from this law that hiding is commutative: that $P \setminus A \setminus B = P \setminus B \setminus A$.

The second law is concerned with the effect of an abstraction on the occurrence of an event. If the event $a$ does not appear in the abstracted set of events $A$, then it is not hidden and it appears as a prefix to the subsequent process $P \setminus A$. If $a$ does occur in $A$ then it is internal and so the subsequent process $P \setminus A$ is immediately reached.

The third law states that hiding distributes over indexed internal choice: abstracting events from a choice of process will yield the same traces as a single choice from a set of processes which all have their events abstracted. The fourth law is the special case in which no events are offered.

The fifth and sixth laws are special instances of hiding over a prefix choice. In the first case none of the choice events is hidden, resulting in the same choice of events being offered. In the second case all of the choice events are hidden, resulting in the choice between the subsequent processes. These two laws are often applicable when channels are hidden: if the channel $c$ is hidden, then all events in the initial choice of the input process $c?x : T \to P(x)$ become internal in accordance with Law hide-step 3; if $c$ is not hidden, then none of them become internal and the entire input choice remains, in accordance with Law hide-step 2. Finally, the last law states that hiding does not affect termination.

EXAMPLE 4.17 In the case where some events of a prefix choice are hidden, but not all of them, the laws hide-step 1 and $\sqcap$-dist are used to separate out the individual branches of the choice, and then to apply the hiding operator to each one separately.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\left(\begin{array}{l} a \to c \to STOP \\ \Box\ b \to d \to STOP \end{array}\right) \setminus \{b\}} \\
=_T & \text{by } \bigsqcap\text{-dist and } \sqcap\text{-}\Box\text{-equiv}_T \\
 & (a \to c \to STOP) \setminus \{b\} \ \Box\ (b \to d \to STOP) \setminus \{b\} \\
=_T & \text{by hide-step 1} \\
 & a \to (c \to STOP) \setminus \{b\} \\
 & \Box\ (d \to STOP) \setminus \{b\} \\
=_T & \text{by hide-step 1} \\
 & a \to c \to (STOP \setminus \{b\}) \\
 & \Box\ d \to (STOP \setminus \{b\}) \\
=_T & \text{by hide-step 1} \\
 & a \to c \to STOP \\
 & \Box\ d \to STOP
\end{array}
$$

The $b$ event is no longer visible in the resulting process description. $\Box$

## Renaming

The forward renaming operator $f(P)$ behaves the same way as $P$ but performs $f(a)$ whenever $P$ would have performed $a$. Its traces are the traces of $P$ with every event mapped through $f$.

The set of traces of $f(P)$ can be defined:

$$\text{traces}(f(P)) \quad = \quad \{f(tr) \mid tr \in \text{traces}(P)\}$$

| | |
|---|---|
| $f(x : C \to P(x)) = y : f(C) \to f(P(f^{-1}(y)))$ if $f$ is $1 - 1$ | $\langle f(.)\text{-step 1}\rangle$ |
| $f(x : C \to P(x)) = y : f(C) \to \bigsqcap_{x \mid f(x)=y} f(P(x))$ | $\langle f(.)\text{-step 2}\rangle$ |
| $f(SKIP) = SKIP$ | $\langle f(.)\text{-term}\rangle$ |
| $l : (x : C \to P(x)) = y : (l.C) \to P(f_l^{-1}(y))$ | $\langle l :\text{-step}\rangle$ |
| where $l.C = \{l.c \mid c \in C\}$ | |
| $l : SKIP = SKIP$ | $\langle l :\text{-term}\rangle$ |
| $f^{-1}(x : C \to P(x)) = y : f^{-1}(C) \to f^{-1}(P(f(y)))$ | $\langle f^{-1}(.)\text{-step}\rangle$ |
| $f^{-1}(SKIP) = SKIP$ | $\langle f^{-1}(.)\text{-term}\rangle$ |

***Fig. 4.13*** Laws for renaming

This is indeed a set of traces, since the restrictions on alphabet renaming mean that $f$ maps $\Sigma$ into $\Sigma$, and $f(\checkmark) = \checkmark$.

For instance,

$$\text{traces}(a \to b \to a \to STOP) \quad = \quad \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, a\rangle\}$$

and so if $f(a) = c$ and $f(b) = d$ then

$$\text{traces}(f(a \to b \to a \to STOP)) \quad = \quad \{\langle\rangle, \langle c\rangle, \langle c, d\rangle, \langle c, d, c\rangle\}$$

If $g(a) = g(b) = c$ then

$$\text{traces}(g(a \to b \to a \to STOP)) \quad = \quad \{\langle\rangle, \langle c\rangle, \langle c, c\rangle, \langle c, c, c\rangle\}$$

If the mapping $f$ is one-one, then renaming with $f$ has a straightforward interaction with prefix choice, as given by Law $f(.)$-step 1 in Figure 4.13. A choice of events from $C$ becomes a choice of events from $f(C) = \{f(c) \mid c \in C\}$. The fact that $f$ is injective means that the event $y$ chosen corresponds to exactly one event $x(= f^{-1}(y))$ from the original choice of events from $C$, so the subsequent behaviour is that of $P(x)$ transformed through $f$.

In general, the renaming operator interacts with prefix choice as given by Law $f(.)$-step 2 in Figure 4.13. If a process $P$ initially is prepared to perform any event from $C$, then the initial choice for $f(P)$ is the set of events $f(C)$. However, the result of choosing $y$ could be any of the processes which follow an event mapping to $y$: if $a$ and $b$ both appear in $C$, and $f$ maps them both to the same event $c$, then $f(P)$ is in effect offering $c$ in two different ways, once resulting from $a$ and once resulting from $b$. The process subsequent to $c$ can be either $f(P(a))$ or $f(P(b))$.

All of the term laws state that the various sorts of renaming cannot affect a process' ability to terminate.

EXAMPLE 4.18 The process $P$ initially offers a choice from the set of three events $\{a, b, c\}$:

$$\begin{array}{rcl} P & = & a \rightarrow d \rightarrow STOP \\ & & \mid b \rightarrow e \rightarrow STOP \\ & & \mid c \rightarrow STOP \end{array}$$

Let the mapping $f$ be defined by

$$\begin{array}{rcl} f(a) & = & k \\ f(b) & = & k \\ f(c) & = & l \\ f(d) & = & m \\ f(e) & = & n \end{array}$$

so $f$ maps both $a$ and $b$ to the same event $k$. Then the process $f(P)$ reduces under Law $f(.)$-step 2 to

$$\begin{array}{l} k \rightarrow (m \rightarrow STOP \sqcap n \rightarrow STOP) \\ \Box\ l \rightarrow STOP \end{array}$$

The process $f(P)$ only offers a choice between two events, where $P$ offered a choice between three. The process following the choice of $k$ can be one of two possibilities, derived from the behaviour of $P$ following $a$, or following $b$. □

Process relabelling $l : P$ is a special form of forward renaming in which all events $a$ are associated with the label $l$, by means of the renaming function $f_l$ which maps $a$ to $l.a$ for any event $a \neq \checkmark$, and $f_l(\checkmark) = \checkmark$. The set of traces is given by the trace definition of forward renaming:

$$\text{traces}(l : P) \quad = \quad \{f_l(tr) \mid tr \in \text{traces}(P)\}$$

The mapping function $f_l$ is one-one, so the relabelling law is a specialization of Law $f(.)$-step 1.

The backward renaming operator $f^{-1}(P)$ also behaves in a similar fashion to $P$, but any event $a$ that is performed by $f^{-1}(P)$ corresponds to an event $f(a)$ performed by $P$. Hence a trace $tr$ of $f^{-1}(P)$, when mapped through the function $f$, must yield a trace $f(tr)$ of $P$.

$$\text{traces}(f^{-1}(P)) \quad = \quad \{tr \mid f(tr) \in \text{traces}(P)\}$$

The interaction between backward renaming and prefix choice is straightforward, and the laws are given in Figure 4.13.

A set of events $C$ offered as a choice by $P$ becomes a choice over $f^{-1}(C)$ offered by $f^{-1}(P)$, since it is precisely the events in $f^{-1}(C)$ that map to an event that $P$ can initially perform—an event in $C$. If a particular event $a$ is chosen, then the subsequent behaviour is determined by the behaviour of $P$ following $f(a)$.

EXAMPLE 4.19 The process $P'$ initially offers a choice from the set of two events $\{k, l\}$.

$$\begin{array}{lll} P' & = & k \to (m \to STOP \mid n \to STOP) \\ & & \mid l \to STOP \end{array}$$

If the mapping $f$ is defined as in Example 4.18, then the process $f^{-1}(P')$ reduces under Law $f^{-1}$-step to

$$\begin{array}{l} a \to (d \to STOP \mid e \to STOP) \\ \Box\ b \to (d \to STOP \mid e \to STOP) \\ \Box\ c \to STOP \end{array}$$

The resulting choice is between three events. In the case where $a$ or $b$ is chosen, the subsequent behaviour is that of $f^{-1}(P(f(a))) = f^{-1}(P(f(b))) = f^{-1}(P(k))$

The process $P'$ is equivalent to $f(P)$ where $P$ was defined in Example 4.18. However, $f^{-1}(P')$ is not equivalent to $P$, since there are some possibilities for $f^{-1}(P')$ which are not possible for $P$; one example is the performance of event $e$ following event $a$. This is manifested in the sets of traces in the fact that $\langle a, e\rangle \in \text{traces}(f^{-1}(P'))$ but $\langle a, e\rangle \notin \text{traces}(P)$. Hence $P$ and $f^{-1}(P') = f^{-1}(f(P))$ are not trace equivalent. □

## Sequential Composition

The sequential composition $P_1;\ P_2$ behaves as $P_1$ until $P_1$ terminates successfully, at which point it passes control to $P_2$. Since termination of $P_1$ does not denote termination of the entire construct, $P_1$'s ✓ event is made internal.

The traces of $P_1;\ P_2$ fall into two categories: traces of $P_1$ before termination, and terminating traces of $P_1$ followed by traces of $P_2$.

$$\begin{array}{lll} \text{traces}(P_1;\ P_2) & = & \{tr \mid tr \in \text{traces}(P_1) \wedge \checkmark \notin \sigma(tr)\} \\ & & \cup \{tr_1 \frown tr_2 \mid tr_1 \frown \langle \checkmark \rangle \in \text{traces}(P_1) \wedge tr_2 \in \text{traces}(P_2)\} \end{array}$$

There are a number of laws appropriate to sequential composition. These are given in Figure 4.14.

Law ; -assoc simply states that sequential composition is associative. The unit laws state that $SKIP$ is a left and right unit of sequential composition: that $SKIP$ is absorbed by

| | |
|---|---|
| $(P_1;\ P_2);\ P_3 = P_1;\ (P_2;\ P_3)$ | $\langle$; -assoc$\rangle$ |
| $SKIP;\ P = P$ | $\langle$; -unit-l$\rangle$ |
| $P;\ SKIP =_T P$ | $\langle$; -unit-r$\rangle$ |
| $(x : C \rightarrow P(x));\ P_1 = x : C \rightarrow (P(x);\ P_1)$ | $\langle$; -step$\rangle$ |
| $STOP;\ P = STOP$ | $\langle$; -zero-l$\rangle$ |

***Fig. 4.14*** Laws for sequential composition

sequential composition (though the right unit law holds only in the traces model). Law ; -step states that a prefix choice in a sequential composition is equivalent to a prefix choice of sequentially composed processes. Law ; -zero-l is a special case of Law ; -step, in which no events are initially offered—this yields a left zero for sequential composition.

EXAMPLE 4.20 The process $P_1$ is defined as follows:

$$\begin{array}{rcl} P_1 & = & a \rightarrow SKIP \\ & & \mid b \rightarrow STOP \end{array}$$

If $P_1$ is sequentially composed with process $P_2 = c \rightarrow SKIP$ then the result is

$$\begin{array}{rcl} P_1;\ P_2 & = & \left( \begin{array}{l} a \rightarrow SKIP \\ \Box\ b \rightarrow STOP \end{array} \right);\ c \rightarrow SKIP \\ & = & a \rightarrow (SKIP;\ c \rightarrow SKIP) \\ & & \mid b \rightarrow (STOP;\ c \rightarrow SKIP) \\ & = & a \rightarrow c \rightarrow SKIP \\ & & \mid b \rightarrow STOP \end{array}$$

The $c$ event can follow the $a$ but not the $b$. □

## Interrupt

The process $P_1 \triangle P_2$ executes as $P_1$, but at any stage before termination it can begin executing as $P_2$. There are therefore two possibilities for any given trace: it is either a trace of $P_1$, or

$$(P_1 \bigtriangleup P_2) \bigtriangleup P_3 = P_1 \bigtriangleup (P_2 \bigtriangleup P_3) \qquad \langle \bigtriangleup\text{-assoc} \rangle$$

$$(x : C \to P_1(x)) \bigtriangleup P_2 = P_2 \,\Box\, (x : C \to (P_1(x) \bigtriangleup P_2)) \qquad \langle \bigtriangleup\text{-step} \rangle$$

$$STOP \bigtriangleup P = P \qquad \langle \bigtriangleup\text{-unit-l} \rangle$$

$$P \bigtriangleup STOP = P \qquad \langle \bigtriangleup\text{-unit-r} \rangle$$

$$SKIP \bigtriangleup P = SKIP \,\Box\, P \qquad \langle \bigtriangleup\text{-term} \rangle$$

***Fig. 4.15*** Laws of interrupt

else it is a non-terminating trace of $P_1$ followed by a trace of $P_2$.

$$\begin{array}{rcl} \text{traces}(P_1 \bigtriangleup P_2) & = & \text{traces}(P_1) \\ & & \cup \\ & & \{tr_1 \frown tr_2 \mid tr_1 \in \text{traces}(P) \wedge \checkmark \notin \sigma(tr_1) \\ & & \qquad \wedge\, tr_2 \in \text{traces}(P_2)\} \end{array}$$

Interrupt satisfies a number of laws, given in Figure 4.15, concerning its interaction with choice and with termination. Law $\bigtriangleup$-assoc states that the interrupt operator is associative: the bracketing of different levels of interrupt is irrelevant. Law $\bigtriangleup$-step shows how a prefix choice interrupted by $P_2$ unwinds: either it behaves as $P_2$ immediately, or else one of the events of the prefix choice occurs, resulting in the subsequent process which may still be interrupted. Law $\bigtriangleup$-unit-l is a special case of $\bigtriangleup$-step in which a process that does nothing may be interrupted by $P$: in this case, the only possible activity is generated by $P$. Law $\bigtriangleup$-unit-r states that the process *STOP* is ineffective as an interrupting process, since there are no events it can perform to interrupt another process. Finally, Law $\bigtriangleup$-term states that if termination occurs, then the interrupting process is discarded.

## Distributive laws

In addition to all of the laws given above for the various CSP operators, there is also a law for each of them concerning distributivity over internal choice. All of the CSP operators (except recursion) distribute over both binary and indexed internal choice, in the traces model, and in fact in all of the models for CSP. For example, the laws for prefix will be as follows:

$$a \to (P_1 \sqcap P_2) = (a \to P_1) \sqcap (a \to P_2) \qquad \langle\text{prefix-dist}\rangle$$

$$a \to \textstyle\bigsqcap_{i \in J} P_i = \bigsqcap_{i \in J}(a \to P_i) \qquad \langle\text{prefix-Dist}\rangle$$

These laws effectively state that no observer can distinguish the case where the internal choice is made after performance of the $a$ from the case where it is made beforehand. In fact the second law subsumes the first.

Binary operators will also distribute over internal choice. For example,

$$P_1\ {}_A\|_B\ (P_2 \sqcap P_3) = (P_1\ {}_A\|_B\ P_2) \sqcap (P_1 \parallel P_3) \qquad \langle\|\text{-dist}\rangle$$

$$P_1\ {}_A\|_B\ \textstyle\bigsqcap_{i \in J} P_i = \bigsqcap_{i \in J}(P_1\ {}_A\|_B\ P_i) \qquad \langle\|\text{-Dist}\rangle$$

Since the parallel operator is symmetric, as indicated by Law $\|$-sym, it follows from these laws that it will also distribute over internal choice in its left-hand argument.

For binary operators that are not symmetric, both a left-hand and a right-hand version of distributivity are given. One example is sequential composition:

$$(P_1 \sqcap P_2);\ P_3 = (P_1;\ P_3) \sqcap (P_2;\ P_3) \qquad \langle;\ \text{-dist-l}\rangle$$

$$P_1;\ (P_2 \sqcap P_3) = (P_1;\ P_2) \sqcap (P_1;\ P_3) \qquad \langle;\ \text{-dist-r}\rangle$$

$$(\textstyle\bigsqcap_{i \in J} P_i);\ P = \bigsqcap_{i \in J}(P_i;\ P) \qquad \langle;\ \text{-Dist-l}\rangle$$

$$P;\ (\textstyle\bigsqcap_{i \in J} P_i) = \bigsqcap_{i \in J}(P;\ P_i) \qquad \langle;\ \text{-dist-r}\rangle$$

All CSP operators (except recursion) are distributive in all arguments over internal choice, and so there will be corresponding distributivity laws for each CSP operator. Law choice-equiv$_T$ of Figure 4.4 stating that $P_1 \sqcap P_2 =_T P_1 \mathrel{\Box} P_2$ means that in the traces model, though not more generally, all operators also distribute over external choice.

## 4.2 RECURSION

The case of recursion has been left until last, since it requires a different treatment to all of the other operators. Traces of processes constructed using the other operators can be deduced from the traces of their components, but in the case of a recursively defined process $N = P$ or $N = F(N)$ which should define the traces of $N$, the traces of the component $P$ or $F(N)$ depend on the traces of $N$ itself, resulting in a circularity. For instance, if $P = F(N) = a \rightarrow N$, then the traces of $P$ are going to be

$$\{\langle\rangle\} \cup \{\langle a\rangle \frown tr \mid tr \in \mathsf{traces}(N)\}$$

which depends on the set $\mathsf{traces}(N)$.

The traces of $N$ can be derived directly from the operational semantics by use of the characterization

$$\mathsf{traces}(N) \quad = \quad \{tr \mid N \overset{tr}{\Longrightarrow}\}$$

but the intention of providing trace semantics is to remove the need to consider processes at the operational level and to support reasoning purely at the level of traces.

Recursion involves defining a process in terms of itself, $N = F(N)$, so it is not surprising that a circularity arises concerning the traces of $N$ simultaneously determining and being determined by the traces of $F(N)$. Even before the traces of $N$ can be determined, there is one fact that must hold:

$$\mathsf{traces}(N) \quad = \quad \mathsf{traces}(F(N))$$

The recursive definition defines an *equation* which must be satisfied by the set $\mathsf{traces}(N)$. In fact, $\mathsf{traces}(N)$ is a *fixed point* of the function on trace sets represented by the CSP expression $F$; when that function is applied to $\mathsf{traces}(N)$ to obtain $\mathsf{traces}(F(N))$, then the result is again $\mathsf{traces}(N)$. This fact is extremely valuable: there are well-established techniques for finding fixed points of functions, and for reasoning about them. They will allow the traces of recursively defined processes to be identified by reasoning purely in terms of traces.

Traces are records of finite executions, so every trace of a recursive process $N = P$ may be obtained by unwinding the recursive definition a finite number of times. Every process contains the empty trace as one of its possible traces, so it follows that $\langle\rangle \in \mathsf{traces}(N)$, or, equivalently,

$$\mathsf{traces}(STOP) \quad \subseteq \quad \mathsf{traces}(N)$$

Applying the function $F$ to each side of the subset relationship yields that

$$\mathsf{traces}(F(STOP)) \quad \subseteq \quad \mathsf{traces}(F(N)) = \mathsf{traces}(N)$$

which states simply that the traces obtained by unwinding the recursive function $F$ once are all traces of $N$. This is justified because all of the CSP operators are *monotonic* with respect to $\subseteq$: in other words, if $\mathsf{traces}(P_1) \subseteq \mathsf{traces}(P_2)$, then $\mathsf{traces}(F(P_1)) \subseteq \mathsf{traces}(F(P_2))$ for any function $F$ constructed out of CSP operators and terms.

It is a standard induction to show that for any $n$

$$\text{traces}(F^n(STOP)) \quad \subseteq \quad \text{traces}(F(N)) = \text{traces}(N)$$

which corresponds to the fact that all of the traces obtained by unwinding the definition $N = F(N)$ $n$ times are still traces of the recursive process $N$.

All of the $F^n(STOP)$ processes correspond to the finite unwindings of the recursive definition, so between them they cover all of the possible traces of $N = P$. Hence

$$\text{traces}(N = F(N)) \quad = \quad \bigcup_{n\in\mathbb{N}} \text{traces}(F^n(STOP))$$

This concurs with the expectation that the resulting set of traces must be a fixed point of the function corresponding to $F$.

EXAMPLE 4.21 Consider the recursively defined process $LIGHT = on \rightarrow off \rightarrow LIGHT$ of Example 1.14. The recursive function is $F(Y) = on \rightarrow off \rightarrow Y$.

$$\begin{aligned}
\text{traces}(STOP) &= \{\langle\rangle\} \\
\text{traces}(F(STOP)) &= \{\langle\rangle, \langle on\rangle, \langle on, off\rangle\} \\
\text{traces}(F(F(STOP))) &= \{\langle\rangle, \langle on\rangle, \langle on, off\rangle, \langle on, off, on\rangle, \langle on, off, on, off\rangle\}
\end{aligned}$$

It appears that

$$\begin{aligned}
\text{traces}(F^n(STOP)) \quad = \quad & \{\langle on, off\rangle^i \mid 0 \leqslant i \leqslant n\} \\
& \cup \\
& \{\langle on, off\rangle^i \frown \langle on\rangle \mid 0 \leqslant i < n\}
\end{aligned}$$

and this conjecture may be established by induction. The base case $(n = 0)$ is immediate, so there is only the inductive step to consider. Assuming the result for $n$:

$$\begin{aligned}
& \text{traces}(F^{n+1}(STOP)) \\
= \quad & \text{traces}(on \rightarrow off \rightarrow F^n(STOP)) \\
= \quad & \{\langle\rangle, \langle on\rangle\} \\
& \cup\{\langle on, off\rangle \frown tr \mid tr \in F^n(STOP)\} \\
= \quad & \{\langle\rangle, \langle on\rangle\} \\
& \cup\{\langle on, off\rangle \frown tr \mid tr \in \{\langle on, off\rangle^i \mid 0 \leqslant i \leqslant n\}\} \\
& \cup\{\langle on, off\rangle \frown tr \mid tr \in \{\langle on, off\rangle^i \frown \langle on\rangle \mid 0 \leqslant i < n\}\} \\
= \quad & \{\langle\rangle, \langle on\rangle\} \\
& \cup\{\langle on, off\rangle^i \mid 1 \leqslant i \leqslant n+1\} \\
& \cup\{\langle on, off\rangle^i \frown \langle on\rangle \mid 1 \leqslant i < n+1\} \\
= \quad & \{\langle on, off\rangle^i \mid 0 \leqslant i \leqslant n+1\} \\
& \cup\{\langle on, off\rangle^i \frown \langle on\rangle \mid 0 \leqslant i < n+1\}
\end{aligned}$$

which establishes the result for $n + 1$.

The traces of *LIGHT* are given by $\bigcup_{n \in \mathbb{N}} \mathsf{traces}(F^n(STOP))$, which is given by

$$\begin{aligned} \mathsf{traces}(LIGHT) &= \{\langle on, off\rangle^i \mid i \in \mathbb{N}\} \\ &\quad \cup\{\langle on, off\rangle^i \frown \langle on\rangle \mid i \in \mathbb{N}\} \end{aligned}$$

All finite alternating sequences of *on* and *off* are present. □

The first law for recursion is straightforward.

$$`N = F(N)\text{'} \Rightarrow N =_T F(N) \qquad \langle \text{recursion-unwinding}_T \rangle$$

The law simply captures the discussion above. The left hand side is a statement about the definition of the process $N$: that is a process defined by a recursive definition $N = F(N)$. The right hand side is a result (in the traces model for this chapter) about the traces of the process so defined (and there are corresponding results for the models defined in later chapters). The traces associated with $N$ must be the same as those associated with $F(N)$, and this law states exactly this. It is generally used to 'unwind' recursive definitions, or (used from right to left) to fold them up. For instance, it may be used to show that the process $N = a \to N$ begins with the occurrence of two $a$ events. Initially $N =_T a \to N$ follows from the definition of $N$, and then $a \to N =_T a \to a \to N$ may be deduced from another application of the law. This makes explicit the fact that $N$ begins with two $a$s, since it follows that $N =_T a \to a \to N$.

## Unique fixed points

Functions used in recursive definitions have been seen to have at least one fixed point. The functions correspond to functions on sets of traces. In some cases, there may be exactly one fixed point: exactly one set of traces that the function maps to itself. For instance, in the case of the function $F(Y) = a \to Y$, the only fixed point of the function is the process identified with the set of traces

$$\{\langle a\rangle^n \mid n \in \mathbb{N}\}$$

This corresponds to the only process which satisfies the equation $N =_T a \to N$.

In other cases, there may be a multitude of fixed points. For instance, in the case of the function $F(Y) = Y \,\Box\, a \to STOP$ (on sets of traces) the set of traces

$$\{\langle\rangle, \langle a\rangle\}$$

is a fixed point of the function. However, so is the set of traces

$$\{\langle\rangle, \langle a\rangle, \langle b\rangle\}$$

and indeed any set of traces which contains both $\langle\rangle$ and $\langle a\rangle$ will be a fixed point of the function.

In the case where a CSP function $F$ has a unique fixed point, it follows that *all* CSP processes which are solutions of the equation $Y =_T F(Y)$ must have the same set of traces, since there is only one such set possible. This means that if $N = F(N)$ is a recursively defined process and $F$ has a unique fixed point, and if it can then be shown that another process $P$ satisfies the equation $P =_T F(P)$, then the conclusion $N =_T P$ follows.

## Guardedness

This result is so useful that it is worthwhile exploring a general condition under which a CSP function will indeed have a single fixed point. This condition is *guardedness*, which is present in a function $F$ when any execution of $F(N)$ must perform some visible event before reaching the first invocation of $N$. If this is the case, then every occurrence of the name $N$ is said to be *guarded* in $F(N)$. It will be more precisely defined by the following clauses, which give rules for deducing when a process name $N$ is guarded in a process expression which may contain a number of process names.

The hiding operator may remove guards, by internalizing guarding events. It is the only operator which has this effect: all other operators preserve guardedness when it is already present. Guards are introduced either by means of the event prefixing operators, or else through a sequential composition whose left-hand process (which is providing the guard) does not terminate immediately.

A process name $N$ is *event guarded* in process expression $P$ if either

1. $N$ does not appear in $P$; or
2. (a) $P$ does not contain the hiding operator; and
   (b) every occurrence of process name $N$ is either
      i. within the scope of a prefixing operator (prefix, prefix choice, input, or output); or
      ii. contained within the second argument of a sequential composition whose first argument does not terminate immediately.

A process $P$ terminates immediately if one of its possible traces is $\langle\checkmark\rangle$: it can terminate having performed no actions. Equivalently, if the equation $P =_T P \sqcap SKIP$ can be established for $P$ then $P$ can terminate immediately. The equation provides evidence that $SKIP$ describes one of the possible executions already possible for $P$.

EXAMPLE 4.22

- $N$ is guarded in $a \rightarrow N \sqcap b \rightarrow STOP$, since it is in the scope of the component process $a \rightarrow N$;

- $N$ is guarded in $in?m : T \to N \;|||\; in?n : T \to N$;
- $N$ is guarded in $(a \to SKIP \;\Box\; b \to SKIP);\; N$ because the left-hand process cannot terminate immediately;
- $N$ is guarded in $(SKIP \;\Box\; a \to SKIP);\; b \to N$ because it is in the scope of $b \to N$;
- $N$ is not guarded in $(SKIP \;\Box\; a \to SKIP);\; N$;
- $N$ is not guarded in $a \to b \to (N \setminus \{a\})$ because the expression contains the hiding operator;
- $N$ is guarded in $a \to M$ (where $M \neq N$) because it does not appear;
- $N$ is guarded in $a \to M \setminus \{b\}$ (where $M \neq N$) because it does not appear.

$\Box$

EXAMPLE 4.23

- The function $F(N) = on \to off \to N$ is guarded, both by the event *on* and by the event *off*.
- The function $F(N) = on \to off \to N \;\Box\; off \to on \to N$ is guarded, since each branch of the choice is guarded.
- The function $F(N) = (on \to N) \;|||\; (off \to N)$ is guarded, since each component of the interleaving composition is guarded.
- The function $F(N) = (on \to N) \;\Box\; N$ is not guarded, since one of the components of the choice is unguarded. This means that uniqueness of a fixed point is not guaranteed. In fact, in this case $F$ has a number of distinct fixed points.
- The function $F(N) = STOP \underset{\Sigma}{\|} N$ is not guarded, since the right hand component of the parallel composition is unguarded. However, in this particular case there is only one fixed point.
- The function $F(N) = on \to (N \setminus \{on\})$ is not guarded, because the use of the hiding operator destroys the guard. This function has a number of distinct fixed points.

$\Box$

The second law for recursion can now be given:

$$(F(Y) \text{ guarded} \wedge (F(P_1) =_T P_1) \wedge (F(P_2) =_T P_2)) \Rightarrow P_1 =_T P_2 \qquad \langle \mathsf{UFP}_T \rangle$$

The law is labelled $\mathsf{UFP}_T$, for 'Unique Fixed Point'. It states that if two processes are both fixed points of a guarded function (which must have a unique fixed point), then those two processes must be equivalent. The guardedness is sufficient to establish that the fixed point is unique, but it is not necessary—see Exercise 4.16. The law is often applied when one of the processes is defined in terms of $F$, for example where $P_1 = F(P_1)$; once $P_2$ is shown to be trace equivalent to $F(P_2)$, the equality $P_1 =_T P_2$ follows.

EXAMPLE 4.24 Consider the recursive processes

$$N = (a \rightarrow N) \mathrel{\Box} b \rightarrow STOP$$

and

$$M = a \rightarrow M$$

where the aim is to establish that $N =_T M \mathrel{\triangle} (b \rightarrow STOP)$.

Note first that the function $F_N(Y) = (a \rightarrow Y) \mathrel{\Box} (b \rightarrow STOP)$ used for the definition of $N$ is guarded. This means that $\mathsf{UFP}_T$ is applicable, provided $N =_T F_N(N)$ and $M =_T F_N(M)$. The condition for $N$ follows immediately by Law recursion-unwinding, from its recursive definition. The equivalence will follow if it can be shown that process $M \mathrel{\triangle} (b \rightarrow STOP)$ is a fixed point of the function $F_N(Y)$:

$$\begin{array}{lll} M \mathrel{\triangle} (b \rightarrow STOP) & =_T & \text{by recursion-unwinding} \\ & & (a \rightarrow M) \mathrel{\triangle} (b \rightarrow STOP) \\ & =_T & \text{by } \triangle\text{-step} \\ & & (a \rightarrow (M \mathrel{\triangle} (b \rightarrow STOP))) \mathrel{\Box} b \rightarrow STOP \\ & =_T & \text{by definition of } F_N \\ & & F_N(M \mathrel{\triangle} (b \rightarrow STOP)) \end{array}$$

So the laws for trace equality show that $M \mathrel{\triangle} (b \rightarrow STOP)$ is a fixed point of $F_N$, the function which defines $N$, and so it has the same traces as $N$. □

EXAMPLE 4.25 The ticket and change machine of Example 2.4 is a parallel combination of two recursive processes:

$$MACHINE \quad = \quad TICKET \;{}_{\{cash,ticket\}}\|_{\{cash,change\}}\; CHANGE$$

where the component processes are given by the following recursive definitions:

$$\begin{array}{rcl} TICKET & = & cash \rightarrow ticket \rightarrow TICKET \\ CHANGE & = & cash \rightarrow change \rightarrow CHANGE \end{array}$$

Their alphabets are given by $\alpha T = \alpha(\mathit{TICKET})$ and $\alpha C = \alpha(\mathit{CHANGE})$.

The parallel expansion law $\|$-step may be applied to expand the parallel combination to a sequence of choices:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{TICKET}\ {}_{\alpha T}\|_{\alpha C}\ \mathit{CHANGE}} \\
=_T & \text{by recursion-unwinding} \\
 & (\mathit{cash} \rightarrow \mathit{ticket} \rightarrow \mathit{TICKET})\ {}_{\alpha T}\|_{\alpha C}\ (\mathit{cash} \rightarrow \mathit{change} \rightarrow \mathit{CHANGE}) \\
=_T & \text{by } \|\text{-step} \\
 & \mathit{cash} \rightarrow (\mathit{ticket} \rightarrow \mathit{TICKET}\ {}_{\alpha T}\|_{\alpha C}\ \mathit{change} \rightarrow \mathit{CHANGE}) \\
=_T & \text{by } \|\text{-step} \\
 & \mathit{cash} \rightarrow (\ \ \mathit{ticket} \rightarrow (\mathit{TICKET}\ {}_{\alpha T}\|_{\alpha C}\ \mathit{change} \rightarrow \mathit{CHANGE}) \\
 & \qquad\quad \mid \mathit{change} \rightarrow ((\mathit{ticket} \rightarrow \mathit{TICKET})\ {}_{\alpha T}\|_{\alpha C}\ \mathit{CHANGE})) \\
=_T & \text{by recursion-unwinding} \\
 & \mathit{cash} \rightarrow \\
 & \quad\ \mathit{ticket} \rightarrow \\
 & \qquad ((\mathit{cash} \rightarrow \mathit{ticket} \rightarrow \mathit{TICKET})\ {}_{\alpha T}\|_{\alpha C}\ \mathit{change} \rightarrow \mathit{CHANGE}) \\
 & \quad \mid \mathit{change} \rightarrow \\
 & \qquad ((\mathit{ticket} \rightarrow \mathit{TICKET})\ {}_{\alpha T}\|_{\alpha C}\ \mathit{cash} \rightarrow \mathit{change} \rightarrow \mathit{CHANGE}) \\
=_T & \text{by } \|\text{-step} \\
 & \mathit{cash} \rightarrow \\
 & \quad\ \mathit{ticket} \rightarrow \\
 & \qquad \mathit{change} \rightarrow ((\mathit{cash} \rightarrow \mathit{ticket} \rightarrow \mathit{TICKET})\ {}_{\alpha T}\|_{\alpha C}\ \mathit{CHANGE}) \\
 & \quad \mid \mathit{change} \rightarrow \\
 & \qquad \mathit{ticket} \rightarrow (\mathit{TICKET}\ {}_{\alpha T}\|_{\alpha C}\ (\mathit{cash} \rightarrow \mathit{change} \rightarrow \mathit{CHANGE})) \\
=_T & \text{by recursion-unwinding} \\
 & \mathit{cash} \rightarrow (\ \ \mathit{ticket} \rightarrow \mathit{change} \rightarrow (\mathit{TICKET}\ {}_{\alpha T}\|_{\alpha C}\ \mathit{CHANGE}) \\
 & \qquad\quad \mid \mathit{change} \rightarrow \mathit{ticket} \rightarrow (\mathit{TICKET}\ {}_{\alpha T}\|_{\alpha C}\ \mathit{CHANGE}))
\end{array}$$

Observe that Law $\|$-step applies to processes of the form $x : C \rightarrow P(x)$, which is why the recursive definitions of *TICKET* and *CHANGE* must be unfolded (by an application of recursion-unwinding) before that law can apply.

The above equivalence establishes that $\mathit{TICKET}\ {}_{\alpha T}\|_{\alpha C}\ \mathit{CHANGE}$ is a fixed point of the guarded function

$$\begin{array}{lcll}
F(Y) & = & \mathit{cash} \rightarrow & \mathit{ticket} \rightarrow \mathit{change} \rightarrow Y \\
 & & & \mid \mathit{change} \rightarrow \mathit{ticket} \rightarrow Y
\end{array}$$

and so it is trace equivalent to the recursively defined sequential process

$$\begin{array}{lcll}
\mathit{MACHINE}' & = & \mathit{cash} \rightarrow & \mathit{ticket} \rightarrow \mathit{change} \rightarrow \mathit{MACHINE}' \\
 & & & \mid \mathit{change} \rightarrow \mathit{ticket} \rightarrow \mathit{MACHINE}'
\end{array}$$

This provides a description of a process equivalent to *MACHINE*, but with the parallelism removed. □

EXAMPLE 4.26 The stop-and-wait protocol of Example 3.2 is defined as

$$SAWP \quad = \quad (S \parallel R) \setminus (mid.T \cup \{ack\})$$

where the sender and receiver are defined by $S$ and $R$ respectively.

$$\begin{aligned} S &= in?x : T \to mid!x \to ack \to S \\ R &= mid?y : T \to out!y \to ack \to R \end{aligned}$$

Several applications of Law ||-step yield that

$$S \parallel R \quad =_T \quad in?x : T \to mid!x \to out!x \to ack \to (S \parallel R)$$

Hiding the *mid* and *ack* channels has the following effect:

$$\begin{aligned} & (S \parallel R) \setminus (mid.T \cup \{ack\}) \\ =_T \quad & \text{by the previous equivalence} \\ & (in?x : T \to mid!x \to out!x \to ack \to (S \parallel R)) \setminus (mid.T \cup \{ack\}) \\ =_T \quad & \text{by hide-step 2} \\ & in?x : T \to (mid!x \to out!x \to ack \to (S \parallel R)) \setminus (mid.T \cup \{ack\}) \\ =_T \quad & \text{by hide-step 3} \\ & in?x : T \to (out!x \to ack \to (S \parallel R)) \setminus (mid.T \cup \{ack\}) \\ =_T \quad & \text{by hide-step 2} \\ & in?x : T \to out!x \to (ack \to (S \parallel R)) \setminus (mid.T \cup \{ack\}) \\ =_T \quad & \text{by hide-step 3} \\ & in?x : T \to out!x \to ((S \parallel R)) \setminus (mid.T \cup \{ack\}) \end{aligned}$$

Hence $(S \parallel R) \setminus (mid.T \cup \{ack\})$ is a fixed point of the guarded function $F(Y) = in?x : T \to out!x \to Y$ which is the function used to define the one-place buffer *COPY* of Example 1.15. It follows from UFP$_T$ that

$$(S \parallel R) \setminus (mid.T \cup \{ack\}) \quad =_T \quad COPY$$

which establishes that the stop-and-wait protocol really does implement a one-place buffer. □

## Mutual recursion

The approach taken to mutually defined families of processes is a generalization of the approach taken above. In a mutual recursion, each of the processes is defined in terms of a number of the processes. The general case of a mutual recursion is concerned with a family, or *vector* of process names $\underline{N}$. Each member of the family is referred to with a particular index $i$, and the $i$th component of vector $\underline{N}$ is referred to as $\underline{N}_i$.

For example, three processes *HIGH*, *MID*, and *LOW* may be defined by means of a mutual recursion. This family of process names may be considered in terms of the vector $\underline{N}$, indexed by the set $\{0, 1, 2\}$, with $\underline{N}_0 = HIGH$, $\underline{N}_1 = MID$, and $\underline{N}_2 = LOW$. Another way of writing this is $\underline{N} = \langle HIGH, MID, LOW \rangle$.

The mutual recursion defining $\underline{N}$ uses a corresponding family, or vector of functions $\underline{F}$, with the same indexing set. Each element of $\underline{F}$ is a CSP function on a vector of processes which gives a CSP process as output. Hence the entire vector $\underline{F}$ is a function from vectors of processes to vectors of processes. The recursive definition then takes the form $\underline{N} = \underline{F}(\underline{N})$.

For instance, the three processes *HIGH*, *MID*, and *LOW* might be defined as follows:

$$\begin{aligned} HIGH &= around \rightarrow HIGH \\ &\quad \mid down \rightarrow MID \\[1ex] MID &= up \rightarrow HIGH \\ &\quad \mid down \rightarrow LOW \\[1ex] LOW &= jump \rightarrow HIGH \\ &\quad \mid up \rightarrow MID \\ &\quad \mid down \rightarrow up \rightarrow LOW \end{aligned}$$

In this case, each of *HIGH*, *MID*, and *LOW* is defined in terms of a function of the three processes, and the traces associated with each of these processes is dependent on those traces associated with the others.

All three processes are *simultaneously* defined by the recursive equations. There is a CSP function associated with each process, which in each case is a function of three arguments:

$$\begin{aligned} F_0(H, M, L) &= around \rightarrow H \mid down \rightarrow M \\ F_1(H, M, L) &= up \rightarrow H \mid down \rightarrow L \\ F_2(H, M, L) &= jump \rightarrow H \mid up \rightarrow M \mid down \rightarrow up \rightarrow L \end{aligned}$$

Observe that some functions do not mention all of the names in their definition.

The recursive definitions may then be rewritten as

$$\begin{aligned} HIGH &= F_0(HIGH, MID, LOW) \\ MID &= F_1(HIGH, MID, LOW) \\ LOW &= F_2(HIGH, MID, LOW) \end{aligned}$$

or alternatively as

$$\underline{N} = \langle F_0(\underline{N}), F_1(\underline{N}), F_2(\underline{N}) \rangle$$

or alternatively as

$$\underline{N} = \underline{F}(\underline{N})$$

To calculate the traces of each process defined by the mutual recursion, the same approach is taken as in the treatment of single recursion, taking as the starting point that the only trace known to be in each component is the empty trace $\langle\rangle$. The process *STOP* will be the first approximation to these processes, and successive unwindings of the definition provide successive approximations: the point for beginning the recursive unwinding is the vector $\underline{STOP}$, and successive unwindings are then given by $\underline{F}^n(\underline{STOP})$. This allows successive approximations to each process $\underline{N}_i$ to be built up as the *i*th component $\underline{F}^n(\underline{STOP})_i$ of the approximations. The traces of $\underline{N}_i$, where $\underline{N} = \underline{F}(\underline{N})$, are given by

$$\text{traces}(\underline{N}_i) = \bigcup(\text{traces}(\underline{F}^n(\underline{STOP})))_i$$

In the case of *HIGH*, *MID*, and *LOW*, the family of functions $F_0$, $F_1$, and $F_2$ together define a mapping from a family of three processes *H*, *M*, and *L* to another family of three processes $F_0(H, M, L)$, $F_1(H, M, L)$, and $F_2(H, M, L)$. To calculate the traces associated with each of the processes defined recursively as part of this family, it is necessary to begin with the family *STOP*, *STOP*, and *STOP*, similarly to the starting point for a single recursion: that $\langle\rangle$ is the only trace known to be in each process. Unwinding the recursive definitions once yields the family $H_1 = F_0(STOP, STOP, STOP)$, $M_1 = F_1(STOP, STOP, STOP)$, and $L_1 = F_2(STOP, STOP, STOP)$, which are written in full as follows:

$$\begin{aligned} H_1 &= around \rightarrow STOP \\ &\quad \mid down \rightarrow STOP \\[1em] M_1 &= up \rightarrow STOP \\ &\quad \mid down \rightarrow STOP \\[1em] L_1 &= jump \rightarrow STOP \\ &\quad \mid up \rightarrow STOP \\ &\quad \mid down \rightarrow up \rightarrow STOP \end{aligned}$$

and their traces are then given as follows:

$$\begin{aligned} \text{traces}(H_1) &= \{\langle\rangle, \langle around\rangle, \langle down\rangle\} \\ \text{traces}(M_1) &= \{\langle\rangle, \langle up\rangle, \langle down\rangle\} \\ \text{traces}(L_1) &= \{\langle\rangle, \langle jump\rangle, \langle up\rangle, \langle down\rangle, \langle down, up\rangle\} \end{aligned}$$

The next unwinding yields the three processes:

$$\begin{aligned} H_2 &= F_0(H_1, M_1, L_1) \\ M_2 &= F_1(H_1, M_1, L_1) \\ L_2 &= F_2(H_1, M_1, L_1) \end{aligned}$$

Observe that all of the processes from the first unwinding are required in order to calculate the processes reached after the second unwinding. Each stage is calculated from the previous stage:

$$\begin{aligned} H_{i+1} &= F_0(H_i, M_i, L_i) \\ M_{i+1} &= F_1(H_i, M_i, L_i) \\ L_{i+1} &= F_2(H_i, M_i, L_i) \end{aligned}$$

The traces contained in the recursive processes are precisely those reached after some finite number of unwindings. Hence the traces of the process *HIGH* are the traces of all the $H_i$ approximations, and the traces of *MID* and *LOW* are obtained similarly:

$$\begin{aligned} \mathsf{traces}(HIGH) &= \bigcup_i \mathsf{traces}(H_i) \\ \mathsf{traces}(MID) &= \bigcup_i \mathsf{traces}(M_i) \\ \mathsf{traces}(LOW) &= \bigcup_i \mathsf{traces}(L_i) \end{aligned}$$

EXAMPLE 4.27 A collection of three processes $\langle C_i \mid 0 \leqslant i \leqslant 2 \rangle$ indexed by the set $\{0, 1, 2\}$ may be defined as follows:

$$C_i = \begin{cases} around \rightarrow C_i & \text{if } i = 0 \\ \left( \begin{array}{l} around \rightarrow C_i \\ \Box\ down \rightarrow C_{i-1} \end{array} \right) & \text{otherwise} \end{cases}$$

The conditional statement simply allows the expression of a family of process definitions as a single parameterized definition. It is shorthand for the family of process definitions

$$\begin{aligned} C_0 &= around \rightarrow C_0 \\ C_1 &= around \rightarrow C_1 \Box down \rightarrow C_0 \\ C_2 &= around \rightarrow C_2 \Box down \rightarrow C_1 \end{aligned}$$

The notation $C_{i,j}$ will refer to the $j$th approximation to $C_i$. Since the starting point for the unwindings is *STOP*, each $C_{i,0}$ must be *STOP*. The first approximations are

$$\begin{aligned} C_{0,1} &= around \rightarrow STOP \\ C_{1,1} &= around \rightarrow STOP \Box down \rightarrow STOP \\ C_{2,1} &= around \rightarrow STOP \Box down \rightarrow STOP \end{aligned}$$

and their traces are as follows:

$$\begin{aligned}
\mathsf{traces}(C_{0,1}) &= \{\langle\rangle, \langle \mathit{around}\rangle\}\\
\mathsf{traces}(C_{1,1}) &= \{\langle\rangle, \langle \mathit{around}\rangle, \langle \mathit{down}\rangle\}\\
\mathsf{traces}(C_{2,1}) &= \{\langle\rangle, \langle \mathit{around}\rangle, \langle \mathit{down}\rangle\}
\end{aligned}$$

The second approximations are

$$\begin{aligned}
C_{0,2} &= \mathit{around} \rightarrow C_{0,1}\\
C_{1,2} &= \mathit{around} \rightarrow C_{1,1} \;\square\; \mathit{down} \rightarrow C_{0,1}\\
C_{2,2} &= \mathit{around} \rightarrow C_{2,1} \;\square\; \mathit{down} \rightarrow C_{1,1}
\end{aligned}$$

The traces of the second approximations are derived from those of the first:

$$\begin{aligned}
\mathsf{traces}(C_{0,2}) &= \{\langle\rangle, \langle \mathit{around}\rangle, \langle \mathit{around}, \mathit{around}\rangle\}\\
\mathsf{traces}(C_{1,2}) &= \{\langle\rangle, \langle \mathit{around}\rangle, \langle \mathit{down}\rangle, \langle \mathit{around}, \mathit{around}\rangle,\\
&\qquad \langle \mathit{around}, \mathit{down}\rangle, \langle \mathit{down}, \mathit{around}\rangle\}\\
\mathsf{traces}(C_{2,2}) &= \{\langle\rangle, \langle \mathit{around}\rangle, \langle \mathit{down}\rangle, \langle \mathit{around}, \mathit{around}\rangle,\\
&\qquad \langle \mathit{around}, \mathit{down}\rangle, \langle \mathit{down}, \mathit{around}\rangle, \langle \mathit{down}, \mathit{down}\rangle\}
\end{aligned}$$

In general, the traces of the *n*th approximations will be

$$\begin{aligned}
\mathsf{traces}(C_{0,n}) &= \{tr \mid tr \in \{\mathit{around}\}^* \land \#tr \leqslant n\}\\
\mathsf{traces}(C_{1,n}) &= \{tr \mid tr \in \{\mathit{around}, \mathit{down}\}^* \land \#tr \leqslant n \land tr \downarrow \mathit{down} \leqslant 1\}\\
\mathsf{traces}(C_{2,n}) &= \{tr \mid tr \in \{\mathit{around}, \mathit{down}\}^* \land \#tr \leqslant n \land tr \downarrow \mathit{down} \leqslant 2\}
\end{aligned}$$

The traces of each $C_i$ is the union of the traces of the approximations: $\mathsf{traces}(C_i) = \bigcup_n \mathsf{traces}(C_{i,n})$:

$$\begin{aligned}
\mathsf{traces}(C_0) &= \{tr \mid tr \in \{\mathit{around}\}^*\}\\
\mathsf{traces}(C_1) &= \{tr \mid tr \in \{\mathit{around}, \mathit{down}\}^* \land tr \downarrow \mathit{down} \leqslant 1\}\\
\mathsf{traces}(C_2) &= \{tr \mid tr \in \{\mathit{around}, \mathit{down}\}^* \land tr \downarrow \mathit{down} \leqslant 2\}
\end{aligned}$$

Each $C_i$ can do a maximum of *i down* events. □

## Unique fixed points

A mutual recursion $\underline{N} = \underline{P}$ is event guarded in $\underline{F}$ if any name $N_i$ appearing on the right hand side of any equation has its corresponding process $P_i$ event guarded for each name: whenever $N_i$ appears in any of the $P_j$, then $P_i$ must be event guarded for each $N_j$. If a process name $N_i$ does not appear in any of the right hand processes $P_j$, then there will never be any recursive calls to the process $P_i$ and so it need not itself be guarded—only those processes which will be recursively called need to provide guards. For example, in the recursive definition

$$\begin{array}{rcl} START & = & LEFT \\ LEFT & = & right \to RIGHT \\ RIGHT & = & left \to LEFT \end{array}$$

only process variables *LEFT* and *RIGHT* appear in any of the right hand process expressions, and both processes corresponding to those variables are event guarded. It follows that the mutual recursion is event guarded. Observe that the function $F_{START}(S, L, R) = L$ is not event guarded. However, *START* does not appear on the right hand side of the definition, and so this family of definitions is event guarded.

An event guarded function has a unique fixed point. This is a direct generalization of the same result for the case of a single recursion. It allows two families of processes to be shown to be equal, by establishing that they are both fixed points of a function whose variables are guarded. This is expressed in the third rule we give for recursion, which is a more general form of the Unique Fixed Point result:

$$\boxed{(\underline{F}(\underline{Y}) \text{ guarded} \land (\underline{F}(\underline{P_1}) =_T \underline{P_1}) \land (\underline{F}(\underline{P_2}) =_T \underline{P_2})) \Rightarrow \underline{P_1} =_T \underline{P_2} \qquad \langle \mathsf{UFP}_T \rangle}$$

EXAMPLE 4.28 This rule may be used for simplifying an interleaved combination of recursive processes.

$$\begin{array}{rcl} BOUNCE & = & up \to down \to BOUNCE \\ WOBBLE & = & left \to right \to WOBBLE \end{array}$$

The process $BOUNCE \mid\mid\mid WOBBLE$ cannot be written as a single recursive process. Expanding the interleaving yields

$$\begin{array}{ll} & BOUNCE \mid\mid\mid WOBBLE \\ =_T & \text{recursion-unwinding} \\ & (up \rightarrow down \rightarrow BOUNCE) \mid\mid\mid (left \rightarrow right \rightarrow WOBBLE) \\ =_T & \mid\mid\mid\text{-step} \\ & up \rightarrow ((down \rightarrow BOUNCE) \mid\mid\mid (left \rightarrow right \rightarrow WOBBLE)) \\ & \Box\ left \rightarrow ((up \rightarrow down \rightarrow BOUNCE) \mid\mid\mid (right \rightarrow WOBBLE)) \\ =_T & \text{rule recursion-unwinding} \\ & up \rightarrow ((down \rightarrow BOUNCE) \mid\mid\mid WOBBLE) \\ & \Box\ left \rightarrow (BOUNCE \mid\mid\mid right \rightarrow WOBBLE) \end{array}$$

Similarly

$$\begin{array}{ll} \multicolumn{2}{l}{(down \rightarrow BOUNCE) \mid\mid\mid WOBBLE} \\ =_T & down \rightarrow (BOUNCE \mid\mid\mid WOBBLE) \\ & \Box\ left \rightarrow ((down \rightarrow BOUNCE) \mid\mid\mid (right \rightarrow WOBBLE)) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{(down \rightarrow BOUNCE) \mid\mid\mid (right \rightarrow WOBBLE)} \\ =_T & down \rightarrow (BOUNCE \mid\mid\mid (right \rightarrow WOBBLE)) \\ & \Box\ right \rightarrow ((down \rightarrow BOUNCE) \mid\mid\mid WOBBLE) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{BOUNCE \mid\mid\mid (right \rightarrow WOBBLE)} \\ =_T & up \rightarrow ((down \rightarrow BOUNCE) \mid\mid\mid (right \rightarrow WOBBLE)) \\ & \Box\ right \rightarrow (BOUNCE \mid\mid\mid WOBBLE) \end{array}$$

The result is a vector of four parallel processes

$$\underline{BW} = \left\langle \begin{array}{l} BOUNCE \mid\mid\mid WOBBLE \\ BOUNCE \mid\mid\mid (right \rightarrow WOBBLE) \\ (down \rightarrow BOUNCE) \mid\mid\mid (right \rightarrow WOBBLE) \\ (down \rightarrow BOUNCE) \mid\mid\mid WOBBLE \end{array} \right\rangle$$

given as functions of each other. The functions, given in terms of processes $\underline{N}$ indexed from 0 to 3, may be extracted and made explicit:

$$\begin{array}{lcl} F_0(\underline{N}) & = & (up \rightarrow \underline{N}_1) \Box (left \rightarrow \underline{N}_3) \\ F_1(\underline{N}) & = & (down \rightarrow \underline{N}_0) \Box (left \rightarrow \underline{N}_2) \\ F_2(\underline{N}) & = & (down \rightarrow \underline{N}_3) \Box (right \rightarrow \underline{N}_1) \\ F_3(\underline{N}) & = & (up \rightarrow \underline{N}_2) \Box (right \rightarrow \underline{N}_0) \end{array}$$

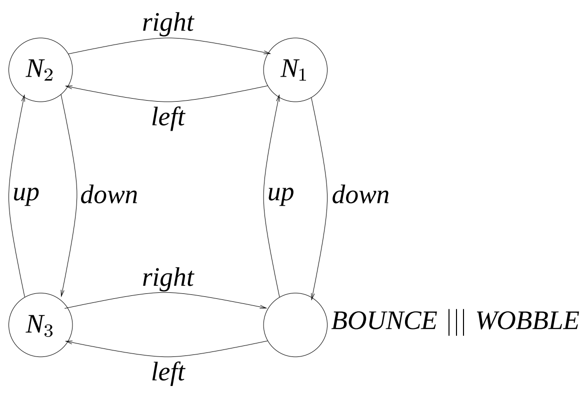


**Fig. 4.16** The transitions of $BOUNCE \mid\mid\mid WOBBLE$

Then if $\underline{N} = \underline{F}(\underline{N})$, then $\mathsf{UFP}_T$ allows the deduction that $\underline{BW} =_T \underline{N}$, since they are both fixed points of $\underline{F}$. This means that $BOUNCE \mid\mid\mid WOBBLE =_T \underline{N}_0$, so it can be rewritten as a mutual recursion to remove explicit parallelism. Its state space is illustrated in Figure 4.16. □

EXAMPLE 4.29 Suppose that the counter of Example 1.19 is adapted so that it can only perform increments, and no longer has decrements as an option:

$$COUNTUP(n) \quad = \quad increment \rightarrow COUNTUP(n+1)$$

Then any of the $COUNTUP(n)$ processes can perform arbitrary sequences of *increment* events, and no other events. It appears that each process is equivalent to $N = increment \rightarrow N$. In fact, this can be shown by setting a vector of processes $\underline{N}$ so that each $\underline{N}_i = N$ as defined by the recursion. Then

$$\begin{aligned} \underline{N}_n &=_T N \\ &=_T increment \rightarrow N \\ &=_T increment \rightarrow \underline{N}_{n+1} \end{aligned}$$

so the vector $\underline{N}$ is a fixed point of the guarded function used to define $\underline{COUNTUP}$, and hence by $\mathsf{UFP}_T$ the two families of processes must be the same. It follows that each $COUNTUP(n)$ is trace equivalent to $N = increment \rightarrow N$. □

EXAMPLE 4.30 The counter of Example 1.19 is defined as a mutual recursion, which maintains the difference between the numbers of *increment*s and *decrement*s, and allows *decrement* provided this number is strictly positive:

$$\begin{aligned} COUNT(0) &= increment \rightarrow COUNT(1) \\ COUNT(n+1) &= increment \rightarrow COUNT(n+2) \\ &\quad \Box\ decrement \rightarrow COUNT(n) \end{aligned}$$

The following process expression *SPAWN* describes a process with the same behaviour as $COUNT(0)$: every time *increment* is performed, a process is spawned which can independently perform *decrement* exactly once.

$$SPAWN \quad = \quad increment \to (SPAWN \mid\mid\mid decrement \to STOP)$$

To show that $SPAWN =_T COUNT(0)$ it is sufficient to find a family of processes $\underline{S}$ which is a fixed point of the function defining $\underline{COUNT}$, and such that $\underline{S}_0 = SPAWN$.

Define

$$\underline{S}_n \quad = \quad SPAWN \mid\mid\mid (\big|\big|\big|_{i=1}^{n}(decrement \to STOP))$$

Observe (recalling Exercise 4.7) that

$$\big|\big|\big|_{i=1}^{n+1}(decrement \to STOP) \quad =_T \quad decrement \to (\big|\big|\big|_{i=1}^{n} decrement \to STOP)$$

Then

$$\begin{array}{rcl} S_0 & = & SPAWN \\ & =_T & increment \to (SPAWN \mid\mid\mid decrement \to STOP) \\ & =_T & increment \to S_1 \\ \\ \underline{S}_{n+1} & = & SPAWN \mid\mid\mid \big|\big|\big|_{i=1}^{n+1}(decrement \to STOP) \\ & =_T & (increment \to (SPAWN \mid\mid\mid decrement \to STOP)) \\ & & \mid\mid\mid decrement \to \big|\big|\big|_{i=1}^{n}(decrement \to STOP) \\ & =_T & increment \to (\ SPAWN \mid\mid\mid decrement \to STOP \\ & & \qquad\qquad \mid\mid\mid \big|\big|\big|_{i=1}^{n+1}(decrement \to STOP)) \\ & & \Box\ decrement \to (SPAWN \mid\mid\mid \big|\big|\big|_{i=1}^{n} decrement \to STOP) \\ & =_T & increment \to \underline{S}_{n+2} \Box\ decrement \to \underline{S}_n \end{array}$$

The $\underline{S}_n$ meet the same guarded equations as the $COUNT(n)$, so they must be the same by UFP$_T$. This means that $\underline{S}_0 =_T SPAWN =_T COUNT(0)$. □

## 4.3 TESTING

A completely different approach to understanding CSP processes can be given directly in terms of the operational semantics. Some very simple and natural notions of how processes should be distinguished and when they should be considered equivalent can be given purely in terms of the possibility of a single event's occurrence within some test. It turns out that this approach gives the same results as the denotational traces model for CSP, and this equivalence provides a better understanding of the traces model.

Processes may be analyzed in terms of how they behave in particular contexts. The response of processes to particular *tests* can be used to compare different process expressions, and two processes will be considered equivalent if each of their responses to any test is the same: two processes are judged equivalent if no test can tell them apart.

Tests will themselves be constructed from the CSP language, extended with a special process *SUCCESS* which will be used to indicate that an execution has succeeded by means of performing a special 'success' event $\omega \notin \Sigma^{\checkmark}$. The operational semantics of *SUCCESS* is given by

$$\boxed{\frac{}{SUCCESS \overset{\omega}{\rightarrow} STOP}}$$

For example, the test

$$T_0 = a \rightarrow SUCCESS$$

will reach the success state after the occurrence of an $a$ event, and the test

$$T_1 = a \rightarrow SUCCESS \;\square\; b \rightarrow SUCCESS$$

succeeds after either an $a$ or a $b$ event.

A test $T$ can be used to test a CSP process $P$ by running $P$ and $T$ together in parallel, and hiding all events apart from the success event $\omega$:

$$(P \underset{\Sigma}{\parallel} T) \setminus \Sigma$$

Since all events apart from $\omega$ are hidden, the only visible event that this combination might possibly perform is an $\omega$ event, which denotes a successful execution[1]. The test $T_0$ above will succeed if $a$ is one of the first events that the tested process $P$ can perform. For example, the process $a \rightarrow b \rightarrow STOP$ will have a successful execution with $T_0$, since $(a \rightarrow SUCCESS \underset{\Sigma}{\parallel} a \rightarrow b \rightarrow STOP) \setminus \Sigma$ can perform $\omega$ after the internal synchronization on $a$. However, $b \rightarrow a \rightarrow STOP$ has no successful execution.

[1]Unlike other events, $\omega$ may also occur after termination to allow testing for termination. A new construct $SKIP_\omega$ which has one transition $SKIP_\omega \overset{\checkmark}{\longrightarrow} SUCCESS$ may also be used in the construction of tests

The approach of *may testing* is concerned with the possibility of successful execution of a process under a test. The notation $P$ **may** $T$ indicates that there is some successful execution of $P$ under test $T$:

$$P \textbf{ may } T \quad = \quad ((P \underset{\Sigma}{\parallel} T) \setminus \Sigma) \overset{\langle \omega \rangle}{\Longrightarrow}$$

Two processes $P_1$ and $P_2$ will be distinguished under may testing if there is some test $T$ which distinguishes them: in other words, either $P_1$ **may** $T$ and $\neg\ (P_2 \textbf{ may } T)$ or else $P_2$ **may** $T$ and $\neg\ (P_1 \textbf{ may } T)$. If there is no such test, then they will be considered equivalent under may testing:

$$P_1 \equiv_{may} P_2 \quad = \quad \forall\, T \bullet P_1 \textbf{ may } T \Leftrightarrow P_2 \textbf{ may } T$$

For example, $a \rightarrow b \rightarrow STOP$ and $b \rightarrow a \rightarrow STOP$ are distinguished by the test $a \rightarrow SUCCESS$. On the other hand, the processes $a \rightarrow STOP \sqcap b \rightarrow STOP$ and $a \rightarrow STOP \;\Box\; b \rightarrow STOP$ may pass exactly the same tests, and so will be equivalent under may testing.

$$\begin{array}{rcl} (a \rightarrow b \rightarrow STOP & \not\equiv_{may} & b \rightarrow a \rightarrow STOP) \\ a \rightarrow STOP \;\Box\; b \rightarrow STOP & \equiv_{may} & a \rightarrow STOP \sqcap b \rightarrow STOP \end{array}$$

The definition of may testing equivalence, although it provides a natural approach to process equivalence, would be cumbersome to use in practice since it requires consideration of all possible tests in order to show that two processes are equivalent. A significant result is that this form of equivalence is exactly the same as traces equivalence:

$$P_1 \equiv_{may} P_2 \quad \Leftrightarrow \quad \mathsf{traces}(P_1) = \mathsf{traces}(P_2)$$

This means that the traces model provides exactly the framework required for establishing may testing equivalence. Technically, the traces model is said to be *fully abstract* with respect to may testing. The traces model contains exactly the information required to compare and contrast processes in terms of how they might behave in a may testing context. If two processes have different trace sets, then there will be some test which can distinguish them, and conversely if they have exactly the same trace sets, then no test can distinguish them.

Testing also naturally gives rise to a refinement relation:

$$P_1 \sqsubseteq_{may} P_2 \quad = \quad \forall\, T \bullet \neg\ (P_1 \textbf{ may } T) \Rightarrow \neg\ (P_2 \textbf{ may } T)$$

This states that if a specification process $P_1$ is unable to pass a given test $T$, then this indicates a limitation on what $P_1$ considered as a specification allows, and so no implementation should be able to pass the test $T$ either. Although this definition is intuitive, it would be laborious to apply directly in practice because it would involve consideration of an infinity of tests. However, it does provide an alternative understanding of refinement in the traces model (introduced in the next Chapter, on Page 166), since these two characterizations of refinement are equivalent:

$$P_1 \sqsubseteq_{may} P_2 \quad \Leftrightarrow \quad P_1 \sqsubseteq_T P_2$$

## 4.4 CONGRUENCE

May testing defines a way of identifying when two processes should be considered equivalent. All that is required for a relation to be an equivalence relation $\equiv$ on a set $S$ is that it is reflexive, symmetric, and transitive:

**reflexive:** $\forall x : S \bullet x \equiv x$

**symmetric:** $\forall x, y : S \bullet x \equiv y \Leftrightarrow y \equiv x$

**transitive:** $\forall x, y, z : S \bullet x \equiv y \wedge y \equiv z \Rightarrow x \equiv z$

For example, an equivalence on the positive integers $\mathbb{N}^+$ might identify two integers if they are the same modulo 3, so 1 and 4 are considered equivalent modulo 3. This could be written as $1 \equiv_3 4$.

A *congruence* is a stronger form of equivalence, which is preserved by the operations on that set. In other words, if the same operation is carried out on two elements that are equivalent, then the two results should again be equivalent. Equivalence modulo 3 on $\mathbb{N}^+$ is a congruence when the only operations of interest are addition and multiplication. However, if exponentiation is also allowed as an operation, then it is no longer a congruence, since for example $1 \equiv 4$ but $\neg(2^1 \equiv_3 2^4)$. This operation allows a context in which some equivalent numbers can be distinguished, by giving a different result when applied to each of them. Whether or not an equivalence on $S$ is also a congruence on $S$ depends on the operations allowed.

Any equivalence $\equiv$ on a set $S$ will have an associated congruence $\cong$: the weakest congruence which is stronger than it. Two conditions must hold: firstly $\cong$ is stronger than $\equiv$ if

1. $\forall x, y : S \bullet x \cong y \Rightarrow x \equiv y$

For example, equivalence modulo 6, is stronger than equivalence modulo 3—if two numbers are equivalent modulo 6, then they are equivalent modulo 3: $x \equiv_6 y \Rightarrow x \equiv_3 y$. However, equivalence modulo 2 is not stronger than equivalence modulo 3, since for example $2 \equiv_2 10$ but $\neg(2 \equiv_3 10)$.

Being the *weakest* congruence stronger than $\equiv$-equivalence means that the associated congruence should be weaker than any other congruence $\cong'$ which is stronger than $\equiv$:

2. If $\cong'$ is a congruence stronger than $\equiv$ then it is also stronger than $\cong$.

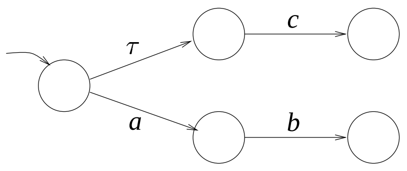
For example, under the operations of addition, multiplication, and exponentiation, equivalence modulo 6 is the weakest congruence stronger than equivalence modulo 3.

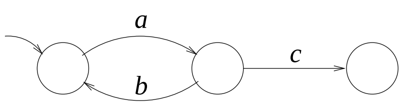
Generally in considering CSP processes, an equivalence relation itself is not so useful as its associated congruence relation. This is because processes are components of systems, and so their behaviour in a variety of contexts is more important than their behaviour in isolation. An equivalence relation allows natural distinctions between processes to be expressed, and the associated congruence establishes what else is required for equivalence to be preserved in all contexts.

A denotational model such as the traces model automatically provides a congruence for the language simply by virtue of the compositional way the language semantics is defined. Since trace equivalence is the same as may testing equivalence, this means that may testing equivalence must also be a congruence, so any two processes which cannot be distinguished by any test can replace each other within any process context without changing the overall result.

## Exercises

EXERCISE 4.1 What are the traces associated with the following finite state machines ?





EXERCISE 4.2 Give the traces of the following processes:

1. $a \to STOP \mid b \to c \to STOP$
2. $a \to STOP \mid b \to (c \to STOP \mid d \to STOP)$
3. $a \to STOP \,\Box\, a \to b \to STOP$
4. $\bigsqcap_{n \in \mathbb{Z}} out!n \to out!(n^2) \to STOP$
5. $a \to b \to RUN_{\{a,b\}}$
6. $(a \to b \to STOP) \sqcap RUN_{\{a,c\}}$

EXERCISE 4.3 Why is the empty trace explicitly included in the definition of indexed external choice? Why is it not also included in the definition of indexed internal choice?

EXERCISE 4.4 Prove the soundness of Law $\Box$-zero from the definition given for $\mathsf{traces}(P_1 \,\Box\, P_2)$.

EXERCISE 4.5 Prove the soundness of Law $\Box$-unit.

EXERCISE 4.6 Simplify each of the following where possible, where $\alpha(P) \subseteq A$:

1. $P \ {}_A\|_A \ RUN_{A^{\checkmark}}$
2. $P \ {}_A\|_B \ RUN_{B^{\checkmark}}$ (where $B \subseteq A$)
3. $P \ {}_A\|_B \ RUN_{B^{\checkmark}}$ (where $B \not\subseteq A$)
4. $P \ {}_A\|_B \ RUN_{(B\setminus A)^{\checkmark}}$ (where $A \subseteq B$)
5. $P \ {}_A\|_A \ STOP$
6. $P \ {}_A\|_B \ STOP$ (where $A \subseteq B$)
7. $P \ {}_A\|_B \ STOP$ (where $A \not\subseteq B$)

EXERCISE 4.7 Given that $|||_{i=1}^{1} \ a \to STOP =_T a \to STOP$, prove (by induction on $n$) that

$$|||_{i=1}^{n+1} \ a \to STOP \quad =_T \quad a \to (|||_{i=1}^{n} \ a \to STOP)$$

EXERCISE 4.8 What are the traces of the following processes?

1. $coin \to change \to SKIP \ \Box \ coin \to ticket \to SKIP$
2. $coin \to change \to SKIP \ \| \ coin \to ticket \to SKIP$
3. $coin \to change \to SKIP \ ||| \ coin \to ticket \to SKIP$
4. $coin \to change \to SKIP \ \triangle \ coin \to ticket \to SKIP$

EXERCISE 4.9 What are the traces of $P_1$, $P_2$, and $P_1 \ {}_A\|_B \ P_2$ of Example 4.13 of Page 101.

EXERCISE 4.10 Is $f(f^{-1}(P)) =_T P$ true for any alphabet renaming function $f$ ? How about $f^{-1}(f(P)) =_T P$ ? In each case give a proof or a counterexample.

EXERCISE 4.11 What are the traces of the recursive process *TASKS* of Example 3.24?

EXERCISE 4.12 Calculate the traces of the recursive process

$$P \quad = \quad up \to (P \ \Box \ SKIP); \ down \to SKIP$$

EXERCISE 4.13 Calculate the traces of the recursive process

$$P \quad = \quad in?x : \mathbb{N} \to ((out!x \to P) \gg COPY)$$

EXERCISE 4.14 Is $F(Y) = a \to Y \,\Box\, b \to Y$ guarded? What are the traces of $N = F(N)$ ? Are there any other fixed points?

EXERCISE 4.15 Is $F(Y) = a \to Y \;{}_{\{a\}}\|_{\{a\}}\; Y$ guarded? What are the traces of $N = F(N)$? Are there any other fixed points?

EXERCISE 4.16 Is $F(Y) = ((x : \Sigma \to Y) \,\Box\, SKIP) \;|||\; Y$ guarded? What are the traces of $N = F(N)$? Are there any other fixed points?

EXERCISE 4.17 Rewrite $ATT \parallel CUST$ of Example 2.3 as a recursion which does not contain any parallel operators.

EXERCISE 4.18 Show that $(SPY \parallel MASTER) \setminus \{relay\}$ of Example 3.1 is equivalent to $listen?x \to RECORD(x)$, where $RECORD$ is defined by a mutual recursion

$$\begin{aligned} RECORD(x) \quad = \quad & listen?y \to log!x \to RECORD(y) \\ & \Box\; log!x \to listen?y \to RECORD(y) \end{aligned}$$

EXERCISE 4.19 If $OUT_x = out!x \to OUT_x$, then a variable can described recursively as $V_x = OUT_x \bigtriangleup in?y \to V_y$. It can also be defined using the equation $VAR_x = in?y \to VAR_y \,\Box\, out!x \to VAR_x$. Use the Unique Fixed Point law to show that these define the same process.

EXERCISE 4.20 Find a test which distinguishes $a \to b \to STOP$ from $a \to b \to c \to STOP$ under may testing.

EXERCISE 4.21 Find a test that distinguishes $M = (a \to b \to M \sqcap b \to a \to M)$ from $N = (a \to N \sqcap b \to N)$ under may testing.

EXERCISE 4.22 Is there a test that distinguishes $a \to STOP \sqcap a \to b \to STOP$ from $a \to b \to STOP$ under may testing?

EXERCISE 4.23 Is there a test that distinguishes $a \to b \to STOP$ from $a \to STOP \;|||\; b \to STOP$ under may testing?

EXERCISE 4.24 Is there a test that distinguishes $N = a \to N$ from $\bigsqcap_{n \in \mathbb{N}} A(n)$ under may testing, where $A(0) = STOP$ and $A(n+1) = a \to A(n)$ ?

EXERCISE 4.25 Can the processes $STOP$ and $SKIP$ be told apart without the presence of $SKIP_\omega$ in the language of tests?

EXERCISE 4.26 A congruence over a language is dependent on the constructs of the language. If new operators are introduced to the language, then the requirements for an equivalence to be a congruence alter, since the equivalence must also be respected by the new operators.

Suppose that a new operator $Int(P)$ is added to the language of CSP, with transitions as follows:

$$\frac{P \xrightarrow{a} P'}{Int(P) \xrightarrow{a} Int(P')} \qquad \frac{P \xrightarrow{\tau} P'}{Int(P) \xrightarrow{\tau} STOP}$$

$Int(P)$ can deadlock at any stage where internal transitions are possible for $P$.

1. Is may testing equivalence still a congruence in the language extended with this new operator?
2. What is the congruence associated with the equivalence relation $\equiv_{may}$ for the extended language?
3. Can a denotational definition be given for this operator in the traces model?

# 5 Specification and verification with traces

## 5.1 PROPERTY-ORIENTED SPECIFICATION

Systems are designed to satisfy particular requirements, and one of the uses of their semantics is to enable them to be judged against given specifications. In the traces model, a specification on a CSP process is given in terms of the traces it may engage in. It will characterize the traces that are acceptable and those that are not. A process meets the specification if all of its executions are acceptable: no matter which choices are taken, any execution of the process is guaranteed not to violate the specification.

If $S(tr)$ is a predicate on traces $tr$, then process $P$ *meets* or *satisfies* $S(tr)$ if $S(tr)$ holds of every trace $tr$ of $P$.

$$P \textbf{ sat } S(tr) \quad = \quad \forall\, tr \in \mathsf{traces}(P) \bullet S(tr)$$

The specification $S(tr)$ is said to be a *property-oriented specification,* since the required property is captured directly in terms of restrictions on traces. The predicate $S$ may be expressed in any notation, though in practice first order logic and elementary set and sequence notation are generally sufficient.

EXAMPLE 5.1 The requirement that there should not be more *down* events than *up* events is captured as the predicate

$$S(tr) = tr \downarrow down \leqslant tr \downarrow up$$

This states that the length of the trace restricted to the *down* event (i.e. the number of occurrences of *down*) should not exceed the length of the trace restricted to the *up* event. For a process to satisfy this specification, no possible trace should have more *down* events than *up* events: at no stage must the specification be violated.

The process $up \rightarrow up \rightarrow down \rightarrow STOP$ has as its trace set

$$\mathsf{traces}(up \rightarrow up \rightarrow down \rightarrow STOP) \quad = \quad \{ \ \langle\rangle, \langle up \rangle, \langle up, up \rangle, \langle up, up, down \rangle \ \}$$

and every trace $tr$ in that set meets $S(tr)$. It follows that

$$up \rightarrow up \rightarrow down \rightarrow STOP \quad \mathbf{sat} \quad tr \downarrow down \leqslant tr \downarrow up$$

□

If a process $P$ fails to meet a specification $S(tr)$, then this must be because it has some (finite) trace for which $S$ fails to hold: there is a point where the performance of a particular event takes the execution of $P$ outside the specification. To meet a trace specification, it is necessary to ensure that at every stage of an execution no violating events are performed. This kind of specification is called a *safety* specification, which requires that nothing 'bad' should ever happen, and it is precisely this kind of property that are expressed as specifications on traces.

Since every process has the empty trace $\langle\rangle$ as one of its traces, any specification $S$ which is satisfiable by some process, it must hold for the empty trace. If $S(\langle\rangle)$ does not hold, then this means that the specification is violated before the process even begins to execute, and so no process could meet it. The check $S(\langle\rangle)$ is a necessary condition for satisfiability.

Conversely the process *STOP* has the empty trace as its only trace. Hence any satisfiable specification will be met by *STOP*. It is certainly the safest process in the sense of trace specifications: it is always safe to do nothing, even if it is not very useful. In subsequent chapters other forms of specification will be discussed, notably *liveness* specifications, which *STOP* will not satisfy. But within the context of observations as traces, and considering only safety specifications, *STOP* is the process that meets all satisfiable specifications.

EXAMPLE 5.2 The requirement $PR(tr)$ that some event $a$ should always precede another event $b$ may be expressed a number of different ways:

$$\begin{aligned} PR_1(tr) &= (tr = tr_0 \frown \langle b \rangle \frown tr_1) \Rightarrow tr_0 \upharpoonright a \neq \langle\rangle \\ PR_2(tr) &= (tr = tr_0 \frown \langle b \rangle \frown tr_1) \Rightarrow tr \upharpoonright a \neq \langle\rangle \\ PR_3(tr) &= tr \upharpoonright a = \langle\rangle \Rightarrow tr \upharpoonright b = \langle\rangle \end{aligned}$$

The first predicate, $PR_1$, states that if a trace may be split around the event $b$, then the segment before the $b$ event should contain an $a$ event. The second predicate, $PR_2$, states that if a trace contains a $b$ event, then it should contain an $a$ event somewhere within it. $PR_2$ is not equivalent to $PR_1$ at the level of traces, since there are some traces (the simplest being $\langle b, a \rangle$) which meet $PR_2$ but not $PR_1$. However, they are equivalent at the level of *specifications* on processes, in the sense that $P$ **sat** $PR_1(tr) \Leftrightarrow P$ **sat** $PR_2(tr)$ for any process $P$.. This is a result of the downward closure property $T2$ again. If a trace $tr$ of $P$ meets $PR_2$ and $P$ **sat** $PR_2$, then any prefix of $tr$ should also meet $PR_2$. In particular, if $tr_0 \frown \langle b \rangle \frown tr_1$ is such a trace, then $tr_0 \frown \langle b \rangle$ is another such trace. Since $PR_2$ states that $a$ is in the latter trace, it must be in $tr_0$, the part of the trace that precedes $b$.

The predicate $PR_3(tr)$ is equivalent to $PR_2(tr)$, but expressed rather differently: it states that if $a$ has not yet occurred, then $b$ cannot yet have occurred. □

EXAMPLE 5.3 The typical requirement on a buffer or queue process is that messages are output in the same order as, and subsequent to, their input. This is captured by the specification that at any stage, the sequence of outputs that have been observed must be a prefix of the sequence of inputs.

$$B(tr) \quad = \quad tr \Downarrow out \leqslant tr \Downarrow in$$

This safety specification can be violated only by a process performing the wrong output at some point. □

## 5.2 VERIFICATION

The compositional nature of the trace semantics for CSP allows a compositional proof system to be provided for trace specifications. Specifications of processes may be deduced from specifications about their components, in a way which reflects the trace semantics of the CSP operators. The proof system is given as a set of proof rules for all of the CSP operators, in each case giving as conclusion a specification which holds of a composite process, from antecedents which describe specifications which hold for the component processes. The proof rules are sound with respect to the trace semantics given for the CSP operators, and complete relative to completeness of the specification language.

There are three rules whose validity is due to the nature of **sat** specification, and which therefore hold for all CSP processes.

The first is that any process meets the vacuous specification $true(tr)$, which holds for all traces $tr$.

$$\frac{}{P \textbf{ sat } true(tr)}$$

The second is that any specification may be weakened:

$$\frac{P \textbf{ sat } S(tr)}{P \textbf{ sat } T(tr)} \quad [\, \forall\, tr : TRACE \bullet S(tr) \Rightarrow T(tr) \,]$$

The final rule states that if $S(tr)$ and $T(tr)$ have been established separately, then the specification consisting of their conjunction is also established:

$$\frac{\begin{array}{l} P \ \mathbf{sat}\ S(tr) \\ P \ \mathbf{sat}\ T(tr) \end{array}}{P \ \mathbf{sat}\ (S \wedge T)(tr)}$$

This last rule allows separate proofs for $S(tr)$ and $T(tr)$, and then for the results to be combined. For instance, one proof may establish that $P$ **sat** $tr \downarrow a \leqslant tr \downarrow b$, and another proof may establish that $P$ **sat** $tr \downarrow b \leqslant tr \downarrow c$. This rule allows the deduction that

$$P \ \mathbf{sat}\ (tr \downarrow a \leqslant tr \downarrow b \wedge tr \downarrow b \leqslant tr \downarrow c)$$

and the previous rule, which allows weakening of a specification within a **sat** specification, is used to deduce

$$P \ \mathbf{sat}\ (tr \downarrow a \leqslant tr \downarrow c)$$

This chain of reasoning is independent of the nature of the process $P$, once the initial specifications for $P$ are established.

## *STOP*

There is only one trace of the process *STOP*: the empty trace. The strongest specification that is met by the process *STOP* is that $tr = \langle\rangle$. This is encapsulated in the rule

$$\frac{}{STOP \ \mathbf{sat}\ tr = \langle\rangle}$$

The rule has no antecedents, corresponding to the fact that *STOP* has no component processes.

The weakening rule given above can be used to show that any specification which is satisfiable by any process must be satisfiable by *STOP*. If $P$ **sat** $S(tr)$ (for some process $P$), then $S$ must hold of the empty trace: $S(\langle\rangle)$. This follows from the fact that $\langle\rangle$ is a trace of every process, and hence in particular of $P$. This means that $(tr = \langle\rangle) \Rightarrow S(tr)$, which may be used in an instance of the weakening rule:

$$\frac{STOP \ \mathbf{sat}\ (tr = \langle\rangle)}{STOP \ \mathbf{sat}\ S(tr)} \quad [\ \forall\, tr : TRACE \bullet (tr = \langle\rangle) \Rightarrow S(tr)\ ]$$

In fact, the side condition is equivalent to the assertion $S(\langle\rangle)$.

## Prefix

A trace of the process $a \rightarrow P$ is either empty, or begins with the event $a$ followed by a trace of $P$. If $P$ **sat** $S(tr)$ then the part of the trace after $a$ (that is: $tail(tr)$) must meet the specification $S$. This leads to the following proof rule:

$$\frac{P \ \textbf{sat}\ S(tr)}{a \rightarrow P \ \textbf{sat} \ \begin{array}{l} tr = \langle\rangle \\ \vee \\ head(tr) = a \wedge S(tail(tr)) \end{array}}$$

For instance, to show that $b \rightarrow a \rightarrow STOP$ **sat** $tr \downarrow a \leqslant tr \downarrow b$, the fact that $STOP$ **sat** $tr = \langle\rangle$ may be used. The proof rule for prefix allows the deduction that

$$a \rightarrow STOP \ \textbf{sat} \ (tr = \langle\rangle \vee (head(tr) = a \wedge (tail(tr) = \langle\rangle)))$$

which may be weakened to

$$a \rightarrow STOP \ \textbf{sat} \ tr \downarrow a \leqslant 1$$

This intermediate weakening allows a more concise specification to be carried through the rest of the proof.

A second application of the prefix rule then yields

$$b \rightarrow a \rightarrow STOP \ \textbf{sat} \ \begin{array}{l} tr = \langle\rangle \\ \vee \\ head(tr) = b \wedge tail(tr) \downarrow a \leqslant 1 \end{array}$$

and each disjunct in turn may be weakened to produce

$$b \rightarrow a \rightarrow STOP \ \textbf{sat} \ \begin{array}{l} tr \downarrow a \leqslant tr \downarrow b \\ \vee \\ tr \downarrow b \geqslant 1 \wedge tr \downarrow a \leqslant 1 \end{array}$$

which weakens finally to produce

$$b \rightarrow a \rightarrow STOP \ \textbf{sat} \ tr \downarrow a \leqslant tr \downarrow b$$

## Prefix Choice

The prefix choice operator generalizes the prefix operator: it contains a number of component processes, and the first event that is performed can be any one of the menu of events offered.

The antecedent to the rule assumes a family of specifications $S_a(tr)$, one for each of the components $P(a)$.

$$\frac{\forall\, a \in A \bullet P(a)\ \mathbf{sat}\ S_a(tr)}{x : A \rightarrow P(x)\ \mathbf{sat}\ \begin{array}{l} tr = \langle\rangle \\ \vee \\ \exists\, a \in A \bullet head(tr) = a \wedge S_a(tail(tr)) \end{array}}$$

### Output and Input

The output process $c!v \rightarrow P$ is simply a particular kind of prefix process, and the proof rule reflects this:

$$\frac{P\ \mathbf{sat}\ S(tr)}{c!v \rightarrow P\ \mathbf{sat}\ \begin{array}{l} tr = \langle\rangle \\ \vee \\ head(tr) = c.v \wedge S(tail(tr)) \end{array}}$$

Similarly, the input process $c?x : T \rightarrow P(x)$ is a special form of prefix choice, and so the proof rule is very similar:

$$\frac{\forall\, v \in T \bullet P(v)\ \mathbf{sat}\ S_v(tr)}{c?x : T \rightarrow P(x)\ \mathbf{sat}\ \begin{array}{l} tr = \langle\rangle \\ \vee \\ \exists\, v \in T \bullet head(tr) = c.v \wedge S_v(tail(tr)) \end{array}}$$

## *SKIP*

The process *SKIP* does nothing except terminate successfully. It has only two possible traces, one for the situation before it has terminated successfully, and the other for the situation after. These two traces are $\langle\rangle$ and $\langle\checkmark\rangle$, so the inference rule, which has no antecedents, is as follows:

$$\frac{}{SKIP\ \mathbf{sat}\ tr = \langle\rangle \vee tr = \langle\checkmark\rangle}$$

## *RUN*

The process *RUN* is able to engage in any trace. If it is able to meet a specification, then that specification must allow all possible traces. This will therefore be an extremely weak specification, since it places no restrictions on the traces that are acceptable. This specification is equivalent to *true*:

$$\frac{}{\mathit{RUN}\ \mathbf{sat}\ \mathit{true}(tr)}$$

In fact this rule is superfluous, since the conclusion may already be derived from the first **sat** rule given above, concerning the vacuous specification *true*. However, it is given here for the process *RUN* in order to cover that process explicitly.

## External Choice

The process $P_1 \ \Box\ P_2$ behaves either as $P_1$ or as $P_2$. If $P_1$ **sat** $S(tr)$ and $P_2$ **sat** $T(tr)$ then the choice process $P_1 \ \Box\ P_2$ satisfies the disjunction of these two specifications:

$$\frac{\begin{array}{l} P_1\ \mathbf{sat}\ S(tr) \\ P_2\ \mathbf{sat}\ T(tr) \end{array}}{P_1 \ \Box\ P_2\ \mathbf{sat}\ S(tr) \lor T(tr)}$$

Any trace of $P_1 \ \Box\ P_2$ will meet either $S(tr)$ if it arises from $P_1$, or else $T(tr)$ if it is generated from $P_2$.

The indexed external choice $\Box_{i \in I} P_i$ behaves in a similar way, meeting the disjunction of the specifications met by its components:

$$\frac{\forall\, i \in I \bullet P_i\ \mathbf{sat}\ S_i(tr)}{\Box_{i \in I} P_i\ \mathbf{sat}\ \exists\, i \in I \bullet S_i(tr)}$$

Any trace of the indexed choice must meet at least one of the component specifications.

## Internal choice

The internal choice operator has the same trace semantics as the external choice operator, so the inference rules will also be the same:

$$\frac{\begin{array}{l} P_1\ \mathbf{sat}\ S(tr) \\ P_2\ \mathbf{sat}\ T(tr) \end{array}}{P_1 \sqcap P_2\ \mathbf{sat}\ S(tr) \lor T(tr)}$$

$$\frac{\forall\, i \in J \bullet P_i\ \mathbf{sat}\ S_i(tr)}{\bigsqcap_{i \in J} P_i\ \mathbf{sat}\ \exists\, i \in J \bullet S_i(tr)}$$

## Parallel composition

A trace $tr$ of the process $P_1 \; {}_A\|_B \; P_2$ is comprised of a contribution from $P_1$ and a contribution from $P_2$, contained within the alphabets $A^{\checkmark}$ and $B^{\checkmark}$ respectively. In fact, the projection of the trace onto $A^{\checkmark}$—$tr \upharpoonright A^{\checkmark}$—is a trace of $P_1$, and the projection $tr \upharpoonright B^{\checkmark}$ is a trace of $P_2$. If $P_1$ **sat** $S(tr)$, then this means that $S(tr \upharpoonright A^{\checkmark})$ must hold. Similarly, if $P_2$ **sat** $T(tr)$, then $T(tr \upharpoonright B^{\checkmark})$ must hold. Finally, only events in $A^{\checkmark}$ or $B^{\checkmark}$ are possible for the parallel combination, so it follows that $\sigma(tr) \subseteq (A \cup B)^{\checkmark}$. This leads to the following proof rule:

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S(tr) \\ P_2 \textbf{ sat } T(tr) \end{array}}{P_1 \; {}_A\|_B \; P_2 \textbf{ sat } S(tr \upharpoonright A^{\checkmark}) \wedge T(tr \upharpoonright B^{\checkmark}) \wedge \sigma(tr) \subseteq (A \cup B)^{\checkmark}}$$

This rule demonstrates the way in which parallel composition corresponds to conjunction: the constraints $S$ and $T$ both hold, on their respective alphabets.

For instance, the following recursive processes meet specifications as follows:

$$\begin{array}{lll} P_1 = b \to a \to P_1 & \textbf{sat} & S(tr) = (tr \downarrow a \leqslant tr \downarrow b) \\ P_2 = c \to b \to P_2 & \textbf{sat} & T(tr) = (tr \downarrow b \leqslant tr \downarrow c) \end{array}$$

Then the combination $P_1 \; {}_{\{a,b\}}\|_{\{b,c\}} \; P_2$ meets the specification

$$S(tr \upharpoonright \{a,b\}^{\checkmark}) \wedge T(tr \upharpoonright \{b,c\}^{\checkmark}) \wedge \sigma(tr) \subseteq \{a,b,c\}^{\checkmark}$$

Observe that $S(tr)$ is concerned only with the occurrence of the events $a$ and $b$ in the trace $tr$, and its truth depends only on the trace restricted to those two events. Thus $S(tr) \Leftrightarrow S(tr \upharpoonright \{a,b\}^{\checkmark})$, and similarly $T(tr) \Leftrightarrow T(tr \upharpoonright \{b,c\}^{\checkmark})$. This often turns out to be the case when processes are combined in parallel, since the interface set for a process generally contains all of the events that it can perform, and specifications on such processes are usually concerned only with constraints on their performable events.

The specification is then equivalent to

$$S(tr) \wedge T(tr) \wedge \sigma(tr) \subseteq \{a,b,c\}^{\checkmark}$$

which reduces to

$$tr \downarrow a \leqslant tr \downarrow b \leqslant tr \downarrow c \wedge \sigma(tr) \subseteq \{a,b,c\}^{\checkmark}$$

The constraints imposed by $P_1$ and $P_2$ separately ensure that $c$ must occur at least as often as $a$ does.

The rule for the indexed parallel operator follows a similar pattern. Each component $P_i$ with interface $A_i$ imposes its own constraint $S_i(tr)$ on the projection of the overall trace onto the alphabet $A_i$.

$$\frac{\forall\, i \in I \bullet P_i \textbf{ sat } S_i(tr)}{\big\|_{A_i}^{i \in I} P_i \textbf{ sat } (\forall\, i \in I \bullet S_i(tr \upharpoonright A_i^{\checkmark})) \wedge \sigma(tr) \subseteq (\bigcup_{i \in I} A_i)^{\checkmark}}$$

## Interleaving

An interleaved combination $P_1 \;|||\; P_2$ performs traces $tr$ which consist of a trace $tr_1$ of $P_1$ interleaved with a trace $tr_2$ of $P_2$. This leads to the following inference rule:

$$\frac{\begin{array}{l} P_1 \;\mathbf{sat}\; S(tr) \\ P_2 \;\mathbf{sat}\; T(tr) \end{array}}{P_1 \;|||\; P_2 \;\mathbf{sat}\; \exists\, tr_1, tr_2 \bullet (S(tr_1) \wedge T(tr_2) \wedge tr \;\textsf{interleaves}\; tr_1, tr_2)}$$

The resulting specification on $tr$ met by $P_1 \;|||\; P_2$ states that $tr$ interleaves two traces meeting $S$ and $T$ respectively.

In practice the nature of $S$ and $T$ often allow this specification to be weakened to a more direct requirement $R(tr)$, by establishing the following:

$$\forall\, tr_1, tr_2 \bullet ((S(tr_1) \wedge T(tr_2) \wedge tr \;\textsf{interleaves}\; tr_1, tr_2) \Rightarrow R(tr))$$

If $R(tr)$ holds whenever $tr$ interleaves two traces meeting $S$ and $T$, then it must hold for the particular traces $tr_1$ and $tr_2$ whose existence is asserted in the proof rule above. In such cases, it can be deduced that $P_1 \;|||\; P_2 \;\mathbf{sat}\; R(tr)$.

This may be captured in an alternative proof rule:

$$\frac{\begin{array}{l} P_1 \;\mathbf{sat}\; S(tr) \\ P_2 \;\mathbf{sat}\; T(tr) \\ \forall\, tr_1, tr_2, tr : TRACE \bullet \left( \begin{array}{l} S(tr_1) \wedge T(tr_2) \\ \wedge\; tr \;\textsf{interleaves}\; tr_1, tr_2 \end{array} \right) \Rightarrow R(tr) \end{array}}{P_1 \;|||\; P_2 \;\mathbf{sat}\; R(tr)}$$

EXAMPLE 5.4 Consider that $S(tr)$ is a specification which states that $a$ must appear in the trace before $b$ does. This may be captured as follows:

$$S(tr) \quad = \quad tr \upharpoonright a = \langle\rangle \Rightarrow tr \upharpoonright b = \langle\rangle$$

This states that if no $a$ has occurred, then no $b$ can have occurred. This is equivalent on processes to stating that any $b$ event must be preceded by an $a$ event because the trace sets of processes are prefix closed.

If both $P_1$ and $P_2$ meet specification $S(tr)$, then any $b$ performed by one of the components of $P_1 \;|||\; P_2$ must have been preceded by an $a$ event performed by the same component; so it appears that $P_1 \;|||\; P_2 \;\mathbf{sat}\; S(tr)$. In order to establish this using the second proof rule for interleaving, it is necessary to check the third antecedent:

$$\left.\begin{array}{l} (tr_1 \upharpoonright a = \langle\rangle \Rightarrow tr_1 \upharpoonright b = \langle\rangle) \wedge \\ (tr_2 \upharpoonright a = \langle\rangle \Rightarrow tr_2 \upharpoonright b = \langle\rangle) \wedge \\ tr \;\textsf{interleaves}\; tr_1, tr_2 \end{array}\right\} \Rightarrow (tr \upharpoonright a = \langle\rangle \Rightarrow tr \upharpoonright b = \langle\rangle)$$

This result is reasonably intuitive, but to establish it formally requires an inductive proof on *tr* because of the inductive definition of interleaves. $\Box$

Interleaved combinations $P_1 \; ||| \; P_2$ are often used to make explicit the fact that the component processes have no direct interaction with one another, in cases where the alphabets of $P_1$ and $P_2$ (apart from termination) are disjoint. In such cases, it is sometimes more convenient for proof if the combination is rewritten to the form $P_1 \;{}_{\alpha(P_1)}\|_{\alpha(P_2)}\; P_2$ as follows:

$$P_1 \; ||| \; P_2 \quad = \quad P_1 \;{}_{\alpha(P_1)}\|_{\alpha(P_2)}\; P_2 \qquad \text{if } \alpha(P_1) \cap \alpha(P_2) = \{\}$$

This form is supported by the more straightforward proof rule for alphabetized parallel with its more direct relationship between the behaviour of the whole process and traces of its components.

## Interface parallel

The approach to this operator is similar to the approach taken to pure interleaving. A trace *tr* of $P_1 \underset{A}{\|} P_2$ must arise from two traces $tr_1$ and $tr_2$ of $P_1$ and $P_2$ respectively, where $tr \;\mathsf{synch}_A\; tr_1, tr_2$. This results in the following inference rule:

$$\frac{\begin{array}{l} P_1 \;\mathbf{sat}\; S(tr) \\ P_2 \;\mathbf{sat}\; T(tr) \end{array}}{P_1 \underset{A}{\|} P_2 \;\mathbf{sat}\; \exists\, tr_1, tr_2 \bullet (S(tr_1) \wedge T(tr_2) \wedge tr \;\mathsf{synch}_A\; tr_1, tr_2)}$$

As in the case of the interleaving operator it may be possible in particular cases of $S$ and $T$ to show that

$$\forall\, tr_1, tr_2, tr \bullet (S(tr_1) \wedge T(tr_2) \wedge tr \;\mathsf{synch}_A\; tr_1, tr_2) \Rightarrow R(tr)$$

which would allow the deduction of

$$P_1 \underset{A}{\|} P_2 \;\mathbf{sat}\; R(tr)$$

For example, if both $P_1$ and $P_2$ meet the specification $S(tr) = (tr \downarrow B \leqslant tr \downarrow A)$ where $B \cap A = \{\}$, then $P_1 \underset{A}{\|} P_2$ should meet the specification $R(tr) = (tr \downarrow B \leqslant 2 * tr \downarrow A)$, since events from $B$ are performed independently by $P_1$ and $P_2$, but events from $A$ are performed together. Indeed it is straightforward to check that

$$\forall\, tr_1, tr_2, tr \bullet (S(tr_1) \wedge S(tr_2) \wedge tr \;\mathsf{synch}_A\; tr_1, tr_2) \Rightarrow R(tr)$$

from which the required result follows.

EXAMPLE 5.5 Another example is provided by the special case $P_1 \underset{\{\}}{\|} P_2$ which interleaves its components on all events except termination. Let $term(tr) = \checkmark \in \sigma(tr)$ denote that the trace corresponds to a terminating execution. If $P_1$ **sat** $(term(tr) \Rightarrow S(tr))$ and $P_2$ **sat** $(term(tr) \Rightarrow T(tr))$, then $P_1 \underset{\{\}}{\|} P_2$ on termination will meet the appropriate combination of $S$ and $T$. For example, let $\mathsf{sum}(seq)$ be the sum of the elements of a sequence $seq \in \mathbb{Z}^*$ . Then for instance the specification $S_n(tr) = (term(tr) \Rightarrow \mathsf{sum}(tr \Downarrow C) = n)$ states that at the time of termination, the sum of all the values passed along all the channels in the set $C$ is $n$. If $P_1$ **sat** $S_n(tr)$, and $P_2$ **sat** $S_m(tr)$, then $P_1 \underset{\{\}}{\|} P_2$ **sat** $S_{m+n}(tr)$: by the time of termination, the sum of the values passed by the parallel combination will be $m + n$. □

## Hiding

A trace of the process $P \setminus A$ arises from a trace of $P$ simply by removing all of the events in $A$ from the trace. Hence for any trace of $P \setminus A$ there is a corresponding trace of $P$. The inference rule thus takes the following form:

$$\frac{P \ \mathbf{sat}\ S(tr)}{P \setminus A \ \mathbf{sat}\ \exists\, tr_1 \bullet tr_1 \setminus A = tr \wedge S(tr_1)}$$

For instance, the process $P$ might be given by $P = a \rightarrow b \rightarrow c \rightarrow P$, and the property that has been proven for $P$ is that $tr \downarrow c \leqslant tr \downarrow b \leqslant tr \downarrow a$. The rule allows the deduction that

$$P \setminus \{b\} \quad \mathbf{sat} \quad \exists\, tr_1 \bullet tr_1 \setminus \{b\} = tr \wedge tr_1 \downarrow c \leqslant tr_1 \downarrow b \leqslant tr_1 \downarrow a$$

and this specification (observe it is on $tr$) is equivalent to $tr \downarrow c \leqslant tr \downarrow a$, since it holds for $tr$ precisely when this latter specification does.

In fact, the inference rule simplifies in the case where the specification $S(tr)$ is *independent* of the set $A$ being hidden, in the sense that it holds independently of the presence or absence of events from $A$ in the trace. A specification is $A$-independent if

$$\forall\, tr \bullet S(tr) \Leftrightarrow S(tr \setminus A)$$

For such a specification, the predicate $\exists\, tr_1 \bullet (S(tr_1) \wedge tr_1 \setminus A = tr)$ is equivalent to $S(tr)$, since if there is some such $tr_1$ then $S(tr_1 \setminus A)$—that is $S(tr)$— holds. And if there is no such $tr_1$, then $S(tr)$ does not hold, since $tr$ is a candidate $tr_1$. The conclusion of the rule simplifies, and the resulting rule is

$$\frac{P \ \mathbf{sat}\ S(tr)}{P \setminus A \ \mathbf{sat}\ S(tr)} \quad [\, S(tr) \text{ is } A\text{-independent}\,]$$

Generally, predicates which are concerned only with certain events in the trace are likely to be good candidates for independence from other events. For example, the specification $tr \downarrow c \leqslant tr \downarrow a$ is concerned only with the projection of the trace onto the events $a$ and $c$, and so it is $A$-independent for any set $A$ which does not contain $a$ or $c$. Hence one instantiation of the revised rule for the recursive process $P = a \to b \to c \to P$ would be

$$\frac{P \ \mathbf{sat}\ tr \downarrow c \leqslant tr \downarrow a}{P \setminus \{b\} \ \mathbf{sat}\ tr \downarrow c \leqslant tr \downarrow a} \quad [\ \text{`}tr \downarrow c \leqslant tr \downarrow a\text{' is } \{b\}\text{-independent}\,]$$

If the set $A$ does not contain the channel $c$, (and recall it cannot contain $\checkmark$), then a specification stating that a particular result will be provided on channel $c$ before termination is $A$-independent, and so it is maintained by hiding the set $A$. The relevant instantiation of the rule is as follows:

$$\frac{P \ \mathbf{sat}\ (term(tr) \Rightarrow tr \Downarrow c = \langle 42 \rangle)}{P \setminus A \ \mathbf{sat}\ (term(tr) \Rightarrow tr \Downarrow c = \langle 42 \rangle)}$$

since the predicate '$term(tr) \Rightarrow tr \Downarrow c = \langle 42 \rangle$' is $A$-independent.

Observe that specifications might not be $A$-independent even if they do not mention events from $A$ explicitly. The specification $term(tr) \Rightarrow \#tr > 5$, for example, states that the process must do more than 5 events before terminating. This is not $A$-independent (for any $A \neq \{\}$), since all events are counted towards the length of the trace; this specification is concerned with all events in $\Sigma$.

## Renaming

A trace $tr$ of a renamed process $f(P)$ will be a renamed trace $f(tr_1)$ for some $tr_1$ of $P$. The inference rule for translating specifications through a forward renaming is then as follows:

$$\frac{P \ \mathbf{sat}\ S(tr)}{f(P) \ \mathbf{sat}\ \exists\, tr_1 \bullet S(tr_1) \wedge f(tr_1) = tr}$$

For particular specifications $S(tr)$ it is possible to translate $S$ through $f$ to a specification $R$. This will be valid provided $R(tr)$ can be shown to translate $S$ correctly:

$$\forall\, tr \bullet (S(tr) \Rightarrow R(f(tr)))$$

For example, consider the restriction of the trace to a particular set. If $P$ **sat** $tr \downarrow A \leqslant n$, and $A = f^{-1}(B)$ for some set $B$ (in the sense that $A = \{a \mid f(a) \in B\}$), then it might be expected that $f(P)$ **sat** $tr \downarrow B \leqslant n$. The result that

$$\forall\, tr \bullet (tr \downarrow A \leqslant n \Rightarrow tr \downarrow B \leqslant n)$$

allows this conclusion to be deduced.

In the case where $f$ is a 1–1 function, its inverse $f^{-1}$ is also a (partial) function, and there is only one possibility for the trace $tr_1$ which maps under $f$ to $tr$, namely $f^{-1}(tr)$. In this case the inference rule simplifies as follows:

$$\frac{P \textbf{ sat } S(tr)}{f(P) \textbf{ sat } S(f^{-1}(tr))} \quad [\, f \text{ injective} \,]$$

The specification $S$ may often itself be transformed by $f$ when simplifying $S(f^{-1}(tr))$. For example, consider a process $B$ meeting the specification of a buffer:

$$B \textbf{ sat } tr \Downarrow out \leqslant tr \Downarrow in$$

If the channel names are renamed to *left* and *right* by the application of an injective renaming function $f$ defined by

$$\begin{array}{rcl} f(in.m) & = & left.m \\ f(out.m) & = & right.m \\ f(left.m) & = & in.m \\ f(right.m) & = & out.m \end{array}$$

then $f(B)$ **sat** $f^{-1}(tr) \Downarrow out \leqslant f^{-1}(tr) \Downarrow in$. The sequence of messages $f^{-1}(tr) \Downarrow out$ is the same as the sequence $tr \Downarrow f(out)$, that is $tr \Downarrow right$. Similarly, $f^{-1}(tr) \Downarrow in$ is equivalent to $tr \Downarrow left$, and so the result is that

$$f(B) \textbf{ sat } tr \Downarrow right \leqslant tr \Downarrow left$$

The inverse renaming operator is more straightforward. If $tr$ is a trace of $f^{-1}(P)$, then $f(tr)$ is a trace of $P$, and so it must satisfy whatever specification $P$ is known to satisfy. The inference rule is as follows:

$$\frac{P \textbf{ sat } S(tr)}{f^{-1}(P) \textbf{ sat } S(f(tr))}$$

For example, let $P$ be the process $P = a \to d \to P$. Then

$$P \quad \textbf{sat} \quad tr \downarrow d \leqslant tr \downarrow a$$

so for any function $f$ it follows that

$$f^{-1}(P) \quad \textbf{sat} \quad f(tr) \downarrow d \leqslant f(tr) \downarrow a$$

The process and the specification may be independently simplified.

If for instance $f$ is the function defined by

$$\begin{aligned} f(a) = f(b) = f(c) &= a \\ f(d) = f(e) &= d \end{aligned}$$

the process $f^{-1}(P)$ reduces to

$$\begin{aligned} f^{-1}(P) \;=\; & a \to (d \to f^{-1}(P) \;\Box\; e \to f^{-1}(P)) \\ & \Box\; b \to (d \to f^{-1}(P) \;\Box\; e \to f^{-1}(P)) \\ & \Box\; c \to (d \to f^{-1}(P) \;\Box\; e \to f^{-1}(P)) \end{aligned}$$

which is the same as the recursively defined process

$$\begin{aligned} P' \;=\; & a \to (d \to P' \;\Box\; e \to P') \\ & \Box\; b \to (d \to P' \;\Box\; e \to P') \\ & \Box\; c \to (d \to P' \;\Box\; e \to P') \end{aligned}$$

by the unique fixed point law $\mathsf{UFP}_T$.

Furthermore, the specification

$$f(tr) \downarrow d \leqslant f(tr) \downarrow a$$

reduces to

$$tr \downarrow f^{-1}(\{d\}) \leqslant tr \downarrow f^{-1}(\{a\})$$

which expands to

$$tr \downarrow \{d, e\} \leqslant tr \downarrow \{a, b, c\}$$

and so it is finally established that

$$P' \quad \textbf{sat} \quad tr \downarrow \{d, e\} \leqslant tr \downarrow \{a, b, c\}$$

### Sequential composition

The process $P_1 ;\ P_2$ behaves entirely as $P_1$ until $P_1$ terminates, after which it behaves as $P_2$. Any given trace of $P_1 ;\ P_2$ admits one of two possibilities: either it is a trace of $P_1$ which has not yet reached termination, or else it is a trace of $P_1$ followed by a trace of $P_2$. The proof rule reflects this dichotomy:

$$\frac{\begin{array}{l} P_1 \ \textbf{sat}\ S(tr) \\ P_2 \ \textbf{sat}\ T(tr) \end{array}}{\begin{array}{ll} P_1 ;\ P_2 \ \textbf{sat} & \neg\ term(tr) \wedge S(tr) \\ & \vee \\ & \exists\, tr_1, tr_2 \bullet tr = tr_1 \frown tr_2 \wedge S(tr_1 \frown \langle \checkmark \rangle) \wedge T(tr_2) \end{array}}$$

The first case covers those traces from $P_1$ that have not yet terminated. The second case is concerned with those traces corresponding to executions that have passed control from $P_1$ to $P_2$ at some point: in this case, $tr_1 \frown \langle \checkmark \rangle$ is the part of the trace from $P_1$ up to termination, and $tr_2$ is the contribution from $P_2$, after control has been passed. Observe that the $\checkmark$ from $P_1$'s termination does not appear in $tr$, reflecting the trace semantics of sequential composition.

A degenerate case concerns the situation where the specification $S(tr)$ for $P_1$ does not allow for termination: $S(tr) \Rightarrow \neg\ term(tr)$. In this case the first disjunct above reduces to $S(tr)$, and the second reduces to $false(tr)$, since $S(tr_1 \frown \langle \checkmark \rangle)$ cannot hold. In other words, in the situation where $P_1$ cannot terminate, the result is that $P_1 ;\ P_2$ **sat** $S(tr)$, corresponding to the fact that all executions will be entirely due to $P_1$.

### Interrupt

A trace of the interrupt process $P_1 \triangle P_2$ is either a trace of $P_1$, or else a non-terminated trace of $P_1$ followed by a trace of $P_2$. The inference rule is as follows:

$$\frac{\begin{array}{l} P_1 \ \textbf{sat}\ S(tr) \\ P_2 \ \textbf{sat}\ T(tr) \end{array}}{\begin{array}{ll} P_1 \triangle P_2 \ \textbf{sat} & S(tr) \\ & \vee \\ & \exists\, tr_1, tr_2 \bullet tr = tr_1 \frown tr_2 \wedge \neg\ term(tr_1) \wedge S(tr_1) \wedge T(tr_2) \end{array}}$$

## 5.3 RECURSION INDUCTION

If process $N$ is recursively defined by the equation $N = P$ or equivalently by $N = F(N)$ (where $F(Y) = P[Y/N]$), then a rule which is sufficient to establish that $N$ **sat** $S(tr)$ is the following:

$$\frac{\forall\, Y \bullet (Y \ \textbf{sat}\ S(tr) \Rightarrow F(Y) \ \textbf{sat}\ S(tr))}{N \ \textbf{sat}\ S(tr)} \quad [\ S(\langle\rangle)\ ]$$

This rule is sound because it provides all the ingredients for establishing by induction that $N$ **sat** $S(tr)$. The traces of $N$ are those of $\bigcup_i \mathsf{traces}(F^i(STOP))$, all the finite unwindings of $F(Y)$ starting from the process $STOP$. The inductive hypothesis is that $F^i(STOP)$ **sat** $S(tr)$. The side condition $S(\langle\rangle)$ provides the base case, since it is equivalent to the statement $STOP$ **sat** $S(tr)$, which is the same as $F^0(STOP)$ **sat** $S(tr)$. The antecedent of the rule provides the basis for the inductive step: that if an arbitrary process $Y$ meets the specification $S(tr)$, then so does $F(Y)$. Hence from the fact that $F^i(STOP)$ **sat** $S(tr)$ it follows that $F(F^i(STOP))$ **sat** $S(tr)$: that is, $F^{i+1}(STOP)$ **sat** $S(tr)$. The conclusion that $\forall\, i \bullet F^i(STOP)$ **sat** $S(tr)$ follows by induction. and so $\forall\, tr \in \bigcup_i \mathsf{traces}(F^i(STOP)) \bullet S(tr)$, which means that $N$ **sat** $S(tr)$.

Establishing the antecedent to the rule will depend on the CSP operators used in the recursive function $F(Y)$. Typically, the rules appropriate to these operators would be used.

EXAMPLE 5.6 The recursively defined process $N = a \to b \to N$ alternates on performance of the events $a$ and $b$. One specification which it meets is that the number of $b$ events never exceeds the number of $a$ events performed. This is expressed in the specification $S(tr) = tr \downarrow b \leqslant tr \downarrow a$

To establish the antecedent, it is necessary to show that $Y$ **sat** $S(tr) \Rightarrow a \to b \to Y$ **sat** $S(tr)$. Assuming that $Y$ **sat** $S(tr)$, it follows from an application of the rule for prefix that

$$b \to Y \ \mathbf{sat}\ tr = \langle\rangle \vee (tr = \langle b\rangle \frown tr' \wedge S(tr'))$$

and so it follows from another application of that rule that

$$\begin{aligned} a \to b \to Y \ \mathbf{sat}\ tr = \langle\rangle \vee \ & (tr = \langle a\rangle \frown tr' \\ & \wedge (tr' = \langle\rangle \vee (tr' = \langle b\rangle \frown tr'' \wedge S(tr'')))) \end{aligned}$$

This simplifies to

$$a \to b \to Y \ \mathbf{sat}\ tr = \langle\rangle \vee tr = \langle a\rangle \vee (tr = \langle a, b\rangle \frown tr'' \wedge tr'' \downarrow b \leqslant tr'' \downarrow a)$$

which implies that

$$a \to b \to Y \ \mathbf{sat}\ S(tr)$$

as required. It is also necessary to check the side condition $S(\langle\rangle)$, which in this case is trivial. Hence the recursion induction rule allows the conclusion that $N$ **sat** $S(tr)$. □

EXAMPLE 5.7 A definition of the mail forwarder process $NODE$ of Example 2.17 is

$$NODE \quad = \quad in?x : M \to (NODE \ ||| \ out!x \to STOP)$$

This process appears to satisfy the specification that any output $v$ must previously have been input. This is expressed as the following predicate on traces:

$$\forall\, v \bullet out.v \textbf{ in } tr \Rightarrow in.v \textbf{ in } tr$$

This can be established for *NODE* by showing that the body of the definition preserves the specification: that if $Y$ satisfies it, then so does $F(Y) = in?x : M \rightarrow (Y \mid\mid\mid out!x \rightarrow STOP)$. This is achieved by using the rules for interleaving and input. Assuming that $Y$ **sat** $\forall\, v \bullet out.v \textbf{ in } tr \Rightarrow in.v \textbf{ in } tr$, it follows that, for any given $w$,

$$Y \mid\mid\mid out.w \rightarrow STOP \quad \textbf{sat} \quad \forall\, v \bullet out.v \textbf{ in } tr \Rightarrow (in.v \textbf{ in } tr \vee v = w)$$

The rule for input yields that

$$\begin{array}{ll} \multicolumn{2}{l}{in?x : M \rightarrow (Y \mid\mid\mid out.x \rightarrow STOP)} \\ \quad \textbf{sat} & tr = \langle\rangle \\ & \vee \\ & head(tr) = in.w \wedge \\ & (\forall\, v \bullet out.v. \textbf{ in } tail(tr) \Rightarrow (in.v \textbf{ in } tail(tr) \vee v = w)) \end{array}$$

and this specification may be weakened to give the result

$$in?x : M \rightarrow (Y \mid\mid\mid out.x \rightarrow STOP) \quad \textbf{sat} \quad \forall\, v \bullet (out.v \textbf{ in } tr \Rightarrow in.v \textbf{ in } tr)$$

It follows that *NODE* meets this specification. □

EXAMPLE 5.8 It may happen that the specification $S(tr)$ itself is not preserved by recursive calls, even though it happens to hold of the recursively defined process. For example, the specification $S(tr) = tr \upharpoonright a = \langle\rangle \vee tr \upharpoonright b = \langle\rangle$ states that the trace $tr$ cannot contain occurrences of both event $a$ and event $b$. This holds for the process $N = a \rightarrow N$, but it is not in general preserved by the function $F(Y) = a \rightarrow Y$ defining the recursion: for instance, $b \rightarrow STOP$ **sat** $S(tr)$, but $F(b \rightarrow STOP)$ does not satisfy $S(tr)$.

One approach in such cases is to find a stronger property $T(tr)$ which is preserved by recursive calls, for which $T(tr) \Rightarrow S(tr)$.

In the case above, a suitable $T(tr)$ would be $tr \upharpoonright b = \langle\rangle$. This is preserved by the body of the recursive definition $F(Y)$, and it also implies $S(tr)$. □

## Mutual recursion

A mutual recursion is treated in an entirely similar way to the single case, though some extra care must be taken to handle the indices of the family of defined processes.

A family of processes $N(i)$ indexed by a set $I$ is defined by an associated family of equations $N(i) = F(i)(\underline{N})$. The entire definition is described at a stroke as $\underline{N} = \underline{F}(\underline{N})$.

In terms of specification, a family of processes may be associated with a family of specifications $S(i)(tr)$, also indexed by $I$. In this case, $\underline{N}$ **sat** $\underline{S}(tr)$ means that each process satisfies the associated specification: $N(i)$ **sat** $S(i)(tr)$, for each index $i$.

Within this framework, the inference rule for mutual recursion is similar to that for single recursion. If $\underline{N} = \underline{F}(\underline{N})$, then

$$\frac{\forall\, \underline{Y} \bullet \underline{Y}\ \mathbf{sat}\ \underline{S}(tr) \Rightarrow \underline{F}(\underline{Y})\ \mathbf{sat}\ \underline{S}(tr)}{\underline{N}\ \mathbf{sat}\ \underline{S}(tr)} \quad [\, \forall\, i \in I \bullet S(i)(\langle\rangle)\, ]$$

The antecedent of this rule is equivalent to the requirement for arbitrary $j \in I$ that $F(j)(\underline{Y})$ **sat** $S(j)(tr)$, under the assumption that $Y(i)$ **sat** $S(i)(tr)$ for every $i \in I$.

EXAMPLE 5.9 Two processes defined through a mutual recursion are *LIGHT* and *ON* of Example 1.17:

$$\begin{aligned} LIGHT &= on \rightarrow ON \\ ON &= off \rightarrow LIGHT \end{aligned}$$

These may be shown to meet the respective pair of specifications

$$\begin{aligned} S_1(tr) &= tr \downarrow off \leqslant tr \downarrow on \leqslant tr \downarrow off + 1 \\ S_2(tr) &= tr \downarrow on \leqslant tr \downarrow off \leqslant tr \downarrow on + 1 \end{aligned}$$

The proof rule for mutual recursion induction requires as its antecedent that the pair of functions preserve the pair of specifications. This means that

$$\begin{aligned} &\forall\, Y \bullet (Y\ \mathbf{sat}\ S_2(tr) &\Rightarrow\quad & on \rightarrow Y\ \mathbf{sat}\ S_1(tr)) \\ &\forall\, Y \bullet (Y\ \mathbf{sat}\ S_1(tr) &\Rightarrow\quad & off \rightarrow Y\ \mathbf{sat}\ S_2(tr)) \end{aligned}$$

These may be established by an application of the proof rule for prefix, and so the conclusion *LIGHT* **sat** $S_1(tr)$ and *ON* **sat** $S_2(tr)$ follows. □

EXAMPLE 5.10 The example of the family of counter processes defined in terms of each other, indexed by ℕ, was given in Example 4.30.

$$\begin{aligned} COUNT(0) &= increment \rightarrow COUNT(1) \\ COUNT(i) &= \quad increment \rightarrow COUNT(i+1) \qquad \text{if } i > 0 \\ &\qquad \mid decrement \rightarrow COUNT(i-1) \end{aligned}$$

The intention is that $COUNT(0)$ can perform no more *decrement* events that *increment* events. In general, the index of any particular $COUNT(i)$ process reached during an execution of $COUNT(0)$ records the number by which occurrences of *increment* exceed those of *decrement*.

This means that to be consistent with the requirement on $COUNT(0)$, each process $COUNT(i)$ can perform up to $i$ more *decrement*s than *increment*s. The corresponding specifications are

$$S(i)(tr) \quad = \quad tr \downarrow decrement \leqslant (i + tr \downarrow increment)$$

To prove that each $COUNT(i)$ **sat** $S(i)(tr)$, it is sufficient to show that this claim is preserved when each process is replaced by its definition. There are essentially two cases to consider, corresponding to the two possibilities $i = 0$ and $i > 0$.

In the case $i = 0$, the definition of $COUNT(0)$ is $increment \rightarrow COUNT(1)$. Under the assumption that $COUNT(1)$ **sat** $S(1)(tr)$, an application of the inference rule for prefix yields that

$$\begin{array}{ll} \multicolumn{2}{l}{increment \rightarrow COUNT(1)} \\ \quad \mathbf{sat} & tr = \langle\rangle \\ & \vee \\ & head(tr) = increment \wedge \\ & tail(tr) \downarrow decrement \leqslant (1 + tail(tr) \downarrow increment) \end{array}$$

The specification can be weakened to obtain

$$increment \rightarrow COUNT(1) \quad \mathbf{sat} \quad S(0)$$

The other case to consider is $i > 0$. In this case, the relevant assumptions are $COUNT(i+1)$ **sat** $S(i+1)(tr)$, and $COUNT(i-1)$ **sat** $S(i-1)(tr)$, since it is these process names that appear in the definition of $COUNT(i)$. The inference rule for prefix choice yields that

$$\begin{array}{ll} \multicolumn{2}{l}{increment \rightarrow COUNT(i+1) \mid decrement \rightarrow COUNT(i-1)} \\ \quad \mathbf{sat} & tr = \langle\rangle \\ & \vee\ head(tr) = increment \wedge S(i+1)(tail(tr)) \\ & \vee\ head(tr) = decrement \wedge S(i-1)(tail(tr)) \end{array}$$

which expands to

$$\begin{array}{lll} \multicolumn{3}{l}{increment \rightarrow COUNT(i+1) \mid decrement \rightarrow COUNT(i-1)} \\ \quad \mathbf{sat} & \multicolumn{2}{l}{tr = \langle\rangle} \\ & \vee & head(tr) = increment \wedge \\ & & tail(tr) \downarrow decrement \leqslant i + 1 + tail(tr) \downarrow increment \\ & \vee & head(tr) = decrement \wedge \\ & & tail(tr) \downarrow decrement \leqslant i - 1 + tail(tr) \downarrow increment \end{array}$$

and each disjunct implies $S(i)(tr)$, which establishes the case.

It follows that $COUNT(i)$ **sat** $S(i)(tr)$ for each $i \in I$, and so (in the special case $i = 0$) that $COUNT(0)$ **sat** $tr \downarrow decrement \leqslant tr \downarrow increment$. $\square$

EXAMPLE 5.11 The general buffer process $BUFFER = BUFFER(\langle\rangle)$ is intended to satisfy the specification $tr \Downarrow out \leqslant tr \Downarrow in$. The process $BUFFER(\langle\rangle)$ is one of a family of processes indexed by sequences of messages: the sequence is intended to represent the contents of the buffer.

$$\begin{aligned}
BUFFER(\langle\rangle) &= in?x : M \rightarrow BUFFER(\langle x\rangle) \\
BUFFER(\langle y\rangle \frown s) &= in?x : M \rightarrow BUFFER(\langle y\rangle \frown s \frown \langle x\rangle) \\
&\quad\ \mid out!y \rightarrow BUFFER(s)
\end{aligned}$$

The corresponding family of specifications is $S(s)(tr) = tr \Downarrow out \leqslant s \frown (tr \Downarrow in)$. The output stream of a buffer with contents $s$ will begin with $s$, and continue with the sequence of messages that have been input to $BUFFER(s)$.

The family of functions defining the $BUFFER(s)$ processes preserves the family of specifications $\underline{S}(tr)$, and so $BUFFER(s)$ **sat** $S(s)(tr)$ for each sequence of messages $s$; and in particular, $BUFFER$ **sat** $tr \Downarrow out \leqslant tr \Downarrow in$. $\square$

## 5.4 CASE STUDY: DISTRIBUTED SUM

This case study illustrates the use of CSP in the description, analysis, and verification of a distributed algorithm to sum a collection of values arranged in a graph. Each node follows its own procedure locally and communicates only with its neighbours, but the output of all this activity is the global sum of all the values.

Let $G = (N, E)$ be a bidirectional (symmetric) connected graph with a set of nodes $N$ and edges $E \subseteq N \times N$. This may be viewed as a connected network of processes which may communicate only with their neighbours.

The graph $G$ has a weight $w_n$ associated with each node $n$. The algorithm of Figure 5.1 calculates the sum of all the weights. Each node $n$ waits for one of its neighbours to send it an initiating message. It records this neighbour as its *parent*. It then sends all of its other neighbouring nodes an initiating message, and simultaneously awaits messages from all of these neighbours: some may be initiating messages, and some may be values. When all of these have been received, the node sends to its parent the sum of all the values received and $w_n$, after which it terminates.

In order to start the algorithm, one special node must be initiated from outside the graph, and return its final result outside the graph: this final result will be the sum of all the weights.

1. `Receive an initiating message from some neighbour` $j$`;`
2. `Send out one initiating message to each of the other neighbours, and receive initiating messages or values from them;`
3. `Add up all the values received, add the weight` $w_n$`, and send the result to node` $j$`.`

***Fig. 5.1*** Behaviour of each node $n$ in the distributed sum algorithm

An example execution is pictured in Figure 5.2, where the nodes are annotated with their weights. The associated communications are given in Figure 5.3. The top node is activated, and sends initiating messages to its neighbouring nodes, which send initiating messages to all of their neighbours in turn. Once activated, a node observes initiating messages and values arriving from other neighbours, and when it has heard from all of its neighbours then it sends the sum total of the values it received plus its own weight back to its parent node. Any pairs of adjacent nodes for which neither is the parent of the other will simply exchange initiating messages.

The execution finally ends when all nodes have communicated to their parent a value consisting of the sum of all values received from their children together with their own weight.

The algorithm will be described and verified in CSP. It is first necessary to settle some appropriate notation. The nodes are named using integers from 0 to $m$ (where there are $m + 1$ nodes in total), so $N = \{i \mid 0 \leqslant i \leqslant m\}$, and 0 is the initial node.

For any node $i \in N$ apart from 0, its set of neighbours or adjacent nodes $adj(i)$ is given by

$$adj(i) \quad = \quad \{j \in N \mid (i,j) \in E\}$$

Node 0 also has $\infty$ in its set $adj(0)$ in addition to its neighbouring nodes, representing its external link.

$$adj(0) \quad = \quad \{j \in N \mid (0,j) \in E\} \cup \{\infty\}$$

The CSP description of the algorithm will describe each node as a CSP process. The nodes are connected in accordance with the graph $G$. Between any two neighbouring nodes $i$ and $j$ there is a communication channel $c_{ij}$ allowing messages to pass from $i$ to $j$. Since the graph is symmetric, for each such channel there will be a complementary channel ($c_{ji}$) in the opposite direction. The channels used in the CSP implementation of the example network above are illustrated in Figure 5.4.

The values that pass along channels need to be accessed, since the algorithm is concerned with summing them. The notation $v_{il}$ will denote the sum of all messages passed along channel $c_{il}$.

$$v_{il}(tr) = \mathsf{sum}(\langle tr \Downarrow c_{il} \rangle)$$

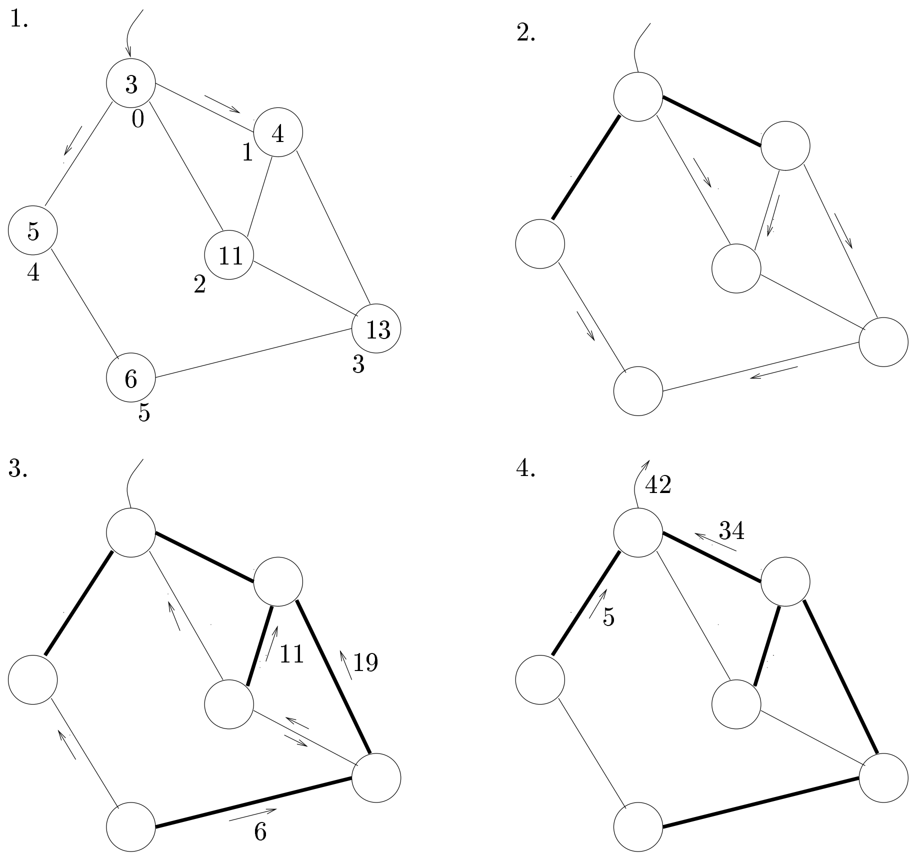


***Fig. 5.2*** Steps of an execution of the distributed sum algorithm; parent edges highlighted

Node 0 begins in the state where it will receive a signal along channel $c_{\infty 0}$, and will then send initiating messages to all of its neighbours and await responses. When it has received all of the responses it communicates the result on the special channel $c_{0\infty}$. Its alphabet is therefore

$$\begin{aligned} A_0 \quad &= \quad \{c_{0l}.n \mid n \in \mathbb{N} \wedge l \in adj(0)\} \cup \{c_{l0}.n \mid n \in \mathbb{N} \wedge l \in adj(0)\} \\ &\quad\quad \cup c_{0\infty}.\mathbb{N} \cup c_{\infty 0}.\mathbb{N} \end{aligned}$$

The alphabets of the other nodes $i \neq 0$ are simply the links with their neighbours:

$$A_i \quad = \quad \{c_{il}.n \mid n \in \mathbb{N} \wedge l \in adj(i)\} \cup \{c_{li}.n \mid n \in \mathbb{N} \wedge l \in adj(i)\}$$

The algorithm is expressed by describing in CSP how each node should behave. One optimization is to consider each initiating message as a communication of the value 0. An

| NODE(0) | NODE(1) | NODE(2) | NODE(3) | NODE(4) | NODE(5) | |
|---|---|---|---|---|---|---|
| →0:i | | | | | | |
| 0→1:i | 0→1:i | | | | | 1 |
| 0→4:i | | | | 0→4:i | | |
| | 1→3:i | | 1→3:i | | | |
| | 1→2:i | 1→2:i | | | | |
| 0→2:i | | 0→2:i | | | | 2 |
| | | | 3→5:i | | 3→5:i | |
| | | | | 4→5:i | 4→5:i | |
| | | | | 5→4:i | 5→4:i | |
| | | | 5→3:6 | | 5→3:6 | |
| | | 3→2:i | 3→2:i | | | |
| 2→0:i | | 2→0:i | | | | 3 |
| | | 2→3:i | 2→3:i | | | |
| | 3→1:19 | | 3→1:19 | | | |
| | 2→1:11 | 2→1:11 | | | | |
| 4→0:5 | | | | 4→0:5 | | |
| 1→0:34 | 1→0:34 | | | | | 4 |
| 0→ :42 | | | | | | |

***Fig. 5.3*** Communications associated with the execution of Figure 5.2

active node will ignore other initiating messages, which is equivalent to adding 0 to the running total, so there is no need to distinguish between the input of an ignored initiating message and the input of a 0 which is added to the total. This identification removes the need to consider these two cases separately, and allows for a more concise treatment of node behaviour.

The system as a whole consists of a network of nodes, with all of the channels between them made internal:

$$\begin{aligned} DISTSUM &= NETWORK \setminus \{c_{ij} \mid (i,j) \in E\} \\ NETWORK &= \big\|_{A_i}^{i \in N} NODE(i) \end{aligned}$$

The property that will be established for *DISTSUM* is that on termination the value communicated on channel $c_{0\infty}$ is indeed the sum of the weights on the nodes:

$$DISTSUM \quad \mathbf{sat} \quad term(tr) \Rightarrow v_{0,\infty}(tr) = \Sigma_{i \in N} w_i$$

In order to establish this property, it is first necessary to define the component *NODE* processes. This will be done in terms of a family of processes $\underline{TOT}$ which keep track of the relevant

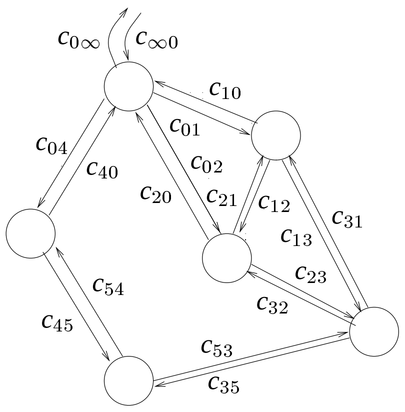


***Fig. 5.4*** Channels of the CSP implementation

interactions between nodes, and which sum values as they arrive. The process $TOT(i, j, M, t)$ contains in its state information:

1. the identity of its node $i$;
2. its parent node $j$;
3. the set of neighbours $M \subseteq adj(i)$ it still awaits inputs from;
4. the running total $t$ (initially $w_i$)

Its behaviour will be to accept input from all the nodes listed in $M$, keeping track of the running total in $t$, and finally sending this total back to $j$.

The notation $\underline{F}$ will be used to refer to the function implicit in the definition of $\underline{TOT}$: the fixed point of $F$ is $\underline{TOT}$.

The special treatment of $NODE(0)$ requires it to be defined slightly differently to the other nodes:

$$NODE(0) \;=\; c_{\infty 0}.0 \to (TOT(0,\infty,adj(0),w_0) \parallel (\big\Vert^{k\in adj(0)} c_{0k}!0 \to SKIP))$$

$$\begin{aligned} NODE(i) \;=\; \square_{j\in adj(i)}\, c_{ji}.0 \to (\;& TOT(i,j,adj(i)\setminus\{j\},w_i) \\ & \parallel \big\Vert^{k\in adj(i)\setminus\{j\}} c_{ik}!0 \to SKIP) \end{aligned}$$

where the $TOT$ processes are defined by

$$\begin{aligned} TOT(i,j,\{\},t) &= c_{ij}!t \to SKIP \\ TOT(i,j,M,t) &= \square_{k\in M}\, c_{ki}?x \to TOT(i,j,M\setminus\{k\},t+x) \quad \text{if } M \neq \{\} \end{aligned}$$

The processing of the incoming values accomplished by *TOT* can be carried out concurrently with the transmission of the initiating messages to the neighbours.

The aim is first to prove that each $TOT(i,j,M,t)$ provides to its parent the sum of the values it has received together with the running total $t$ it was initialized with. In CSP, the process must be shown to meet the $S(i,j,M,t)(tr)$ as follows:

$$TOT(i,j,M,t) \quad \mathbf{sat} \quad term(tr) \Rightarrow v_{ij}(tr) = \Sigma_{l \in adj(i)} v_{li}(tr) + t$$

This is proven by recursion induction. Assume as the inductive hypothesis that $Y(i,j,M,t)$ **sat** $S(i,j,M,t)(tr)$ for each $i,j,M,t$. Then it is sufficient to prove for each $i$, $j$, $M$, and $t$ that $\underline{F}(\underline{Y})(i,j,M,t)$ **sat** $S(i,j,M,t)(tr)$ . Following the definition of $\underline{TOT}$ this is established by considering various cases on $M$.

Case $M = \{\}$: In this case $\underline{F}(\underline{Y})(i,j,\{\},t) = c_{ij}!t \rightarrow SKIP$, and

$$c_{ij}!t \rightarrow SKIP \quad \mathbf{sat} \quad tr = \langle\rangle \vee tr = \langle c_{ij}.t\rangle \vee tr = \langle c_{ij}.t, \checkmark\rangle$$

and so by weakening the specification the result

$$c_{ij}!t \rightarrow SKIP \quad \mathbf{sat} \quad term(tr) \Rightarrow v_{ij} = \Sigma_{l \in adj(i)} v_{li}(tr) + t$$

is obtained, since this specification is true of the trace $\langle c_{ij}.t, \checkmark\rangle$, and vacuously true for the other traces.

Case $M \neq \{\}$: In this case

$$\underline{F}(\underline{Y})(i,j,M,t) = \Box_{k \in M} c_{ki}?x \rightarrow Y(i,j,M \setminus \{k\}, t+x)$$

and

$$\begin{aligned}
&\Box_{k \in M} c_{ki}?x \rightarrow Y(i,j,M \setminus \{k\}, t+x)\\
&\quad \mathbf{sat} \quad tr = \langle\rangle\\
&\qquad\qquad \vee\ \exists\, k \in adj(i), v \bullet (tr = \langle c_{ki}.v\rangle \frown tr' \wedge S(i,j,M \setminus \{k\}, t+v)(tr'))
\end{aligned}$$

Consider the second disjunct of this specification. In this case $tr \neq \langle\rangle$, so $v_{ki}(tr) = v_{ki}(tr') + v$ because of the first event of $tr$. Observe further that $v_{li}(tr) = v_{li}(tr')$ when $l \neq k$, that $v_{ij}(tr) = v_{ij}(tr')$, and also that $term(tr) \Leftrightarrow term(tr')$. The specification is weakened to yield the following

$$\begin{aligned}
&\Box_{k \in M} c_{ki}.0 \rightarrow Y(i,j,M \setminus \{k\}, t)\\
&\quad \mathbf{sat} \quad tr = \langle\rangle\\
&\qquad\qquad \vee\ \exists\, k \in adj(i), v \bullet ((term(tr) \Rightarrow v_{ij}(tr) = \Sigma_{l \in adj(i)} v_{li}(tr) + t))
\end{aligned}$$

Since $k$ and $v$ no longer appear within the scope of the existential quantification, it may be dropped. A final weakening reveals that

$$\begin{array}{l} \Box_{k \in M} c_{ki}.0 \to Y(i,j,M \setminus \{k\},t) \\ \quad \textbf{sat} \quad term(tr) \Rightarrow v_{ij}(tr) = \Sigma_{l \in adj(i)} v_{li}(tr) + t \end{array}$$

This establishes that the specification is preserved by recursive calls. Since the specification is satisfiable, this means that $TOT(i,j,M,t)$ **sat** $S(i,j,M,t)$ for all $i, j, M$, and $t$.

The definition of $NODE(i)$ is also made up of components of the form $c_{ik}!0 \to SKIP$, so these will now be considered. For any arbitrary $k$

$$c_{ik}!0 \to SKIP \quad \textbf{sat} \quad v_{ik}(tr) = 0$$

A parallel combination of such processes satisfies the conjunction of these specifications, restricted to the appropriate alphabets:

$$\Big\|_{c_{ik}}^{k \in adj(i) \setminus \{j\}} c_{ik}!0 \to SKIP \quad \textbf{sat} \quad \forall k \in adj(i) \setminus \{j\} \bullet v_{ik}(tr \Downarrow \{c_{ik}, \checkmark\}) = 0$$

However, each of these specifications depends only on the events in the corresponding alphabets: $v_{ik}(tr \upharpoonright (c_{ik}.\mathbb{N} \cup \{\checkmark\})) = 0 \Leftrightarrow v_{ik}(tr) = 0$, and $S(i,j,M,t)(tr) \Leftrightarrow S(i,j,M,t)(tr \upharpoonright c_{ij}.\mathbb{N} \cup \{\checkmark\} \cup \bigcup\{c_{li}.\mathbb{N} \mid l \in adj(i)\}$. This means that the restrictions to the appropriate alphabets can be lifted, and the parallel combination satisfies the conjunction of the specifications on the full unrestricted trace $tr$.

$$\begin{array}{l} TOT(i,j,M,t) \parallel (\Big\|^{k \in adj(i) \setminus \{j\}} c_{ik}!0 \to SKIP) \\ \quad \textbf{sat} \quad S(i,j,M,t)(tr) \wedge \forall k \in adj(i) \setminus \{j\} \bullet v_{ik}(tr) = 0 \end{array}$$

This specification may be weakened, resulting in

$$\begin{array}{l} TOT(i,j,M,t) \parallel (\Big\|^{k \in adj(i) \setminus \{j\}} c_{ik}!0 \to SKIP) \\ \quad \textbf{sat} \quad term(tr) \Rightarrow \Sigma_{l \in adj(i)} v_{il}(tr) = \Sigma_{l \in adj(i) \setminus \{j\}} v_{li}(tr) + t \end{array}$$

Hence for any $j \in adj(i)$, after some manipulations similar to those above, the following is obtained:

$$\begin{array}{l} c_{ji}.0 \to (TOT(i,j,adj(i) \setminus \{j\}, w_i) \parallel (\Big\|^{k \in adj(i) \setminus \{j\}} c_{ik}!0 \to SKIP)) \\ \quad \textbf{sat} \quad term(tr) \Rightarrow \Sigma_{l \in adj(i)} v_{il}(tr) = \Sigma_{l \in adj(i)} v_{li}(tr) + w_i \end{array}$$

The process for each possible $j$ allows the specification, so the choice over all $j \in adj(i)$ meets the same specification:

$$\begin{array}{l} \Box_{j \in adj(i)} c_{ji}.0 \to (TOT(i,j,adj(i) \setminus \{j\}, w_i) \parallel (\Big\|^{k \in adj(i) \setminus \{j\}} c_{ik}!0 \to SKIP)) \\ \quad \textbf{sat} \quad term(tr) \Rightarrow \Sigma_{l \in adj(i)} v_{il}(tr) = \Sigma_{l \in adj(i)} v_{li}(tr) + w_i \end{array}$$

This specification will be abbreviated by $term(tr) \Rightarrow S(i)(tr)$

Since this choice is how $NODE(i)$ is defined (for $i \neq 0$), it has now been established that

$$NODE(i) \quad \textbf{sat} \quad term(tr) \Rightarrow S(i)(tr)$$

An entirely similar train of reasoning leads to the result that

$$NODE(0) \quad \textbf{sat} \quad term(tr) \Rightarrow \Sigma_{l \in adj(0)} v_{l0}(tr) + w_0 = \Sigma_{l \in adj(0)} v_{0l}(tr) + v_{0\infty}(tr)$$

which will be abbreviated as $NODE(0)$ **sat** $term(tr) \Rightarrow S(0)(tr)$. The only difference from the specifications of the other nodes is that the extra value $v_{0\infty}$ is mentioned separately, as $c_{0\infty}$ refers to the channel that node 0 uses to communicate its result outside the graph.

Each $NODE(i)$ has an alphabet $A_i$. Observe that $(term(tr) \Rightarrow S(i)(tr)) \Leftrightarrow (term(tr \upharpoonright A_i^{\checkmark}) \Rightarrow S(i)(tr \upharpoonright A_i^{\checkmark}))$.

Hence the network meets the conjunction of these specifications (each suitably restricted):

$$\big\|_{A_i} NODE(i) \quad \textbf{sat} \quad \forall\, i \in N \bullet term(tr \upharpoonright A_i^{\checkmark}) \Rightarrow S(i)(tr \upharpoonright A_i^{\checkmark})$$

This specification simplifies to the form

$$term(tr) \Rightarrow \forall\, i \in N \bullet S(i)(tr)$$

which in turn is equivalent to

$$\begin{array}{l} term(tr) \Rightarrow \\ \qquad \begin{array}{rcl} \Sigma_{i \in N} w_i & = & \Sigma_{i \in N}(\Sigma_{j \in adj(i)} v_{ij}(tr) - \Sigma_{j \in adj(i)} v_{ji}(tr)) + v_{0\infty}(tr) \\ & = & \Sigma_{i \in N}(\Sigma_{j \in adj(i)} v_{ij}(tr)) - \Sigma_{i \in N}(\Sigma_{j \in adj(i)} v_{ji}(tr)) + v_{0\infty}(tr) \\ & = & \Sigma_{(i,j) \in E} v_{ij}(tr) - \Sigma_{(i,j) \in E} v_{ji}(tr) + v_{0\infty}(tr) \\ & = & \Sigma_{(i,j) \in E} v_{ij}(tr) - \Sigma_{(i,j) \in E} v_{ij}(tr) + v_{0\infty}(tr) \\ & = & v_{0\infty}(tr) \end{array} \end{array}$$

The penultimate line is justified by the fact that the set of edges $E$ is symmetric: $(i,j) \in E \Leftrightarrow (j,i) \in E$.

This establishes that

$$NETWORK \quad \textbf{sat} \quad term(tr) \Rightarrow v_{0\infty}(tr) = \Sigma_{i \in N} w_i$$

and since this specification is $\{c_{ij} \mid (i,j) \in E\}$-independent, it follows that

$$NETWORK \setminus \{c_{ij} \mid (i,j) \in E\} \quad \textbf{sat} \quad term(tr) \Rightarrow (v_{0\infty}(tr) = \Sigma_{i \in N} w_i)$$

or in other words

$$DISTSUM \quad \textbf{sat} \quad term(tr) \Rightarrow (v_{0\infty}(tr) = \Sigma_{i \in N} w_i)$$

This completes the proof that on termination the sum of the outputs along $c_{0\infty}$ is equal to the sum of the weights on the nodes. Since $NODE(0)$ ensures that at most one value is communicated along channel $c_{0\infty}$, this value must be the sum of the weights.

What has been proven is that if an answer is given out then it will be the right one. This is a safety property: it states that the wrong answer will never be given. Observe that the connectedness of the graph was not used in establishing this property. Connectedness will be needed to show that all nodes in the graph participate in the run, and this does not need to be shown to establish the safety property. Rather, it is already assumed in the antecedent $term(tr)$, since *NETWORK* can terminate only when *all* of its nodes are ready to do so, which requires that they all participate in the execution.

The fact that *DISTSUM* will indeed progress towards termination, and will not deadlock or diverge, will be shown in Chapters 7 and 8, where issues of liveness are addressed.

## 5.5 PROCESS-ORIENTED SPECIFICATION

A specification is simply a description of acceptable or required behaviour. The property-oriented approach described thus far captures specifications in terms of requirements $S(tr)$ on traces that a process can perform. A process meets a specification if all of its traces are acceptable.

Another way of describing a set of acceptable traces is in terms of a CSP process $P_0$. A CSP description corresponds to a set of traces—those traces that it can exhibit. If this set of traces is taken to give precisely those traces that are acceptable, then the process $P_0$ itself acts as a specification. For instance, the process $RUN_{\{a,b\}}$ has as its traces all sequences whose only members are $a$ and $b$ events. As a specification, it captures the requirement that only $a$ and $b$ events are allowed.

Another process $P_1$ meets the specification described by $P_0$ if any trace of $P_1$ is 'allowed' by $P_0$, in the sense that it is a trace of $P_0$. $P_1$ is then considered to be a *refinement* of $P_0$. For instance, $P_1$ might be the recursive process $P_1 = a \to b \to P_1$ which alternates on $a$ and $b$ events. It meets the specification given by $RUN_{\{a,b\}}$, since it performs no events other than $a$'s and $b$'s. This claim is written as $P_0 \sqsubseteq_T P_1$, which is pronounced '$P_0$ is refined by $P_1$ with respect to traces', or '$P_1$ trace-refines $P_0$'. It is defined as follows:

$$P_0 \sqsubseteq_T P_1 \quad = \quad \text{traces}(P_1) \subseteq \text{traces}(P_0)$$

| | |
|---|---|
| $P \sqsubseteq P$ | $\langle \sqsubseteq$-reflex$\rangle$ |
| $P_0 \sqsubseteq P_1 \wedge P_1 \sqsubseteq P_2 \Rightarrow P_0 \sqsubseteq P_2$ | $\langle \sqsubseteq$-trans$\rangle$ |
| $P_0 \sqsubseteq P_1 \wedge P_1 \sqsubseteq P_0 \Rightarrow P_0 = P_1$ | $\langle \sqsubseteq$-anti-sym$\rangle$ |
| $RUN \sqsubseteq_T P$ | $\langle \sqsubseteq_T$-bottom$\rangle$ |
| $P \sqsubseteq_T STOP$ | $\langle \sqsubseteq_T$-top$\rangle$ |
| $P_0 \ \mathbf{sat}\ S(tr) \wedge P_0 \sqsubseteq_T P_1 \Rightarrow P_1 \ \mathbf{sat}\ S(tr)$ | $\langle \sqsubseteq_T$-spec$\rangle$ |

***Fig. 5.5*** Laws for refinement

The ‘traces refinement’ check of FDR (see Appendix B) checks for exactly this refinement relation.

The relation may also be captured algebraically as follows:

$$P_0 \sqsubseteq_T P_1 \quad \Leftrightarrow \quad P_0 =_T P_0 \sqcap P_1$$

Its equivalence to the definition is easily checked, though the interpretation of this characterization is a little different. It states that if $P_0$ is indistinguishable from $P_0 \sqcap P_1$, then any situation where $P_0$ is suitable must allow that $P_0 \sqcap P_1$ is suitable (since this is equal to $P_0$), and so $P_1$ must also be suitable since the internal choice could always be resolved in favour of $P_1$. The process $P_1$ is a refinement of $P_0$ because it will be appropriate in any environment which will find $P_0$ acceptable. An alternative way of thinking about the equivalence is that all of $P_1$'s behaviours must already be allowed by $P_0$, since the introduction of $P_1$ does not introduce any new behaviours. This algebraic characterization of refinement is also appropriate for other semantic models, as will be discussed in later chapters. If the model is clear from the context then the subscript to the refinement symbol will be dropped.

Refinement satisfies a number of laws, given in Figure 5.5: it is reflexive, transitive, and anti-symmetric in all models; the process *RUN* is trace-refined by any other process; *STOP* trace-refines every process; and refinement preserves **sat** specifications.

The resolution of internal choice is a refinement step: $P_0 \sqcap P_1 \sqsubseteq_T P_1$. If either $P_0$ or $P_1$ are acceptable, then certainly $P_1$ by itself is acceptable. Furthermore, all of the CSP operators are monotonic with respect to refinement. What this means is that for any CSP function $F(Y)$ constructed from the CSP operators, the application of $F$ will respect the refinement relation: if $P_0 \sqsubseteq_T P_1$ then $F(P_0) \sqsubseteq_T F(P_1)$. Finally, if

$$\forall Y \bullet (F(Y) \sqsubseteq_T G(Y))$$

then $P_0 = F(P_0) \sqsubseteq_T P_1 = G(P_1)$.

EXAMPLE 5.12 The process-oriented specification $RUN_\Sigma$ specifies that termination may not occur, but imposes no other restriction. □

EXAMPLE 5.13 The specification $P = a \rightarrow (P \mid\mid\mid b \rightarrow STOP)$ specifies that only $a$ and $b$ events may occur, and $b$ may not occur more often than $a$. This process meets the property oriented specification $tr \downarrow b \leqslant tr \downarrow a$.

Now the function defining $P$ may be refined as follows:

$$\begin{aligned} F(Y) &=_T \quad a \rightarrow (Y \mid\mid\mid b \rightarrow STOP) \\ &\sqsubseteq_T \quad a \rightarrow b \rightarrow Y \end{aligned}$$

and so it follows that $P \sqsubseteq_T P_1 = a \rightarrow b \rightarrow P_1$. The process that alternates on $a$ and $b$ refines the process that allows no more $b$'s than $a$'s. Since refinement preserves **sat** specifications, it follows that

$$P_1 = a \rightarrow b \rightarrow P_1 \quad \textbf{sat} \quad tr \downarrow b \leqslant tr \downarrow a$$

This follows from an application of Law $\sqsubseteq_T$-spec. □

EXAMPLE 5.14 A CSP process expression can describe the behaviour required of the distributed summing network *DISTSUM* described in the case study. The resulting specification on *DISTSUM* is captured by the following refinement requirement:

$$c_{\infty,0}.0 \rightarrow c_{0\infty}!(\Sigma_{i \in N} w_i) \rightarrow SKIP \quad \sqsubseteq_T \quad DISTSUM$$

This states that *DISTSUM* is intended to output the appropriate value on the channel $v_{0\infty}$ before terminating. □

EXAMPLE 5.15 When using CSP process expressions as specifications, it is important to ensure that no acceptable traces are excluded. For example, the requirement that $a$ and $b$ events should alternate (beginning with $a$) might use the recursive process $P = a \rightarrow b \rightarrow P$, but if no constraint is required on other events, then the acceptability of other events has to be included explicitly as a component $RUN_{\Sigma \setminus \{a,b\}}$, and the entire specification will be written

$$P \mid\mid\mid RUN_{\Sigma \setminus \{a,b\}}$$

Using only $P$ as a specification would introduce the additional constraint that no other events may occur. □

The model-checking tool FDR (see Appendex B) allows checks concerning the refinement relationship between two (finite state) CSP processes. This is often the quickest way to conduct process verification once the specification has been captured. The tool also assists debugging of implementations when they do not meet the specification by returning a witness trace which may be performed by the implementation but which is not possible for the specification process.

## Exercises

EXERCISE 5.1 Specify that a lift's doors should not be open when the lift starts moving. Assume that it has events *open*, *close*, *moving*, *stopped* in its alphabet.

EXERCISE 5.2 Specify the hygiene requirement that hands should be washed between handling raw meat and cooked meat. Use the events *wash*, *raw*, and *cooked*, to refer to these three activities.

Does the combination $RAW \underset{\{wash\}}{\|} COOKED$ meet your specification?

$$\begin{aligned} RAW &= raw \to wash \to RAW \\ COOKED &= wash \to cooked \to COOKED \end{aligned}$$

EXERCISE 5.3 What does the predicate $tr \leqslant \langle a, b, c\rangle \frown tr$ specify?

EXERCISE 5.4 What does the predicate $last(tr) = b \Rightarrow a \in \sigma(tr)$ specify?

EXERCISE 5.5 If $P_1$ **sat** $tr \downarrow a \leqslant tr \downarrow b + n$ and $P_2$ **sat** $tr \downarrow a \leqslant tr \downarrow b + m$, then prove that $P_1 \mid\mid\mid P_2$ **sat** $tr \downarrow a \leqslant tr \downarrow b + n + m$.

EXERCISE 5.6 Prove the statements on Page 146, that

$$\begin{array}{lll} P_1 = b \to a \to P_1 & \textbf{sat} & S(tr) = tr \downarrow a \leqslant tr \downarrow b \\ P_2 = c \to b \to P_2 & \textbf{sat} & T(tr) = tr \downarrow b \leqslant tr \downarrow c \end{array}$$

EXERCISE 5.7 Which of the following are sound proof rules for the interleaving operator?

$$\frac{\begin{array}{l} P_1 \ \textbf{sat}\ tr \downarrow A \leqslant m \\ P_2 \ \textbf{sat}\ tr \downarrow A \leqslant n \end{array}}{P_1 \mid\mid\mid P_2 \ \textbf{sat}\ tr \downarrow A \leqslant (m+n)}$$

$$\frac{\begin{array}{l} P_1 \ \textbf{sat}\ tr \downarrow a \leqslant tr \downarrow b \\ P_2 \ \textbf{sat}\ tr \downarrow a \leqslant tr \downarrow b \end{array}}{P_1 \mid\mid\mid P_2 \ \textbf{sat}\ tr \downarrow a \leqslant tr \downarrow b}$$

$$\frac{\begin{array}{l} P_1 \ \textbf{sat}\ tr \downarrow a \leqslant tr \downarrow b \\ P_2 \ \textbf{sat}\ tr \downarrow b \leqslant tr \downarrow c \end{array}}{P_1 \mid\mid\mid P_2 \ \textbf{sat}\ tr \downarrow a \leqslant tr \downarrow c}$$

EXERCISE 5.8 Prove the claims of Example 5.9 on Page 156, that

$$\begin{array}{lll}\forall\, Y \bullet (Y\ \mathbf{sat}\ S_2(tr) & \Rightarrow & on \rightarrow Y\ \mathbf{sat}\ S_1(tr))\\ \forall\, Y \bullet (Y\ \mathbf{sat}\ S_1(tr) & \Rightarrow & \mathit{off} \rightarrow Y\ \mathbf{sat}\ S_2(tr))\end{array}$$

EXERCISE 5.9 Prove by recursion induction that the process $DOOR = (open \rightarrow close \rightarrow DOOR) \,\Box\, locked \rightarrow STOP$ meets the following specifications:

1. two consecutive events are not both *open*;
2. two consecutive events are not both *close* (you will have to prove something stronger);
3. $tr \downarrow close \leqslant tr \downarrow open \leqslant tr \downarrow close + 1$.

EXERCISE 5.10 Prove that $STACK = STACK(\langle\rangle)$ of Example 1.23 on Page 17 meets the specification

$$\forall\, v \bullet pop.v\ \mathbf{in}\ tr \Rightarrow push.v\ \mathbf{in}\ tr$$

EXERCISE 5.11 Specify the requirement that every output value (on channel *out*) must be less than or equal to some input value (on channel *in*), in both the property oriented and the process-oriented specification styles.

EXERCISE 5.12 Specify the requirement that a *write* event should always occur between an *engage* event and a *release* event, as a property oriented and as a process-oriented specification.

EXERCISE 5.13 Specify that a guard should never be up while a piece of machinery is switched on. A property-oriented specification should be expressed in terms of events *guard.up*, *guard.down*, *on* and *off*. Express the same specification in a process-oriented way.

EXERCISE 5.14 Can a node $NODE(i)$ (Page 162) output its total to its parent node before it has sent out all of its initiating messages? Can it terminate before sending out all of its initiating messages?

EXERCISE 5.15 Show that $NODE(0)$ satisfies the following specifications

1. $tr \Downarrow c_{0\infty} \neq \langle\rangle \Rightarrow tr \Downarrow c_{\infty 0} \neq \langle\rangle$
2. $tr \upharpoonright \checkmark \neq \langle\rangle \Rightarrow tr \Downarrow c_{0\infty} \neq \langle\rangle$
3. $(tr \Downarrow c_{\infty 0}) \leqslant \langle 0\rangle$
4. $tr \downarrow c_{0\infty}.\mathbb{N} \leqslant 1$

EXERCISE 5.16 Show that $NODE(i)\ \mathbf{sat}\ \#tr \leqslant 1 + 2* \mid adj(i) \mid$

# 6
# Stable failures

The traces model for CSP is concerned only with the sequences of events that processes may perform. Observing a process involves recording events as they occur during an execution. This view is appropriate for the analysis of safety, since the traces associated with a process provide sufficient information to verify safety properties.

Liveness properties are concerned with behaviour that processes are guaranteed to make available. Where safety properties are generally of the form ‘something bad will not happen’, liveness properties are of the form ‘something good will happen’. With the view of processes as interacting components, a process in isolation can never by itself guarantee that any particular event will happen at any point, since its environment may always prevent the event from occurring by refusing to co-operate. However, a process might be able to guarantee the occurrence of events under particular assumptions about what its environment is prepared to allow. It is appropriate to think in terms of what the process is prepared to do rather than what it is guaranteed to do.

For example, a choice process $P_1 = a \to STOP \Box b \to STOP$ is prepared to perform both $a$ and $b$, but neither of these possibilities is guaranteed to occur, since the resolution of the choice is dependent on the environment of the process, and this will not be contained in any description of the process itself. However, the process will be guaranteed to perform $a$ if this is offered by the environment, and similarly for $b$.

On the other hand, if choices are made internally within the process, then some possibilities (as recorded in the traces) are not guaranteed. The internal choice process $P_2 = a \to STOP \sqcap b \to STOP$ has the same traces as $P_1$ but provides different guarantees. An environment which wishes to interact on $a$ is not sure of doing so, and neither is an environment offering $b$, despite the fact that these two events are both possibilities for $P_2$. In fact, an environment needs to be prepared to interact on both $a$ and $b$ to be sure of obtaining some response from $P_2$, though the actual response is unpredictable. $P_2$ might refuse to interact if only $a$ is offered to it, or only $b$, but it cannot if both $a$ and $b$ are simultaneously offered to it.

Trace information is in general too coarse to identify the guaranteed responses of a process. This is apparent from the fact that $P_1$ and $P_2$ have the same traces but different guaranteed behaviour, and more generally from the fact that internal and external choice have the same trace semantics, so they both give rise to the same possibilities, yet exhibit different behaviours in some contexts. Some finer form of process observation is required in order to make the necessary distinctions and provide the desired information about guaranteed process behaviour.

## 6.1 OBSERVING PROCESSES

### Stable refusals

A process $P$ is guaranteed to be able to respond to an offer of an event $a$ if that event can be performed from $P$, provided there are no internal transitions from $P$ which might result in withdrawal of this offer. A process $P$ which can make no internal progress is said to be *stable*, written $P\downarrow$ :

$$P\downarrow \quad = \quad \neg(P \xrightarrow{\tau}\ )$$

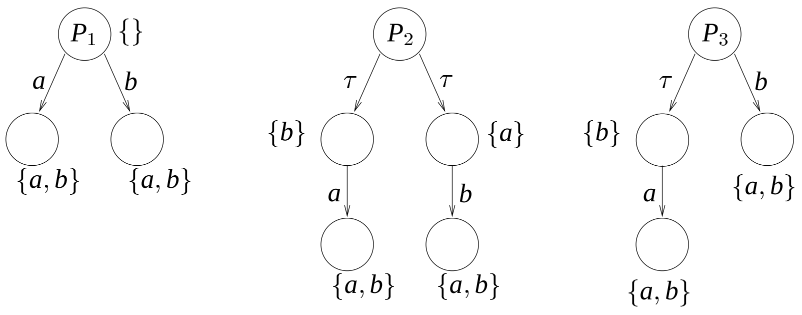
Guarantees are concerned with stable states.

More generally, a stable process $P$ can always respond in some way to the offer of a set of events $X \subseteq \Sigma^{\checkmark}$ if there is at least one $a \in X$ that $P$ can perform. If there is no such $a \in X$, then $P$ *refuses* the entire offer set $X$.

The CSP approach to semantics is to associate processes with observations of their executions, and then to use this information to understand the behaviour of the process as a whole. A single execution of a process $P$ consisting of internal transitions leading to a stable state $P'$ will not provide information about the events that are *guaranteed* to be offered, but will rather provide information about events that can *possibly* be refused. If no events in a set $X$ are possible in the stable state $P'$, then when $P$ is initially offered $X$ it is possible that it will reach a stable state ($P'$) which deadlocks under that offer—no further progress can be made. In this case, the set $X$ is termed a *refusal* of $P$.

A refusal might be thought of as one result of an experiment on the process $P$, where it is executed in an environment which offers the set $X$, and waits as long as necessary to see if any events in $X$ are performed. If no events are performed, then $X$ is considered a refusal of $P$, written $P$ ref $X$. The assertion $P$ ref $X$ that $P$ can possibly refuse the set $X$ is defined as follows:

$$P\ \mathsf{ref}\ X \quad = \quad \exists\, P' \bullet P \overset{\langle\rangle}{\Longrightarrow} P' \wedge P'\downarrow\ \wedge \forall\, a \in X \bullet \neg(P' \xrightarrow{a}\ )$$



$$\begin{array}{lcl} P_1 & = & a \rightarrow STOP \,\Box\, b \rightarrow STOP \\ P_2 & = & a \rightarrow STOP \sqcap b \rightarrow STOP \\ P_3 & = & (c \rightarrow a \rightarrow STOP \,\Box\, b \rightarrow STOP) \setminus \{c\} \end{array}$$

***Fig. 6.1*** Three processes and their stable states labelled with refusals

Another possible result of the experiment is that some event from $X$ is performed. This will be recorded as trace information. The final possible result is that $P$ performs internal transitions for ever, never reaching a stable state nor performing any event. In this case, $P$ is said to be *divergent*, written $P\uparrow$.

$$P\uparrow \quad = \quad \exists \langle P_i \rangle_{i \in \mathbb{N}} \bullet (P = P_0 \wedge \forall\, i \bullet P_i \xrightarrow{\tau} P_{i+1})$$

A process is non-divergent if it does not diverge, and it is divergence-free if none of its reachable states diverge.

The offer of a set of events $A$ will be guaranteed some response from a non-divergent process $P$ precisely when $A$ is not a possible refusal set for $P$.

The refusals of a process $P$ are concerned with the sets of events that might be refused by $P$ before any visible events have occurred. Refusals thus provide information about initial behaviour. The notion of refusal also extends to other stages of an execution. In general, an observer will experiment on a process by repeatedly offering to interact on sets of events, where each offer is either accepted by the process, or not. Once they are made, offers are not withdrawn by the observer, so if an offer is not accepted by the process then the experiment ends.

EXAMPLE 6.1 The transition graphs and associated refusal sets of the following three processes are illustrated in Figure 6.1. Each of them is able to perform only events $a$ and $b$, so all other events will automatically be refused at any stable node, and are not included explicitly.

Since the refusal sets associated with a process state are subset closed, only the maximal refusal set in each case is included.

The process $P_1 = a \to STOP \;\Box\; b \to STOP$ is unable to refuse either $a$ or $b$ in its initial state, but can refuse both of these events after it has performed something.

The process $P_2 = a \to STOP \sqcap b \to STOP$ is unstable, as there are two internal transitions that are possible for it. Each of these leads to a stable state where either $a$ or $b$ is possible, and the other can be refused.

The process $P_3 = (c \to a \to STOP \;\Box\; b \to STOP) \setminus \{c\}$ is initially unstable, although it can perform the event $b$ from its initial unstable state, after which it can refuse $\{a, b\}$. However, there is no refusal set associated with the initial unstable state, and the single internal transition leads to a state in which $b$ is refused. This means that an interacting process wishing to synchronize on $b$ event might succeed, but it is also possible that the internal event will occur first and the $b$ will then be refused. There is no guarantee that $b$ will be accepted, since the internal transition is entirely under the control of process $P_3$ itself and cannot be prevented from occurring. $\Box$

## Stable failures

It is possible that at some point during an execution an offer set $X$ will be refused by the process $P$. This refusal will be recorded together with the finite sequence of events $tr$ that were performed during the execution leading up to the refusal of $X$. The observation $(tr, X)$ is called a *stable failure* of $P$, recording the fact that

$$\exists P'' \bullet P \overset{tr}{\Longrightarrow} P'' \wedge P'' \downarrow \;\wedge P'' \text{ ref } X$$

The process may perform the events in $tr$, and then reach a stable state where it refuses all of the events in the set $X$. If after the performance of $tr$ it is in an environment in which events from the set $X$ are possible but no others then there will be no further progress.

EXAMPLE 6.2 Figure 6.2 gives the transition graph of the process $P$ defined as follows:

$$\begin{aligned} P \quad = \quad & (a \to (c \to STOP \sqcap d \to STOP) \;\Box\; b \to STOP) \\ & \sqcap \\ & (b \to c \to STOP \;\Box\; (c \to (f \to d \to STOP \;\Box\; e \to STOP) \setminus f)) \end{aligned}$$

There are two stable states $P$ can reach purely by performing internal transitions, corresponding to the trace $\langle\rangle$. These reflect the ways the top level choice can be resolved. One of these states is able to refuse the set $\{c, d\}$, so $(\langle\rangle, \{c, d\})$ is a possible failure of $P$. However, $(\langle\rangle, \{b\})$ is not a failure of $P$, since both stable states are able to perform $b$—neither can refuse it. Similarly, $(\langle\rangle, \{a, c\})$ is not a failure of $P$ since each stable state is able to perform some event from the set $\{a, c\}$, even though $\{a\}$ and $\{c\}$ can be refused separately.

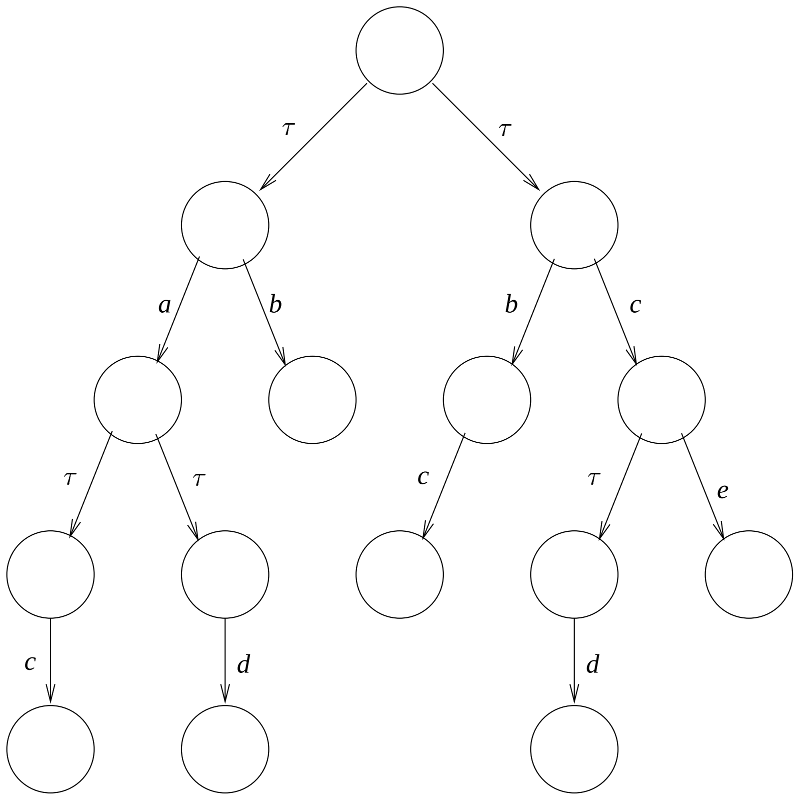


***Fig. 6.2*** Transitions of process $P$ of Example 6.2

Subsequent to the performance of the $a$ event, there are again two stable states that can be reached. One of them is unable to perform any of the set $\{a, b, c\}$, so $(\langle a\rangle, \{a, b, c\})$ is a failure of $P$.

There are two stable states corresponding to the trace $\langle b\rangle$. One of them is able to refuse $c$, so $(\langle b\rangle, \{c\})$ is a failure of $P$. On the other hand, $c$ is possible from the other stable state, so $\langle b, c\rangle$ is a possible trace of $P$, and $(\langle b, c\rangle, \{\})$ is a possible failure.

Finally, there is a single stable state subsequent to an initial $c$ event, and $e$ is not possible from that state, though it is transiently possible immediately after the $c$. Thus $(\langle c\rangle, \{e\})$ is a failure of $P$. $\square$

## Semantic model

The stable failures model for CSP identifies a process $P$ with the traces and the stable failures that are associated with it. This model is more discriminating and hence less abstract than the traces model, but the underlying approach taken to the semantics and to specification and verification is the same. The extra information associated with processes allows them to be analyzed with respect to additional specifications, such as those concerned with liveness requirements.

If two sets $T$ and $SF$ of traces and of stable failures respectively are to correspond to the possible behaviours of some process, there are some consistency conditions that they should meet. These are properties that must hold of any pair of sets which describe some process.

As in the traces model, the set $T$ should meet $T1$ and $T2$ of Page 90: it must be empty and prefix closed. Consistency between $SF$ and $T$ requires that any failure $(tr, X) \in SF$ must have its trace recorded in $T$:

$$SF1 \qquad (tr, X) \in SF \Rightarrow tr \in T$$

There is also a property of subset closure in the refusal component of a behaviour: if a set $X$ can be refused after a trace $tr$, then any subset $X'$ of $X$ can also be refused after that trace.

$$SF2 \qquad (tr, X) \in SF \wedge X' \subseteq X \Rightarrow (tr, X') \in SF$$

Thirdly, if a stable state has been reached from which no events in a set $X'$ are possible, then the refusal set can be augmented with the set $X'$:

$$SF3 \qquad (tr, X) \in SF \wedge \forall\, a \in X' \bullet tr \frown \{a\} \notin T \Rightarrow (tr, X \cup X') \in SF$$

Finally, any terminating trace results in a stable state in which no further events are possible (and so any set can be refused):

$$SF4 \qquad tr \frown \langle \checkmark \rangle \in T \Rightarrow (tr \frown \langle \checkmark \rangle, X) \in F$$

## 6.2 PROCESS SEMANTICS

Each CSP process expression will be associated with appropriate traces and stable failures. These are defined compositionally, so the behaviours associated with a composite process will be defined in terms of the behaviours of its components. The definitions of the traces $\mathsf{traces}(P)$ associated with processes are those of the traces model given in Chapter 4 and are not repeated here. The stable failures associated with a CSP process expression $P$ will be given by $\mathcal{SF}\,[\![P]\!]$.

### *STOP*

The process *STOP* is a stable, deadlocked process. It is not able to perform any event, and can refuse anything.

$$\mathcal{SF}\,[\![STOP]\!] \quad = \quad \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}$$

### Prefixing

In a stable failure of the process $a \to P$, there are two possibilities: either the event $a$ has not occurred, in which case the trace must be $\langle\rangle$ and $P$ is in its stable initial state, able to refuse any event other than $a$; or else the event $a$ has occurred and the rest of the stable failure derives from process $P$.

$$\begin{aligned}\mathcal{SF}\,[\![a \to P]\!] \quad = \quad & \{(\langle\rangle, X) \mid a \notin X\} \\ & \cup \\ & \{(\langle a\rangle \frown tr, X) \mid (tr, X) \in \mathcal{SF}\,[\![P]\!]\}\end{aligned}$$

### Prefix choice

A failure of the process $x : A \to P(x)$ is again one of two possibilities. Either no event has yet occurred, in which any events apart from those in $A$ can be refused; or else an event $a$ in $A$ has occurred, and the subsequent behaviour is that of the corresponding process $P(a)$.

$$\begin{aligned}\mathcal{SF}\,[\![x : A \to P(x)]\!] \quad = \quad & \{(\langle\rangle, X) \mid A \cap X = \{\}\} \\ & \cup \\ & \{(\langle a\rangle \frown tr, X) \mid a \in A \wedge (tr, X) \in \mathcal{SF}\,[\![P(a)]\!]\}\end{aligned}$$

## *SKIP*

The atomic process *SKIP* is used to denote successful termination, and it signals this by means of the termination event $\checkmark$. This is the only event it can perform, and it is stable before and after this event. All other events will be refused before termination, and all events will be refused after termination.

$$\begin{aligned}\mathcal{SF}\,[\![SKIP]\!] \quad = \quad & \{(\langle\rangle, X) \mid \checkmark \notin X\} \\ & \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}\end{aligned}$$

## *DIV*

It is useful to identify the process which does nothing except diverge. This process is denoted *DIV*. It has the same traces as *STOP*, but it has no stable states at all, and hence no stable failures:

$$\begin{aligned}\mathsf{traces}(DIV) \quad &= \quad \{\langle\rangle\} \\ \mathcal{SF}\,[\![DIV]\!] \quad &= \quad \{\}\end{aligned}$$

This process was not introduced in the traces model, so its traces are also given here. It is the minimal process in the stable failures model, because this model records only stable behaviour, and *DIV* does not have any. The model turns a blind eye to divergent behaviour, so the internal activity of this process is not observed. It will be given a more accurate treatment when divergent behaviours are considered in Chapter 8.

## *CHAOS*

The process which can do absolutely anything except diverge is *CHAOS*. This is able to accept or refuse any events, but it is at least guaranteed to stabilize. It has all possible stable failures, and the same traces as *RUN*:

$$\begin{aligned} \mathsf{traces}(CHAOS) &= TRACE \\ \mathcal{SF}[\![CHAOS]\!] &= TRACE \times \mathbb{P}(\Sigma^{\checkmark}) \end{aligned}$$

Chaotic behaviour may be restricted to a particular set of events $A \subseteq \Sigma^{\checkmark}$. The process $CHAOS_A$ allows any events in the set $A$ to be performed or refused, but cannot perform any events outside the set $A$.

$$\begin{aligned} \mathsf{traces}(CHAOS_A) &= \{tr \mid \sigma(tr) \subseteq A\} \\ \mathcal{SF}[\![CHAOS_A]\!] &= \{(tr, X) \mid \sigma(tr) \subseteq A\} \end{aligned}$$

## *RUN*

Although they have the same traces, in the stable failures model *RUN* is better behaved than *CHAOS*, always willing to interact and never refusing any interaction.

$$\mathcal{SF}[\![RUN]\!] = \{(tr, X) \mid X = \{\} \vee \checkmark \in \sigma(tr)\}$$

The process $RUN_A$ parameterized by a particular set $A$ is able to perform events in that set, and to refuse all others.

$$\mathcal{SF}[\![RUN_A]\!] = \{(tr, X) \mid \sigma(tr) \subseteq A \wedge (X \cap A = \{\} \vee \checkmark \in \sigma(tr))\}$$

If $\checkmark \notin A$ then $RUN_A$ cannot terminate.

### External choice

An observer of the choice construct $P_1 \;\Box\; P_2$ might observe an execution of $P_1$, or of $P_2$; there are no other possibilities. Before any events are performed and the choice resolved, any refused set must be refused by both $P_1$ and $P_2$, so both processes must be stable. After the choice is resolved, any refusal need be possible only for the process which resolved the choice.

$$\begin{array}{rcl} \mathcal{SF}\,[\![P_1 \,\Box\, P_2]\!] & = & \{(\langle\rangle, X) \mid ((\langle\rangle, X) \in \mathcal{SF}\,[\![P_1]\!] \cap \mathcal{SF}\,[\![P_2]\!])\} \\ & & \cup \\ & & \{(tr, X) \mid tr \neq \langle\rangle \wedge (tr, X) \in \mathcal{SF}\,[\![P_1]\!] \cup \mathcal{SF}\,[\![P_2]\!]\} \end{array}$$

The properties of idempotence, associativity, and commutativity still hold for external choice in the stable failures model. Furthermore, *STOP* is still a unit, though *RUN* is no longer a zero because *P* might not be initially stable. Instead *RUN* $\Box$ *DIV* is its zero. It has the same traces and stable failures as *RUN* apart from on the empty trace, where it is not stable.

$$P \,\Box\, (RUN \,\Box\, DIV) =_{SF} (RUN \,\Box\, DIV) \qquad \langle \Box_{SF}\text{-zero} \rangle$$

The executions of the indexed external choice $\Box_{i \in I} P_i$ are the executions of all of its components. Its stable failures will be those of its components:

$$\begin{array}{rcl} \mathcal{SF}\,[\![\Box_{i \in I} P_i]\!] & = & \{(\langle\rangle, X) \mid ((\langle\rangle, X) \in \bigcap_{i \in I} \mathcal{SF}\,[\![P_i]\!])\} \\ & & \cup \\ & & \{(tr, X) \mid tr \neq \langle\rangle \wedge (tr, X) \in \bigcup_{i \in I} \mathcal{SF}\,[\![P_i]\!]\} \end{array}$$

In the case where the choice is over the empty set of processes, the intersection $\bigcap_{i \in I} \mathcal{SF}\,[\![P_i]\!]$ is taken to include all possible stable failures, since all of them are vacuously in each of the $\mathcal{SF}\,[\![P_i]\!]$. This means that in this case, any refusal is possible on the empty trace. Furthermore, no events are possible. As in the traces model, an empty choice is equivalent to *STOP*

### Internal choice

The internal choice $P_1 \sqcap P_2$ behaves either as $P_1$ or as $P_2$, and its environment exercises no control over which. The possible observations are precisely those that either $P_1$ or $P_2$ are able to exhibit.

$$\mathcal{SF}\,[\![P_1 \sqcap P_2]\!] \;=\; \mathcal{SF}\,[\![P_1]\!] \cup \mathcal{SF}\,[\![P_2]\!]$$

The stable failures of $P_1 \sqcap P_2$ differ from those of $P_1 \,\Box\, P_2$ in the case where no events have been performed: before the choice has been made. When the trace is empty, a refusal of $P_1 \,\Box\, P_2$ must be generated from both participants, whereas in the case of internal choice, only one of the components of $P_1 \sqcap P_2$ is required to contribute to any refusal. Hence $(\langle\rangle, \{a\})$ is a failure of $a \rightarrow STOP \sqcap b \rightarrow STOP$, but is not a failure of $a \rightarrow STOP \,\Box\, b \rightarrow STOP$.

The indexed internal choice $\bigsqcap_{i \in J} P_i$ is able to behave as any of its component processes, and its behaviours will be the union of those of its constituents:

$$\mathcal{SF}[\![ \bigsqcap_{i \in J} P_i ]\!] \quad = \quad \bigcup_{i \in J} \mathcal{SF}[\![P_i]\!]$$

The internal choice operator also distributes over the external choice operator:

$$\boxed{P_1 \sqcap (P_2 \Box P_3) =_{SF} (P_1 \sqcap P_2) \Box (P_1 \sqcap P_3) \qquad \langle\Box\text{-}\sqcap\text{-dist}\rangle}$$

Any set $X$ that is initially offered can either be accepted by one of the three component processes, or it might be refused, either by $P_1$ or by both $P_2$ and $P_3$. The two extra refusal possibilities for the right hand side—that $X$ should be refused by both $P_1$ and $P_2$, or by both $P_1$ and $P_3$—both imply that $P_1$ can refuse $X$, and hence that the left hand side has this as a refusal too.

EXAMPLE 6.3 This law helps to clarify the possible behaviours associated with a drinks machine, which will either return the cash or will offer a choice between a *tea* and a *coffee*.

$$\begin{aligned} &(ret \to STOP) \sqcap (tea \to STOP \Box coffee \to STOP) \\ &\quad = \quad (ret \to STOP \sqcap tea \to STOP) \Box (ret \to STOP \sqcap coffee \to STOP) \end{aligned}$$

This law states that it makes no difference whether the machine first makes its internal decision and then possibly offers a choice to the customer, or whether the customer makes the choice between tea and coffee first and the machine then decides internally whether to service that choice or return the cash. □

## Alphabetized Parallel

In the parallel combination $P_1 \ {}_A\|_B \ P_2$, processes $P_1$ and $P_2$ synchronize on events in $(A \cap B)^{\checkmark}$, and perform their other events independently.

As in the traces model, any trace of the parallel combination projected onto $A^{\checkmark}$ must be a trace of $P_1$. Further, if $P_1$ is able to refuse some events $X$ in its interface $A^{\checkmark}$, then so too is the combination. Similar considerations apply to $P_2$. If synchronization is required for the performance of events, then either component is able independently to block them.

$$\begin{aligned} \mathcal{SF}[\![P_1 \ {}_A\|_B \ P_2]\!] \quad = \quad \{(tr, X) \mid \ & \exists X_1, X_2 : \mathbb{P}(\Sigma^{\checkmark}) \bullet \\ & X \cap (A \cup B)^{\checkmark} = (X_1 \cap A^{\checkmark}) \cup (X_2 \cap B^{\checkmark}) \\ & \wedge (tr \upharpoonright A^{\checkmark}, X_1) \in \mathcal{SF}[\![P_1]\!] \\ & \wedge (tr \upharpoonright B^{\checkmark}, X_2) \in \mathcal{SF}[\![P_2]\!] \\ & \wedge \sigma(tr) \subseteq (A \cup B)^{\checkmark} \} \end{aligned}$$

All of the laws for the parallel operator given in Figure 4.5, with the exception of $\|$-idempotence, also hold for the stable failures model.

EXAMPLE 6.4 The processes *PETE* and *DAVE* were introduced on Page 37. They both repeatedly and independently made a nondeterministic choice whether to lift a piano or a table.

$$\begin{array}{lcl} PETE & = & lift\_piano \rightarrow PETE \sqcap lift\_table \rightarrow PETE \\ DAVE & = & lift\_piano \rightarrow DAVE \sqcap lift\_table \rightarrow DAVE \end{array}$$

The process *DAVE* had exactly the same description.

Thus either of them can engage in any number of *lift_piano* and *lift_table* events, and then refuse either of them (but not both).

$$\begin{array}{lcll} \mathcal{SF}\,[\![PETE]\!] & = & \{(tr, X) \mid & tr \in \{lift\_piano, lift\_table\}^* \\ & & & \wedge\ \{lift\_piano, lift\_table\} \not\subseteq X\} \end{array}$$

and $\mathcal{SF}\,[\![DAVE]\!] = \mathcal{SF}\,[\![PETE]\!]$.

When these two processes are composed in parallel, then they must agree on the events that appear in the trace, but a refusal will be the union of refusals of the components. If $(tr, X_1) \in \mathcal{SF}\,[\![PETE]\!]$ and $(tr, X_2) \in \mathcal{SF}\,[\![DAVE]\!]$, then $(tr, X_1 \cup X_2) \in \mathcal{SF}\,[\![PETE \parallel DAVE]\!]$. The constraints that each of *PETE* and *DAVE* must be willing to perform one of their events is not reflected in their combination, which can refuse any events at all. The constraints that $\{lift\_piano, lift\_table\} \not\subseteq X_1$ and $\{lift\_piano, lift\_table\} \not\subseteq X_2$ are not strong enough to impose any constraints on $X_1 \cup X_2$.

$$\mathcal{SF}\,[\![PETE \parallel DAVE]\!] \quad = \quad \{(tr, X) \mid tr \in \{lift\_piano, lift\_table\}^*\}$$

Any trace is still possible, but deadlock at any stage is also possible. □

## Interleaving

An interleaving of two processes $P_1 \mid\mid\mid P_2$ executes each of them entirely independently of the other. Since they do not synchronize, an event (other than termination) will be refused by the combination only when it is refused by both processes independently—if one of the processes is ready to perform the event, then so is the combination. Termination requires the participation of both components, so it can be blocked by either. As in the traces model, traces of the combination appear as interleavings of traces of the two component processes.

$$\begin{array}{lcll} \mathcal{SF}\,[\![P_1 \mid\mid\mid P_2]\!] & = & \{(tr, X_1 \cup X_2) \mid \exists\, tr_1, tr_2 \bullet & tr \text{ interleaves } tr_1, tr_2 \\ & & & \wedge\ X_1 \upharpoonright \Sigma = X_2 \upharpoonright \Sigma \\ & & & \wedge\ (tr_1, X_1) \in \mathcal{SF}\,[\![P_1]\!] \\ & & & \wedge\ (tr_2, X_2) \in \mathcal{SF}\,[\![P_2]\!]\} \end{array}$$

The laws given in Figure 4.9 are all true for the stable failures model as well, with the exception of $|||$-zero. Although all (non-terminating) traces will be possible for $P \mathbin{|||} RUN_\Sigma$, it will not be stable unless $P$ is. Instability is introduced by including $DIV$ as another interleaved component, resulting in the process $RUN \mathbin{|||} DIV$ which serves as the zero for interleaving: it has all nonterminating traces, and no stable failures.

$$P \mathbin{|||} (RUN_\Sigma \mathbin{|||} DIV) =_{SF} (RUN_\Sigma \mathbin{|||} DIV) \qquad \langle |||_{SF}\text{-zero}\rangle$$

This law is also true in the traces model, since $RUN_\Sigma \mathbin{|||} DIV$ has the same traces as $RUN_\Sigma$.

## Interface parallel

The process $P_1 \underset{A}{\|} P_2$ is a combination of synchronous and interleaved parallel, synchronizing on events in the set $A^{\checkmark}$ and interleaving outside that set.

Any stable failure of the parallel process $P_1 \underset{A}{\|} P_2$ will be a combination of stable failures of its two components.

$$\begin{array}{rcl} \mathcal{SF}\,[\![P_1 \underset{A}{\|} P_2]\!] & = & \{(tr, X_1 \cup X_2) \mid \exists\, tr_1, tr_2 \bullet \\ & & \qquad tr \ \mathsf{synch}_A \ tr_1, tr_2) \\ & & \qquad \wedge\, X_1 \setminus A^{\checkmark} = X_2 \setminus A^{\checkmark} \\ & & \qquad \wedge\, (tr_1, X_1) \in \mathcal{SF}\,[\![P_1]\!] \\ & & \qquad \wedge\, (tr_2, X_2) \in \mathcal{SF}\,[\![P_2]\!]\} \end{array}$$

The laws for interface parallel given in Figure 4.10 all hold in the stable failures model with the exception of $\underset{A_T}{\|}$-zero which requires instability to be introduced to the zero for the same reason as the zero for interleaving:

$$P \underset{A}{\|} (RUN_{\Sigma \setminus A} \mathbin{|||} DIV) =_{SF} (RUN_{\Sigma \setminus A} \mathbin{|||} DIV) \qquad \langle \underset{A_{SF}}{\|}\text{-zero}\rangle$$

## Hiding

The process $P \setminus A$ will undergo the same executions as $P$, but events in the set $A$ will occur as internal events rather than as external synchronizations. This means that after any trace, a stable refusal $X$ of $P \setminus A$ will correspond to a stable refusal of $P$ in which not only internal

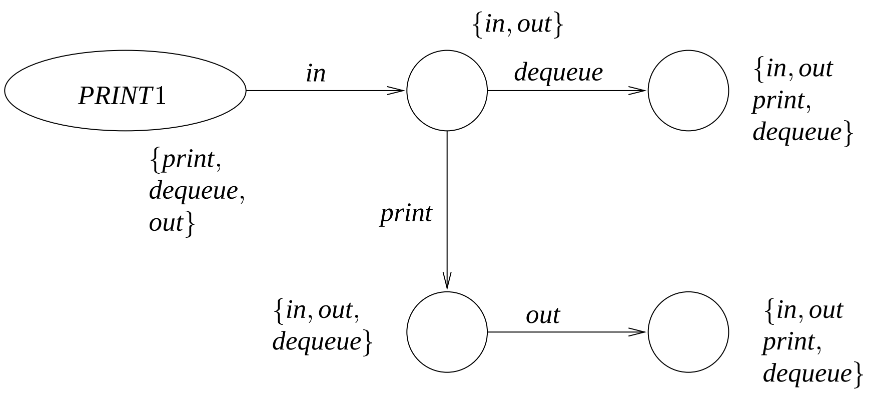

$\{in, out\}$
$PRINT1$
$in$
$dequeue$
$\{in, out$ $print,$ $dequeue\}$
$\{print,$ $dequeue,$ $out\}$
$print$
$\{in, out,$ $dequeue\}$
$out$
$\{in, out$ $print,$ $dequeue\}$

***Fig. 6.3*** Transition graph for *PRINT*1, labelled with maximal refusals

events of $P$ but also all events in $A$ (which have become internal events) are refused. The stable failures of $P \setminus A$ are therefore given by:

$$\mathcal{SF}\,[\![P \setminus A]\!] \;=\; \{(tr \setminus A, X) \mid (tr, X \cup A) \in \mathcal{SF}\,[\![P]\!]\}$$

EXAMPLE 6.5 A one-shot printer queue, a cut-down version of *PRINTQ* of Page 57, uses its channels as follows:

$$\begin{aligned} PRINT1 \;=\; in \rightarrow (\; & print \rightarrow out \rightarrow STOP \\ & \Box\; dequeue \rightarrow STOP) \end{aligned}$$

This has stable failures, illustrated in Figure 6.3, as follows:

$$\begin{aligned} \mathcal{SF}\,[\![PRINT1]\!] \;=\; & \{(\langle\rangle, X) \mid in \notin X\} \\ & \cup \{(\langle in\rangle, X) \mid \{print, dequeue\} \cap X = \{\}\} \\ & \cup \{(\langle in, print\rangle, X) \mid out \notin X\} \\ & \cup \{(\langle in, print, out\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\ & \cup \{(\langle in, dequeue\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \end{aligned}$$

The stable failures of $PRINT1 \setminus \{print\}$ derive from the stable failures of *PRINT*1 whose refusals that can be augmented with $\{print\}$. These are all failures apart from those with trace $\langle in\rangle$. The stable failures of $PRINT1 \setminus \{print\}$ are therefore derived as follows:

$$\begin{aligned} \mathcal{SF}\,[\![PRINT1 \setminus \{print\}]\!] \;=\; & \{(\langle\rangle, X) \mid in \notin X\} \\ & \cup \{(\langle in\rangle, X) \mid out \notin X\} \\ & \cup \{(\langle in, out\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\ & \cup \{(\langle in, dequeue\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \end{aligned}$$

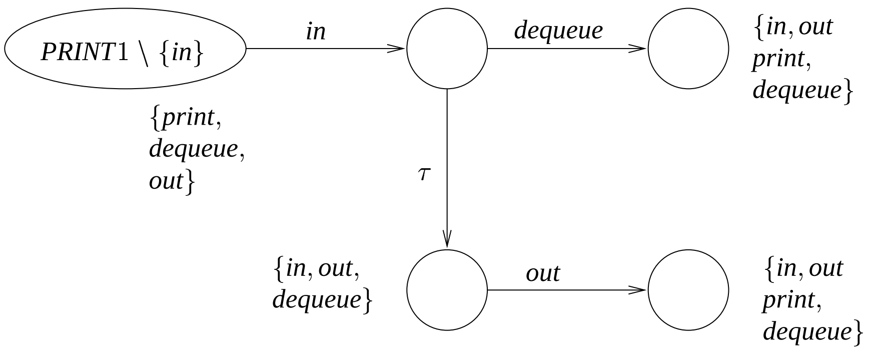

$PRINT1 \setminus \{in\}$
$in$
$dequeue$
$\{in, out$ $print,$ $dequeue\}$
$\{print,$ $dequeue,$ $out\}$
$\tau$
$\{in, out,$ $dequeue\}$
$out$
$\{in, out$ $print,$ $dequeue\}$

**Fig. 6.4** Transition graph for $PRINT1 \setminus \{in\}$, labelled with maximal refusals

These failures are illustrated in Figure 6.4. It emerges that *out* cannot be refused after *in*, but that *dequeue* can be.

□

## Renaming

The forward renamed process $f(P)$ behaves as $P$, except that $f(a)$ can be performed whenever $P$ could have performed $a$. It follows that the process $f(P)$ can refuse a set $X$ if every event that $f$ maps into $X$ can be refused by $P$, since if there is some event $a$ which $P$ cannot refuse, then $f(P)$ would have to be open to $f(a)$. This means that $f^{-1}(X)$ must be a refusal of $P$ whenever $X$ is a refusal of $f(P)$.

$$\mathcal{SF}\,[\![f(P)]\!] \quad = \quad \{(f(tr), X) \mid (tr, f^{-1}(X)) \in \mathcal{SF}\,[\![P]\!]\}$$

The renaming operator in the stable failures model meets all of the laws given in Figure 4.13.

The backward renaming operator $f^{-1}(P)$ also behaves in a similar fashion to $P$, but any event $a$ that is performed by $f^{-1}(P)$ corresponds to an event $f(a)$ performed by $P$. If a set $X$ is offered to the process $f^{-1}(P)$, then this corresponds to $f(X)$ being offered to the underlying process $P$. Hence $f^{-1}(P)$ can refuse $X$ whenever $P$ refuses $f(X)$.

$$\mathcal{SF}\,[\![f^{-1}(P)]\!] \quad = \quad \{(tr, X) \mid (f(tr), f(X)) \in \mathcal{SF}\,[\![P]\!])\}$$

All the laws given in Figure 4.13 for backward renaming also remain valid in the stable failures model.

## Sequential Composition

The sequential composition $P_1\,;\; P_2$ behaves as $P_1$ until $P_1$ terminates successfully, at which point it passes control to $P_2$. A stable failure of $P_1\,;\; P_2$ will arise either from a failure of $P_1$, which also refuses to terminate and transfer control to $P_2$, or else from a terminating trace of $P_1$ followed by a failure of $P_2$.

$$\begin{aligned}\mathcal{SF}[\![P_1;\ P_2]\!] \quad = \quad & \{(tr, X) \mid (tr, X \cup \{\checkmark\}) \in \mathcal{SF}[\![P_1]\!]\} \\ & \cup \{(tr_1 \frown tr_2, X) \mid (\ tr_1 \frown \langle\checkmark\rangle \in \mathsf{traces}(P_1) \\ & \qquad\qquad\qquad\qquad \wedge (tr_2, X) \in \mathcal{SF}[\![P_2]\!])\}\end{aligned}$$

Not all of the laws of sequential composition given in Figure 4.14 are valid in the stable failures model. In particular, $P;\ SKIP = P$ fails because of the possibility of termination in $P$ forming one branch of a choice. For example, the process $P = SKIP \ \Box\ a \rightarrow STOP$ is not able to refuse the event $a$, but $P;\ SKIP$ is able to refuse it by performing $P$'s termination event and resolving the choice. This example also demonstrates that Law $\Box$-; -dist is also not valid in the stable failures model. However, all of the other laws continue to hold.

## Interrupt

The process $P_1 \ \triangle\ P_2$ executes as $P_1$, but at any stage before termination it can begin executing as $P_2$. Any given stable failure $(tr, X)$ is either a stable failure of $P_1$ for which (if not terminating) $P_2$ is also able to refuse $X$ (since $P_2$ is still enabled); or else it is a non-terminating trace of $P_1$ followed by a failure of $P_2$, which must have a non-empty trace (since $P_2$ must perform an event to effect the interrupt).

$$\begin{aligned}\mathcal{SF}[\![P_1 \triangle P_2]\!] \quad = \quad & \{(tr, X) \mid \ (tr, X) \in \mathcal{SF}[\![P_1]\!] \\ & \qquad\qquad\quad \wedge (\checkmark \in \sigma(tr) \vee (\langle\rangle, X) \in \mathcal{SF}[\![P_2]\!])\} \\ & \cup \{(tr_1 \frown tr_2, X) \mid \ tr_1 \in \mathsf{traces}(P_1) \wedge \checkmark \notin \sigma(tr_1) \\ & \qquad\qquad\qquad\qquad \wedge (tr_2, X) \in \mathcal{SF}[\![P_2]\!] \\ & \qquad\qquad\qquad\qquad \wedge tr_2 \neq \langle\rangle\}\end{aligned}$$

All of the laws concerning the interrupt operator that are presented in Figure 4.15 are also true in the stable failures model.

EXAMPLE 6.6 A message authenticator will accept a message, and then either pass it on, or else reject it. It is unstable after its input. It can also be shutdown at any stage during its execution.

The one-message version is described as follows:

$$\begin{aligned}AUTH \quad = \quad & (left?x : T \rightarrow (DIV \ ||| \ \ (right!x \rightarrow STOP \quad ) \\ & \qquad\qquad\qquad\qquad\qquad\quad \sqcap\ reject \rightarrow STOP) \\ & \triangle\ shutdown \rightarrow STOP\end{aligned}$$

The stable failures of the process inside the interrupt are simply pairs of the form $(\langle\rangle, X)$ where $X \cap left.T = \{\}$. Once the first event has occurred, the process becomes unstable and contributes no further stable failures.

The calculation of the stable failures of *AUTH* requires consideration of the traces of the first process. For example, $(\langle left.3, reject, shutdown\rangle, \{right.3\})$ arises from the trace $\langle left.3, reject\rangle$ and the stable failure $(\langle shutdown\rangle, \{right.3\})$. Only sequential composition and interrupt require knowledge of the traces of their first component in order to derive their stable failures.

In fact, the stable failures of *AUTH* will be

$$
\begin{aligned}
&\{(\langle\rangle, X) \mid shutdown \notin X\}\\
&\cup\{(\langle left.x, shutdown\rangle, X) \mid x \in T\}\\
&\cup\{(\langle left.x, right.x, shutdown\rangle, X) \mid x \in T\}\\
&\cup\{(\langle left.x, reject, shutdown\rangle, X) \mid x \in T\}
\end{aligned}
$$

The refusal set in a stable failure is given by $shutdown \to STOP$ only when the trace from that component is not empty: *shutdown* must have occurred. □

## 6.3 RECURSION

A recursive definition $N = P$ defines the process $N$ in terms of a process description $P$ which may itself contain instances of $N$. The stable failures model provides guarantees that any such definition is sound: that any recursive equation has a solution. It also provides a way of determining the stable failures of the appropriate solution—the smallest possible such set of stable failures. This means that any solution to the recursive equation is guaranteed to have at least those stable failures as possible observations. The traces of the appropriate solution are given in the traces model.

EXAMPLE 6.7 The recursive equation $N = N \;\Box\; a \to STOP$ has many fixed points, including $a \to STOP$, $a \to STOP \;\Box\; b \to STOP$, and $a \to STOP \;\Box\; DIV$. The least of these in the stable failures model is $a \to STOP \;\Box\; DIV$, and so this will be the semantics of the process defined by the recursive equation. □

### Operational semantics

The understanding of recursion in the Stable Failures model requires a slightly different operational treatment of recursive unwinding than was presented in Chapter 1, in order to give a satisfactory account of divergence. In particular, unguarded recursions such as the one in Example 6.7 above are considered to be unstable because an infinite sequence of recursive invocations of the process $N$ of Example 6.7 may occur without the occurrence of any external events. Beginning with the process $N$, the process $N \;\Box\; a \to STOP$ is reached from a recursive invocation, and then $N \;\Box\; a \to STOP \;\Box\; a \to STOP$, and so on. To consider this as a divergent sequence, an internal event is associated with a recursive unwinding, resulting in the following rule for recursion in place of the original transition rule given on Page 12.

$$\boxed{\frac{}{N \xrightarrow{\tau} P}\quad [\, N = P \,]}$$

When the process expression $P$ is guarded in $N$, then this initial internal action makes no difference to the visible behaviour of $P$ as compared with the original transition rule for recursive processes: both rules will give rise to the same traces and stable failures. In fact, the traces will be the same for all guarded and unguarded recursive process definitions, and all the results concerning the traces model remain valid if this rule for recursion is used instead. The only difference between the impact of the two rules is on the stability of unguarded recursions.

EXAMPLE 6.8 Concerning the process $N = N \,\Box\, a \to STOP$, the revised rule for recursive unwinding allows the sequence of transitions:

$$\begin{array}{l} N \\ \quad\downarrow \tau \\ N \,\Box\, a \to STOP \\ \quad\downarrow \tau \\ N \,\Box\, a \to STOP \,\Box\, a \to STOP \\ \quad\downarrow \tau \\ \quad\vdots \end{array}$$

The original rule for recursion had no internal transitions for $N$, and only one transition, labelled by $a$, to $STOP$. □

EXAMPLE 6.9 The process $N = STOP \sqcap b \to N$ takes an internal transition to unwind the definition, and then a further transition to resolve the internal choice. Finally, it has either reached $STOP$ or else the stable process $b \to N$. The same possibilities arise if the original transition rule for recursion is used, except that the initial internal transition is absent. The two transition graphs are compared in Figure 6.5. They are each associated with the same traces and stable failures. □

All of the techniques for recursion introduced in Chapter 4 for the traces model are also applicable in the stable failures model.

The traces and stable failures associated with recursively defined process expressions $N = P$ can be obtained directly from the operational semantics, or alternatively by using the denotational semantics. Both of these approaches give the same result.

The process $P$ with free variable $N$ corresponds to a function $F(Y) = P[Y/N]$, and successive applications of the function $F$ will give rise to approximations to the fixed point. The

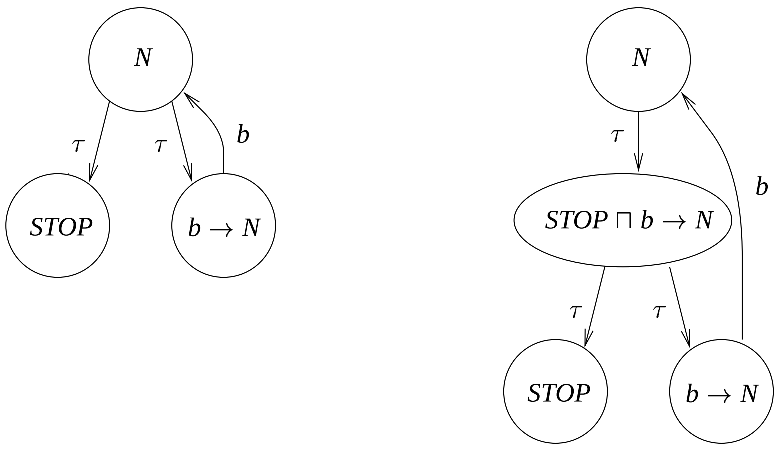


***Fig. 6.5*** Two transition graphs for $N = STOP \sqcap a \rightarrow N$

first approximation is the minimal process $DIV$, and successive approximations are $F(DIV)$, $F(F(DIV))$ and then $F^n(DIV)$ for $n \in \mathbb{N}$. The sequence of approximations $\langle F^n(DIV)\rangle_{n\in\mathbb{N}}$ will define the fixed point, which will consist of those traces and stable failures that appear in some elements of the sequence. The traces of $N$ are those given in the traces model. The stable failures will then be

$$\bigcup_{n\in\mathbb{N}} \mathcal{SF}\,[\![F^n(DIV)]\!]$$

This process is the minimal one which contains all of the approximations.

EXAMPLE 6.10 The process $N = STOP \sqcap b \rightarrow N$ is the fixed point of the function $F(Y) = STOP \sqcap b \rightarrow Y$. For any $n$, the semantics of $F^{n+1}(DIV)$ can be calculated from the semantics of $F^n(DIV)$, resulting in

$$\mathcal{SF}\,[\![F^n(DIV)]\!] \quad = \quad \{(\langle b\rangle^i, X) \mid i < n \wedge X \subseteq \Sigma^{\checkmark}\}$$

The union of these approximations yields

$$\mathcal{SF}\,[\![N]\!] \quad = \quad \{(\langle b\rangle^i, X) \mid i \in \mathbb{N} \wedge X \subseteq \Sigma^{\checkmark}\}$$

which is in accordance with the behaviours predicted from the operational semantics. □

Law recursion-unwinding of Page 118 will hold for any recursive definition $N = P$. The law UFP also holds: all solutions to any guarded equation must have the same stable failures.

$$(F(Y) \text{ guarded} \wedge (F(P_1) =_{SF} P_1) \wedge (F(P_2) =_{SF} P_2)) \Rightarrow P_1 =_{SF} P_2 \qquad \langle \mathsf{UFP}_{SF} \rangle$$

For example, the function $F$ of Example 4.24 is event guarded:

$$\begin{array}{rcl} N & = & F(N) \quad = \quad (a \to N) \,\Box\, b \to STOP \\ M & = & a \to M \\ P & = & M \,\triangle\, (b \to STOP) \end{array}$$

Furthermore $P =_{SF} F(P)$, and $N =_{SF} F(N)$ by definition, so it follows that $N =_{SF} P$.

### Mutual recursion

Mutual recursion is a generalization of single recursion, with an appropriate generalized treatment. The operational transition rule is adjusted in a similar way, modeling the recursive unwinding of any process variable $N_i$ as accompanied by an internal transition. As with the case for single recursion, exactly the same results concerning the traces model remain valid if this transition rule is used instead.

$$\frac{}{N_i \xrightarrow{\tau} P_i} \quad [\, \underline{N} = \underline{P} \,]$$

The stable failures associated with all of the $N_i$ processes will be those that are predicted by the operational semantics. They will give the minimal processes that satisfy the set of defining equations—the ones with the fewest stable failures. The underlying theory of CSP guarantees that such minimal processes must exist for any set of recursive CSP definitions.

The results concerning single recursion carry over to the more general case. The semantics of the $N_i$ are the unions of the semantics of the chain of approximations, starting from $DIV$. Each $N_i$ is defined by a function $F_i(\underline{N})$. If the $j$th approximation to $N_i$ is written as $N_i^j$, then each $N_i^0 = DIV$, and each $N_i^{j+1} = F_i(\underline{N}^j)$, where $\underline{N}^j$ is the vector of all of the $j$th approximations. Each approximation $N_i^j$ is associated with a set of stable failures $\mathcal{SF}\,[\![N_i^j]\!]$. Each limit $N_i$ will have stable failures given by

$$\mathcal{SF}\,[\![N_i]\!] \quad = \quad \bigcup_{j \in \mathbb{N}} \mathcal{SF}\,[\![N_i^j]\!]$$

Law recursion-unwinding will hold for any family of mutually recursive definitions. Whenever $N_i = P_i$ appears as a recursive definition, then $N_i =_{SF} P_i$.

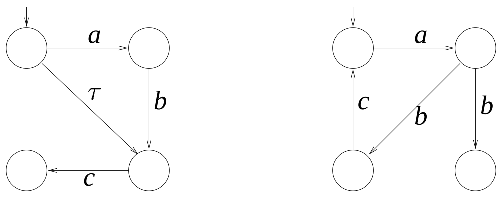
Law UFP also generalizes to mutual recursion. In a mutually recursive definition $\underline{N} = \underline{P}$, a process variable $N_i$ is recursive if it appears in any of the $P_j$. If each process definition $P_i$ associated with any recursive $N_i$ is event guarded in all of the process variables that appear in it, then the recursive definition is event guarded. If two families of processes both satisfy the same guarded recursive equation, then they must be equivalent:

$$\boxed{(\underline{F}(\underline{Y}) \text{ guarded} \wedge (\underline{F}(\underline{P_1}) =_{SF} \underline{P_1}) \wedge (\underline{F}(\underline{P_2}) =_{SF} \underline{P_2})) \Rightarrow \underline{P_1} =_{SF} \underline{P_2}}$$

As in the traces model, a family of process definitions may be rewritten using Law recursion-unwinding to equivalent processes whose definitions are in a form more suitable for further reasoning.

## Exercises

EXERCISE 6.1 What are the stable failures associated with the following state machines ?



EXERCISE 6.2 Give the stable failures of the following processes:

1. $a \rightarrow STOP \mid b \rightarrow c \rightarrow STOP$
2. $a \rightarrow STOP \mid b \rightarrow (c \rightarrow STOP \mid d \rightarrow STOP)$
3. $a \rightarrow STOP \; \Box \; a \rightarrow b \rightarrow STOP$

EXERCISE 6.3 What are the stable failures of the following non-recursive processes:

1. $(coin \rightarrow tea \rightarrow STOP) \; \Box \; (coin \rightarrow coffee \rightarrow STOP)$
2. $(tea \rightarrow STOP) \sqcap (coffee \rightarrow STOP)$
3. $(coin \rightarrow tea \rightarrow STOP) \parallel (coin \rightarrow coffee \rightarrow STOP)$
4. $(coin \rightarrow tea \rightarrow STOP) \mid\mid\mid (coin \rightarrow coffee \rightarrow STOP)$
5. $(coin \rightarrow ((tea \rightarrow STOP) \; \Box \; (coffee \rightarrow STOP))) \setminus \{tea\}$

EXERCISE 6.4 What are the stable failures of the following recursive processes:

1. $VM1 = (coin \to (VM1 \sqcap choc \to VM1))$
2. $VM2 = (coin \to (VM2 \square choc \to VM2))$
3. $VM3 = VM3 \square choc \to STOP$

EXERCISE 6.5 Give a process $P$ for which $P \parallel P \neq_{SF} P$.

Is $P \parallel P =_{SF} P \parallel P \parallel P$ a law of the stable failures model ?

EXERCISE 6.6
What is the behaviour of the following processes:

1. $RUN \parallel CHAOS$
2. $RUN \mid\mid\mid CHAOS$
3. $RUN_A \parallel CHAOS$
4. $RUN_A \mid\mid\mid RUN_B$
5. $RUN_A \square CHAOS$
6. $RUN_A \sqcap CHAOS$

EXERCISE 6.7 Give a single operational rule for *DIV* which is consistent with the stable failures semantics.

EXERCISE 6.8 Give operational rules for *CHAOS* which are consistent with the stable failures semantics.

EXERCISE 6.9 Does the law $\langle$□-⊓-dist$\rangle$ on Page 180 hold in the traces model ?

# 7 Specification and verification with failures

## 7.1 PROPERTY-ORIENTED SPECIFICATION

The introduction of failures information in the stable failures model allows a wider range of specification than was possible in the traces model. Specifications on behaviours describe those executions that are acceptable, and a verification of a system or process $P$ requires an argument to establish that no behaviour of $P$ violates such a specification. Since there are now two sets of behaviours associated with any process—traces, and stable failures—a specification will consist of two parts, each of which describe the required property of observations from the corresponding behaviour set. A specification $S$ can be written as a pair $(S_T(tr), S_{SF}(tr, X))$. Each of the predicates $S_T$ and $S_{SF}$ can be expressed in any notation, though in common with specifications in the traces model first order logic and elementary set and sequence notation tend to be sufficient in practice.

$$
\begin{array}{rcl}
P \ \mathbf{sat} \ (S_T(tr), S_{SF}(tr, X)) & = & \forall\, tr \in \mathsf{traces}(P) \bullet S_T(tr) \\
 & & \wedge\ \forall (tr, X) \in \mathcal{SF}\,[\![P]\!] \bullet S_{SF}(tr, X)
\end{array}
$$

Safety specifications, that 'nothing bad will happen', are requirements on traces, where 'nothing bad' means that no event will occur at an inappropriate point. Safety requirements are captured in this model by using the predicate $S_T$ to constrain the traces that are permitted.

The stable failures model also contains sufficient detail to support the expression of liveness specifications, which require that 'something good will happen'. Within the context of synchronizing concurrent systems, liveness is expressed in terms of a process' willingness to participate in events. This will mean that at particular points of an execution, the process should be guaranteed to offer certain events: any stable state reached by the process should not refuse those events. These conditions are precisely what is expressed by the requirement that certain events should not appear in the refusal set $X$. If the process does not diverge, then it should be guaranteed to reach a stable state where the events are offered.

For example, a stable component that must always be ready for input should meet the specification that input can never be refused: $S_{SF}(tr, X) = in.T \cap X = \{\}$. Whatever trace has occurred previously, the process can never refuse input.

EXAMPLE 7.1 (RAILWAY CROSSING) The process *CROSS*, defined in Example 1.16 on Page 13, raises and lowers a barrier, and records when trains enter and leave the crossing. As well as its safety requirements, it should also meet the liveness requirement that it is ready to lower the gate whenever the gate is up and an approaching train is detected. This is a conditional liveness property, requiring an offer of a particular event only under certain conditions on the trace:

$$S_{SF}(tr, X) \quad = \quad tr = tr' \frown \langle gate.raise, train.approach \rangle \Rightarrow gate.lower \notin X$$

□

EXAMPLE 7.2 (BUFFERS) A common specification is that of a *buffer* or FIFO queue. The safety requirements on a buffer have already been discussed in the previous chapter, but a buffer must also have some liveness requirements: that it must be ready for input when it is empty, and that it must be ready for output when it is non-empty. The specification of a buffer of type $T$ may be expressed as a predicate on traces and on stable failures:

$$\begin{aligned} Buff_T(tr) &= tr \Downarrow out \leqslant tr \Downarrow in \\ Buff_{SF}(tr, X) &= tr \Downarrow out = tr \Downarrow in \Rightarrow in.T \cap X = \{\} \\ &\quad \wedge tr \Downarrow out < tr \Downarrow in \Rightarrow out.T \nsubseteq X \end{aligned}$$

The safety specification, expressed on traces, states that the sequence of outputs must match the sequence of inputs, appearing in the same order. If the sequence of inputs is equal to the sequence of outputs, then the buffer must be empty, and the liveness requirement states that no input may be refused. If the sequence of outputs does not contain all input messages, then the buffer is non-empty, and so not all outputs can be refused. The safety specification allows only one output to be possible, so any output which is not the next element of the sequence can be refused in a stable state.

The specification states nothing about the capacity of the buffer, or even whether the capacity is fixed, or finite or infinite. It also allows events along other channels, since it places no restrictions on the behaviour of the process with regard to other events. However, the specification is conventionally used to describe processes which have only input and output channels: $\sigma(P) \subseteq in.T \cup out.T$. This can be introduced into the specification, as another safety specification:

$$Buff'_T(tr) \quad = \quad Buff_T(tr) \wedge \sigma(tr) \subseteq in.T \cup out.T$$

The specification also implies that a buffer cannot terminate. It requires liveness after any trace, and terminating traces would not be exempt. □

## 7.2 VERIFICATION

The semantic equations associated with the CSP operators support a number of proof rules for reasoning about CSP process descriptions. Proof obligations are of the form $P$ **sat** $(S_T(tr), S_{SF}(tr, X))$. These can be split into a two separate obligations:

1. a traces obligation $P$ **sat** $S_T(tr)$ which can be addressed with the proof system for the traces model presented in Chapter 5;
2. a stable failures obligation $P$ **sat** $S_{SF}(tr, X)$, which states that all the stable failures of $P$ meet predicate $S_{SF}$. Requirements of this form are the concern of this chapter.

This section will present a set of compositional proof rules for establishing stable failures specifications for processes will be presented. Two of these rules (sequential composition, and interrupt) rely on trace specifications of their component processes, reflecting the fact that the definitions of the stable failures of these processes refer to the traces of their components.

### *STOP*

There is only one trace of the process *STOP*: the empty trace. It may be accompanied by any refusal set, so there is no restriction on the refusal $X$. The constraint on any stable failure is simply that its trace is empty.

$$\frac{}{STOP \ \mathbf{sat}\ tr = \langle\rangle}$$

The rule has no antecedents, corresponding to the fact that *STOP* has no component processes.

### Prefix

A failure of the process $a \to P$ either has an empty trace, in which case $a$ cannot be refused, or else begins with the event $a$ followed by a failure of $P$. If $P$ **sat** $S_{SF}(tr, X)$ then the part of the trace after $a$ (that is: $tail(tr)$) together with the refusal $X$ must meet the specification $S_{SF}$.

$$\frac{P \ \mathbf{sat}\ S_{SF}(tr, X)}{a \to P \ \mathbf{sat}\ \begin{array}{l} tr = \langle\rangle \wedge a \notin X \\ \vee \\ head(tr) = a \wedge S_{SF}(tail(tr), X) \end{array}}$$

### Prefix Choice

The prefix choice operator generalizes the prefix operator: it contains a number of component processes, and the first event that is performed can be any one of the menu of events offered.

The antecedent to the rule assumes a family of specifications $S_a(tr, X)$, one for each of the components $P(a)$.

$$\frac{\forall\, a \in A \bullet P(a) \ \mathbf{sat}\ S_a(tr, X)}{\begin{array}{ll} x : A \to P(x) \ \mathbf{sat} & tr = \langle\rangle \wedge A \cap X = \{\} \\ & \vee \\ & \exists\, a \in A \bullet head(tr) = a \wedge S_a(tail(tr), X) \end{array}}$$

### Output and Input

The output process $c!v \to P$ is simply a particular kind of prefix process, and the proof rule reflects this:

$$\frac{P \ \mathbf{sat}\ S_{SF}(tr, X)}{\begin{array}{ll} c!v \to P \ \mathbf{sat} & tr = \langle\rangle \wedge c.v \notin X \\ & \vee \\ & head(tr) = c.v \wedge S_{SF}(tail(tr), X) \end{array}}$$

Similarly, the input process $c?x : T \to P(x)$ is a special form of prefix choice, and so the proof rule is very similar:

$$\frac{\forall\, v \in T \bullet P(v) \ \mathbf{sat}\ S_v(tr, X)}{\begin{array}{ll} c?x : T \to P(x) \ \mathbf{sat} & tr = \langle\rangle \wedge in.T \cap X = \{\} \\ & \vee \\ & \exists\, v \in T \bullet head(tr) = c.v \wedge S_v(tail(tr), X) \end{array}}$$

### *SKIP*

The process *SKIP* does nothing except terminate successfully. It has only two possible stable failures, one for before termination, and one for after.

$$\frac{}{SKIP \ \mathbf{sat}\ (tr = \langle\rangle \wedge \checkmark \notin X) \vee tr = \langle \checkmark \rangle}$$

The refusal set is hardly constrained, apart from the requirement that termination should not be refused before it occurs.

### *DIV*

The process *DIV* has no stable failures at all, so there is no specification that it can violate. It therefore vacuously meets any specification $S$ (even *false*). $false(tr, X)$.

$$\frac{}{DIV \ \mathbf{sat}\ S(tr, X)}$$

This apparently miraculous behaviour of *DIV*—that it can meet any specification—indicates that there is some aspect of the behaviour of *DIV* that is not considered in the stable failures model. The fact that it is divergent exempts it from any need to be concerned with stable failures.

## *CHAOS*

The worst process, *CHAOS*, is able to perform or refuse anything. It will only meet the trivial specification $true(tr, X)$, the weakest specification.

$$\frac{}{CHAOS \ \mathbf{sat} \ true(tr, X)}$$

## *RUN*

The process *RUN* is able to do any trace, but unlike *CHAOS* it is unable to refuse any event before termination.

$$\frac{}{RUN \ \mathbf{sat} \ \checkmark \notin \sigma(tr) \Rightarrow X = \{\}}$$

The specification met by *RUN* imposes no restrictions on the traces that it can perform, only on the refusals that may accompany those traces.

### External Choice

The process $P_1 \ \Box \ P_2$ behaves either as $P_1$ or as $P_2$. If $P_1$ **sat** $S_1(tr, X)$ and $P_2$ **sat** $S_2(tr, X)$ then the choice process $P_1 \ \Box \ P_2$ satisfies the disjunction of these two specifications, and their conjunction when the trace is empty:

$$\frac{\begin{array}{l} P_1 \ \mathbf{sat} \ S_1(tr, X) \\ P_2 \ \mathbf{sat} \ S_2(tr, X) \end{array}}{\begin{array}{ll} P_1 \ \Box \ P_2 \ \mathbf{sat} & (tr = \langle\rangle \Rightarrow S_1(tr, X) \wedge S_2(tr, X)) \\ & \wedge \ (tr \neq \langle\rangle \Rightarrow (S_1(tr, X) \vee S_2(tr, X)) \end{array}}$$

Any refusal of $P_1 \ \Box \ P_2$ before any event has yet been performed must be a refusal of both components.

The rule generalizes to indexed external choices:

$$\frac{\forall \, i \in I \bullet P_i \ \mathbf{sat} \ S_i(tr, X)}{\begin{array}{ll} \Box_{i \in I} P_i \ \mathbf{sat} & tr = \langle\rangle \Rightarrow \bigwedge_{i \in I} S_i(tr, X) \\ & \wedge \ tr \neq \langle\rangle \Rightarrow \bigvee_{i \in I} S_i(tr, X) \end{array}}$$

Any events refused before the choice has been made must be refusable by all of the components. After the choice has been made, the refusal set is the responsibility of the process in whose favour the choice was resolved.

## Internal choice

The internal choice operator can behave as either of its components:

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S_1(tr, X) \\ P_2 \textbf{ sat } S_2(tr, X) \end{array}}{P_1 \sqcap P_2 \textbf{ sat } S_1(tr, X) \vee S_2(tr, X)}$$

The indexed internal choice can behave as any of its components:

$$\frac{\forall\, i \in J \bullet P_i \textbf{ sat } S_i(tr, X)}{\bigsqcap_{i \in J} P_i \textbf{ sat } \exists\, i \in J \bullet S_i(tr, X)}$$

## Parallel composition

A failure $(tr, X)$ of the process $P_1 \;{}_{A_1}\|_{A_2}\; P_2$ is comprised of a contribution from $P_1$ and a contribution from $P_2$, contained within the alphabets $A_1^{\checkmark}$ and $A_2^{\checkmark}$ respectively. In fact, the projection $tr \upharpoonright A_1^{\checkmark}$ is a trace of $P_1$, and the projection $tr \upharpoonright A_2^{\checkmark}$ is a trace of $P_2$. The refusal set $X$ is a made up of $X_1$ and $X_2$ from $P_1$ and $P_2$ respectively.

$$\frac{\begin{array}{l} P_1 \textbf{ sat } S_1(tr, X) \\ P_2 \textbf{ sat } S_2(tr, X) \end{array}}{\begin{array}{ll} P_1 \;{}_{A_1}\|_{A_2}\; P_2 \textbf{ sat } \exists\, X_1, X_2 \bullet & S_1(tr \upharpoonright A_1^{\checkmark}, X_1) \wedge S_2(tr \upharpoonright A_2^{\checkmark}, X_2) \\ & \wedge\, \sigma(tr) \subseteq (A_1 \cup A_2)^{\checkmark} \\ & \wedge\, X \cap (A_1 \cup A_2)^{\checkmark} = (X_1 \cap A_1^{\checkmark}) \cup (X_2 \cap A_2^{\checkmark}) \end{array}}$$

For instance, two processes might both meet an initial liveness specification on the event $a$, that $a$ must be available until it is performed, as follows:

$$\begin{array}{lll} P_1 & \textbf{sat} & S_1(tr, X) = a \notin \sigma(tr) \Rightarrow a \notin X \\ P_2 & \textbf{sat} & S_2(tr, X) = a \notin \sigma(tr) \Rightarrow a \notin X \end{array}$$

Each of them meets the specification that if $a$ has not yet occurred, then it cannot be refused.

The rule yields that the combination $P_1 \;{}_{\{a,b\}}\|_{\{a,c\}}\; P_2$ meets the specification

$$\begin{array}{ll} \exists\, X_1, X_2 \bullet & tr \upharpoonright \{a, b\}^{\checkmark} = \langle\rangle \Rightarrow a \notin X_1 \\ & tr \upharpoonright \{a, c\}^{\checkmark} = \langle\rangle \Rightarrow a \notin X_2 \\ & \wedge\, \sigma(tr) \subseteq \{a, b, c\}^{\checkmark} \\ & \wedge\, X \upharpoonright \{a, b, c\}^{\checkmark} = (X_1 \cap \{a, b\}^{\checkmark}) \cup (X_2 \cap \{a, c\}^{\checkmark}) \end{array}$$

which implies that $a \notin \sigma(tr) \Rightarrow a \notin X$, and so

$$P_1 \;{}_{\{a,b\}}\|_{\{a,c\}}\; P_2 \quad \textbf{sat} \quad a \notin \sigma(tr) \Rightarrow a \notin X$$

If both components are initially live on the event $a$, then so is their parallel combination.

The rule for the indexed parallel operator follows a similar pattern. Each component $P_i$ with interface $A_i$ imposes its own constraint $S_i(tr, X)$ on the projection of the overall behaviour onto the alphabet $A_i^{\checkmark}$.

$$\frac{\forall\, i \in I \bullet P_i \;\textbf{sat}\; S_i(tr, X)}{\begin{array}{ll} \|_{A_i} P_i \;\textbf{sat}\; \exists \langle X_i \rangle \bullet & \forall\, i \in I \bullet S_i(tr \upharpoonright A_i^{\checkmark}, X_i) \\ & \wedge\, X \cap (\bigcup_i A_i)^{\checkmark} = \bigcup_i (X_i \cap A_i^{\checkmark}) \\ & \wedge\, \sigma(tr) \subseteq (\bigcup_{i \in I} A_i)^{\checkmark} \end{array}}$$

## Interleaving

An interleaved combination $P_1 \;|||\; P_2$ performs traces $tr$ which consist of a trace $tr_1$ of $P_1$ interleaved with a trace $tr_2$ of $P_2$. A refusal after such a trace must be a refusal of both processes.

$$\frac{\begin{array}{l} P_1 \;\textbf{sat}\; S_1(tr, X) \\ P_2 \;\textbf{sat}\; S_2(tr, X) \end{array}}{\begin{array}{ll} P_1 \;|||\; P_2 \;\textbf{sat}\; \exists\, tr_1, tr_2, X_1, X_2 \bullet & (S_1(tr_1, X_1) \wedge S_2(tr_2, X_2) \wedge tr \;\textsf{interleaves}\; tr_1, tr_2) \\ & \wedge\, X_1 \cup X_2 = X \wedge X_1 \setminus \{\checkmark\} = X_2 \setminus \{\checkmark\} \end{array}}$$

For instance, consider a process $P_1$ which meets the specification given previously that the process must initially be live on $a \in \Sigma$:

$$P_1 \quad \textbf{sat} \quad a \notin \sigma(tr) \Rightarrow a \notin X$$

Then $P_1$ interleaved with any (non-divergent) process $P_2$ at all will still meet this specification. Firstly any such process has $P_2$ **sat** $true(tr, X)$, so the rule for interleaving yields that

$$\begin{array}{lll} P_1 \;|||\; P_2 & \textbf{sat} & \exists\, tr_1, tr_2, X_1, X_2 \bullet \;\; a \notin \sigma(tr_1) \Rightarrow a \notin X_1 \wedge tr \;\textsf{interleaves}\; tr_1, tr_2 \\ & & \qquad\qquad\qquad\qquad\; \wedge\, X_1 \cup X_2 = X \wedge X_1 \setminus \{\checkmark\} = X_2 \setminus \{\checkmark\} \end{array}$$

Furthermore, if $tr$ interleaves $tr_1, tr_2$ then $\sigma(tr) = \sigma(tr_1) \cup \sigma(tr_2)$, so $a \notin \sigma(tr) \Rightarrow a \notin \sigma(tr_1)$. Also, $a \notin X_1 \Rightarrow a \notin X$. Thus

$$P_1 \;|||\; P_2 \quad \textbf{sat} \quad a \notin \sigma(tr) \Rightarrow a \notin X$$

A single component of an interleaved combination can ensure liveness.

## Interface parallel

A failure $(tr, X)$ of $P_1 \underset{A}{\|} P_2$ must arise from two failures $(tr_1, X_1)$ and $(tr_2, X_2)$ of $P_1$ and $P_2$ respectively, where $tr\ \mathsf{synch}_A\ tr_1, tr_2$, and $X_1$ and $X_2$ coincide on events $P_1$ and $P_2$ can perform independently—those outside $A^{\checkmark}$. This results in the following inference rule:

$$\frac{\begin{array}{l} P_1\ \mathbf{sat}\ S_1(tr, X) \\ P_2\ \mathbf{sat}\ S_2(tr, X) \end{array}}{\begin{array}{ll} P_1 \underset{A}{\|} P_2\ \mathbf{sat} & \exists\, tr_1, tr_2, X_1, X_2 \bullet \\ & \quad (S_1(tr_1, X_1) \wedge S_2(tr_2, X_2) \wedge tr\ \mathsf{synch}_A\ tr_1, tr_2 \\ & \quad \wedge\ X_1 \cup X_2 = X \\ & \quad \wedge\ X_1 \setminus A = X_2 \setminus A) \end{array}}$$

## Hiding

A trace of the process $P \setminus A$ arises from a trace of $P$ simply by removing all of the events in $A$ from the trace. Hence for any trace of $P \setminus A$ with refusal set $X$ there is a corresponding trace of $P$ with refusal set $X \cup A$. The rule is then as follows:

$$\frac{P\ \mathbf{sat}\ S(tr, X)}{P \setminus A\ \mathbf{sat}\ \exists\, tr_1 \bullet (S(tr_1, X \cup A) \wedge tr = tr_1 \setminus A)}$$

EXAMPLE 7.3 Consider the process $P$ given by $P = a \to b \to c \to P$. The proof rule will be used to establish the liveness specification on $P \setminus \{b\}$ that $c$ should be available whenever $a$ is the last event to have occurred:

$$P \setminus \{b\} \quad \mathbf{sat} \quad foot(tr) = a \Rightarrow c \notin X$$

A property that $P$ satisfies is

$$S(tr, X) \quad = \quad (foot(tr) = a \Rightarrow b \notin X) \wedge (foot(tr) = b \Rightarrow c \notin X)$$

If $(tr, X)$ is a stable failure of $P \setminus \{b\}$, then $\exists\, tr_1 \bullet S(tr_1, X \cup \{b\}) \wedge tr = tr_1 \setminus \{b\}$. If $foot(tr) = a$, then $foot(tr_1 \setminus \{b\}) = a$, so either $foot(tr_1) = a$ or $foot(tr_1) = b$. The first of these contradicts $S(tr_1, X \cup \{b\})$, since it implies that $b \notin X \cup \{b\}$; and the second implies that $c \notin X \cup \{b\}$, which in turn implies that $c \notin X$. The specification can thus be weakened to obtain

$$P \setminus \{b\} \quad \mathbf{sat} \quad foot(tr) = a \Rightarrow c \notin X$$

This is the specification required. □

In fact, the inference rule simplifies in the case where the specification $S_{SF}(tr, X)$ is *independent* of the set $A$ being hidden. A failures specification is $A$-independent if $\forall\, tr, X \bullet (S_{SF}(tr, X) \Leftrightarrow S_{SF}(tr \setminus A, X \setminus A))$. For such a specification, the predicate $\exists\, tr_1 \bullet S_{SF}(tr_1, X \cup A) \wedge tr_1 \setminus A = tr$ is equivalent to $S_{SF}(tr, X)$. The resulting rule is

$$\frac{P \textbf{ sat } S_{SF}(tr, X)}{P \setminus A \textbf{ sat } S_{SF}(tr, X)} \quad [\, S_{SF}(tr, X) \text{ is } A\text{-independent}\,]$$

This rule states that if a process $P$ meets a specification $S_{SF}(tr, X)$ independently of the performance or refusal of any events in $A$, then $P \setminus A$ also meets it. Both the specification itself and $P$'s meeting of it are completely independent of its behaviour on $A$.

Observe that the earlier specification $foot(tr) = a \Rightarrow c \notin X$ is not $\{b\}$-independent even though $b$ does not appear anywhere explicitly in the specification, since in this case $foot(tr)$ is not the same as $foot(tr \setminus \{b\})$.

On the other hand, a specification that is $\{b\}$-independent is the liveness requirement that whenever the same number of $a$'s and $c$'s have been performed, then $a$ should be on offer:

$$tr \downarrow \{a\} = tr \downarrow \{c\} \Rightarrow a \notin X$$

The process $P = a \rightarrow b \rightarrow c \rightarrow P$ meets this specification, and so the derived inference rule for hiding yields that $P \setminus \{b\}$ also satisfies it.

## Renaming

A failure $(tr, X)$ of a renamed process $f(P)$ will be a renamed failure $(f(tr_1), X)$ for some $tr_1$ for which $(tr_1, f^{-1}(X))$ is a failure of $P$. The inference rule for translating specifications through a forward renaming is as follows:

$$\frac{P \textbf{ sat } S_{SF}(tr, X)}{f(P) \textbf{ sat } \exists\, tr_1 \bullet S_{SF}(tr_1, f^{-1}(X)) \wedge f(tr_1) = tr}$$

EXAMPLE 7.4 In Example 3.14 a process *OFFICE* models two staff who each answer their own phones. When both phones are mapped to the same number—$f(phone.sylvie) = f(phone.janet) = phone.janet\&sylvie$—then the best guarantee that can be provided is that someone will answer, but with no guarantees as to who it will be.

The specification used to obtain this result through the inference rule is

$$\begin{aligned} OFFICE \quad \textbf{sat} \quad & foot(tr) = phone.janet \vee foot(tr) = phone.sylvie \\ & \qquad \Rightarrow \{answer.janet, answer.sylvie\} \not\subseteq X \end{aligned}$$

The direct result of applying the inference rule with this antecedent yields the result

$$\begin{aligned} f(OFFICE) \quad \textbf{sat} \quad \exists\, tr_1 \bullet \; & (foot(tr_1) = phone.janet \vee foot(tr_1) = phone.sylvie \\ & \quad \Rightarrow \{answer.janet, answer.sylvie\} \not\subseteq f^{-1}(X)) \\ & \wedge f(tr_1) = tr \end{aligned}$$

which is equivalent to the result required:

$$\begin{aligned} f(OFFICE) \quad \textbf{sat} \quad & foot(tr) = phone.janet\&sylvie \\ & \Rightarrow \{answer.janet, answer.sylvie\} \not\subseteq X \end{aligned}$$

Observe that the specification on *OFFICE* covered all of the possible ways in which $foot(tr) = phone.janet\&sylvie$ can arise in $f(OFFICE)$, and showed that in each case the required result followed. In order to establish that $f(P)$ **sat** $R_{SF}(tr, X)$ from the fact that $P$ **sat** $S_{SF}(tr, X)$, it is necessary for all behaviours that meet $S_{SF}$ to satisfy $R_{SF}$ when mapped through the alphabet renaming. Another form of this rule is this:

$$\frac{\begin{array}{l} P \textbf{ sat } S_{SF}(tr, X) \\ \forall\, tr_1, tr, X \bullet (S_{SF}(tr_1, f^{-1}(X) \wedge f(tr_1) = tr) \Rightarrow R_{SF}(tr, X)) \end{array}}{f(P) \textbf{ sat } R_{SF}(tr, X)}$$

If $S_{SF}$ does not constrain all traces $tr_1$ which map to a particular $tr$, then no useful conclusions will be obtained. For example, the result that

$$OFFICE \quad \textbf{sat} \quad foot(tr) = phone.sylvie \Rightarrow answer.sylvie \notin X$$

does not in itself provide any useful information about the behaviour of $f(OFFICE)$ since any trace of $f(OFFICE)$ ending in *phone.janet&sylvie* might have originated from a trace ending in *phone.janet*, and no information is provided about the behaviour in such a circumstance.

□

In the case where $f$ is a 1–1 function, its inverse $f^{-1}$ is well defined and there is only one possibility for the trace $tr_1$ which maps under $f$ to $tr$, namely $f^{-1}(tr)$. In this case the inference rule simplifies as follows:

$$\frac{P \textbf{ sat } S_{SF}(tr, X)}{f(P) \textbf{ sat } S_{SF}(f^{-1}(tr), f^{-1}(X))} \quad [\, f \text{ injective} \,]$$

The backward renaming operator is more straightforward. If $(tr, X)$ is a failure of $f^{-1}(P)$, then $(f(tr), f(X))$ is a failure of $P$, and so it must satisfy whatever specification $P$ is known to satisfy. The inference rule is as follows:

$$\frac{P \textbf{ sat } S_{SF}(tr, X)}{f^{-1}(P) \textbf{ sat } S_{SF}(f(tr), f(X))}$$

The process *SALE* of Example 3.15 is ready to accept payment after a choice of goods has been made.

$$SALE \quad \textbf{sat} \quad foot(tr) = choose \Rightarrow pay \notin X$$

The event renaming function

$$\begin{aligned} f(cash) &= pay \\ f(cheque) &= pay \\ f(credit\_card) &= pay \end{aligned}$$

is used to expand the interface of *SALE* and accept any of these methods of payment. The inference rule yields that

$$f^{-1}(SALE) \quad \mathbf{sat} \quad foot(f(tr)) = choose \Rightarrow pay \notin f(X)$$

and this is equivalent to

$$f^{-1}(SALE) \quad \mathbf{sat} \quad foot(tr) = choose \Rightarrow \{cash, cheque, credit\_card\} \cap X = \{\}$$

After a choice has been made, the process is ready to accept any of the events that map to *pay*—it cannot refuse any of them.

## Sequential composition

Any given failure of $P_1;\ P_2$ must arise from one of two possibilities: either it is a failure of $P_1$ which has not yet reached termination, or else it consists of a trace of $P_1$ followed by a failure of $P_2$. The proof rule reflects this:

$$\frac{\begin{array}{l} P_1 \ \mathbf{sat}\ (S_T(tr), S_1(tr, X)) \\ P_2 \ \mathbf{sat}\ S_2(tr, X) \end{array}}{\begin{array}{ll} P_1;\ P_2 \ \mathbf{sat} & \checkmark \notin \sigma(tr) \wedge S_1(tr, X \cup \{\checkmark\}) \\ & \vee \\ & \exists\, tr_1, tr_2 \bullet tr = tr_1 \frown tr_2 \wedge S_T(tr_1 \frown \langle\checkmark\rangle) \wedge S_2(tr_2, X) \end{array}}$$

The first case covers those failures from $P_1$ that have not yet terminated: in this case, $\checkmark$ must also be refusable. The second case is concerned with those failures corresponding to executions that have passed control from $P_1$ to $P_2$ at some point: in this case, $tr_1 \frown \langle\checkmark\rangle$ is the trace from $P_1$ up to its termination, and so it must meet $P_1$'s trace specification $S_T(tr)$, and $(tr_2, X)$ is the contribution from $P_2$.

## Interrupt

A failure of the interrupt process $P_1 \ \triangle\ P_2$ is either a failure of $P_1$ whose refusal is also a possible initial refusal for $P_2$, or else a non-terminated trace of $P_1$ followed by a failure of $P_2$.

The inference rule is as follows:

$$\frac{\begin{array}{l} P_1 \textbf{ sat } (S_T(tr), S_1(tr, X)) \\ P_2 \textbf{ sat } S_2(tr, X) \end{array}}{\begin{array}{ll} P_1 \bigtriangleup P_2 \textbf{ sat} & S_1(tr, X) \wedge (S_2(\langle\rangle, X) \vee \checkmark \in \sigma(tr)) \\ & \vee \\ & \exists\, tr_1, tr_2 \bullet tr = tr_1 \frown tr_2 \;\; \wedge \checkmark \notin \sigma(tr_1) \wedge S_T(tr_1) \\ & \qquad\qquad\qquad\qquad\quad \wedge\, tr_2 \neq \langle\rangle \wedge S_2(tr_2, X) \end{array}}$$

In the second disjunct, $tr_1$ is the trace contribution from $P_1$, so it meets $P_1$'s trace specification $S_T(tr)$.

EXAMPLE 7.5 If *int* is a special interrupt event, and $P_2$ is initially enabled on this event, unable to refuse it, then

$$P_2 \quad \textbf{sat} \quad int \notin \sigma(tr) \Rightarrow int \notin X$$

Then whatever form process $P_1$ takes, and whatever specification $S_1$ it satisfies, the rule yields that

$$\begin{array}{lll} P_1 \bigtriangleup P_2 & \textbf{sat} & S_1(tr, X) \wedge (int \notin X \vee \checkmark \in \sigma(tr)) \\ & & \vee \\ & & \exists\, tr_1, tr_2 \bullet \quad tr = tr_1 \frown tr_2 \wedge \checkmark \notin \sigma(tr_1) \wedge S_1(tr_1, \{\}) \\ & & \qquad\qquad\quad \wedge\, tr_2 \neq \langle\rangle \wedge (int \notin \sigma(tr_2) \Rightarrow int \notin X) \end{array}$$

which can be weakened to

$$P_1 \bigtriangleup P_2 \quad \textbf{sat} \quad int \notin \sigma(tr) \Rightarrow (int \notin X \vee \checkmark \in \sigma(tr))$$

The interrupt combination will have the interrupt event *int* enabled throughout an execution, until either it occurs or the execution finishes. □

## 7.3 RECURSION INDUCTION

If a recursive definition $N = F(N)$ preserves the satisfiable specification $S_{SF}(tr, X)$—$F(Y)$ **sat** $S_{SF}(tr, X)$ whenever $Y$ **sat** $S_{SF}(tr, X)$—then $N$ must also meet the specification $S_{SF}(tr, X)$.

$$\frac{\forall\, Y \bullet (Y \textbf{ sat } S_{SF}(tr, X) \Rightarrow F(Y) \textbf{ sat } S_{SF}(tr, X))}{N \textbf{ sat } S_{SF}(tr, X)} \quad [\, N = F(N)\, ]$$

In contrast to the rule for trace specifications, no separate check is required for satisfiability of $S_{SF}(tr, X)$, since all such specifications are met by *DIV*.

The rule generalizes to mutual recursion in exactly the same way as it does in the traces model:

$$\frac{\forall\, \underline{Y} \bullet (\underline{Y}\ \mathbf{sat}\ \underline{S}_{SF}(tr,X) \Rightarrow \underline{F}(\underline{Y})\ \mathbf{sat}\ \underline{S}_{SF}(tr,X))}{\underline{N}\ \mathbf{sat}\ \underline{S}_{SF}(tr,X)} \quad [\ \underline{N} = \underline{F}(\underline{N})\ ]$$

EXAMPLE 7.6 The light switch process has a recursive definition:

$$LIGHT \quad = \quad on \rightarrow off \rightarrow LIGHT$$

It appears that whenever the light is on it is ready to be turned off. The appropriate specification to verify is

$$S_{SF}(tr,X) \quad = \quad foot(tr) = on \Rightarrow off \notin X$$

Assume that $Y$ **sat** $S_{SF}(tr,X)$. Then two applications of the inference rule for prefixing establish that

$$\begin{array}{lll} on \rightarrow off \rightarrow Y & \mathbf{sat} & tr = \langle\rangle \wedge on \notin X \\ & & \vee\ tr = \langle on\rangle \wedge off \notin X \\ & & \vee\ tr = \langle on, off\rangle \frown tr' \wedge S_{SF}(tr',X) \end{array}$$

If $foot(tr) = on$, then either $tr = on \wedge off \notin X$ or else $tr = \langle on, off\rangle \frown tr' \wedge foot(tr') = on \wedge S_{SF}(tr',X)$. In either case, $off \notin X$, so the specification weakens to $S_{SF}(tr,X)$, and so

$$on \rightarrow off \rightarrow Y \quad \mathbf{sat} \quad S_{SF}(tr,X)$$

Finally, an application of the inference rule for recursion allows the conclusion that

$$LIGHT = on \rightarrow off \rightarrow LIGHT \quad \mathbf{sat} \quad foot(tr) = on \Rightarrow off \notin X$$

Thus $LIGHT$ **sat** $S_{SF}(tr,X)$ as required. □

EXAMPLE 7.7 (ALTERNATING BIT PROTOCOL) A communications protocol provides a service over a lower level medium which provides a lesser service. The alternating bit protocol provides a service in which messages of type $T$ are relayed between agents without loss, over a medium in which messages can be lost, although they cannot become corrupted or reordered.

The service guaranteed by the underlying medium may be described by a two part CSP specification $Med(in, out)$:

$$\begin{array}{lll} Med_{SF}(in,out) & = & tr \Downarrow out \preccurlyeq tr \Downarrow in \\ & & \wedge\ in.T \cap X = \{\} \vee out.T \nsubseteq X \end{array}$$

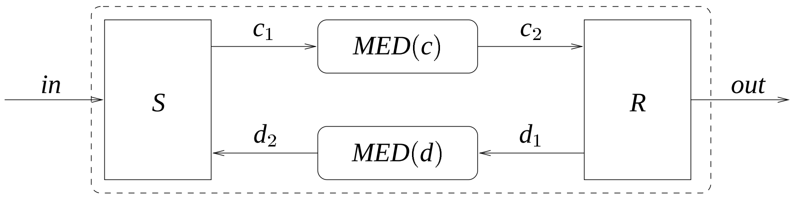


***Fig. 7.1*** The alternating bit protocol

Any output messages should have previously appeared as input messages, and they should appear in the same order, but some messages can be lost. The medium is always ready for either input or output.

A well-behaved process such as *COPY* satisfies the specification $Med(in, out)$, as do less well-behaved processes which can sometimes lose messages.

A medium which accepts input on $c_1$ and provides output on $c_2$ will be referred to as $MED(c)$. Such a medium does not need to be explicitly described in CSP; only the fact that it meets the specification is required for verification.

$$MED(c) \quad \mathbf{sat} \quad Med(c_1, c_2)$$

The protocol itself consists of a sender component $S$ and a receiver component $R$ whose combined behaviour is intended to ensure that no message becomes lost from the system. The two components communicate in each direction over the unreliable medium. The system is illustrated in Figure 7.1.

The sender awaits input, and then transmits it along output channel $c_1$ together with a particular bit $b$. It will wait for an acknowledgement to arrive on channel $d_2$: receipt of the correct bit $b$ indicates that the message arrived, whereas receipt of the incorrect bit $\overline{b}$ should be ignored as being associated with a previous message whose acknowledgement has already been received. The message can be resent as an alternative to waiting for acknowledgement, since it is possible that the message or its acknowledgement were lost during transmission.

$$\begin{array}{rcl} S & = & S(0) \\ S(b) & = & in?x : T \to c_1!(x.b) \to S(b,x) \\ S(b,x) & = & c_1!(x.b) \to S(b,x) \\ & & \Box\ d_2.b \to S(\overline{b}) \\ & & \Box\ d_2.\overline{b} \to S(b,x) \end{array}$$

The receiver process $R$ awaits messages $x.b$ along channel $c_2$, and checks the bit $b$ to see if it is the next bit expected, in which case $x$ must be a fresh message for output. If the bit is not the one expected, then the message must be a repeated transmission, and will not be presented for output. In either case, an acknowledgement consisting of the bit received should be sent out on channel $d_1$.

$$\begin{array}{rcl} R & = & R(0) \\ R(b) & = & c_2?x.b' : (T.\{b\}) \rightarrow out!x \rightarrow d_1!b \rightarrow R(\overline{b}) \\ & & \Box\ c_2?x.b' : (T.\{\overline{b}\}) \rightarrow d_1!\overline{b} \rightarrow R(b) \end{array}$$

The two components $S$ and $R$ do not synchronize on any events, and their combination is described as $S \mid\mid\mid R$ which is equivalent to $S \parallel R$ because of $S$ and $R$'s disjoint alphabets. Similarly, the two channels are independent, and their combination is described as $MED(c) \mid\mid\mid MED(d)$. Finally, the components are composed in parallel, resulting in the following description of the alternating bit protocol:

$$ABP \quad = \quad ((S \mid\mid\mid R) \parallel (MED(c) \mid\mid\mid MED(d))) \setminus (c_1.\mathbb{N} \cup c_2.\mathbb{N} \cup d_1.\mathbb{N} \cup d_2.\mathbb{N})$$

The requirement is that this system should behave as a buffer and satisfy the specification $(Buff_T(tr), Buff_{SF}(tr, X))$.

The safety and liveness aspects of the specification may each be treated in turn. The traces model can be used (see Exercise 7.9) to establish that

$$ABP \quad \mathbf{sat} \quad tr \Downarrow out \leqslant_1 tr \Downarrow in$$

The only further property to establish is deadlock-freedom, in order to establish that the combination is a buffer in terms of liveness as well as safety. The safety specification shows that only one of *in* or *out* can ever be possible: *in* when the buffer is empty, and *out* when non-empty. Hence deadlock-freedom will establish that it must be open to inputs when empty, and ready to output when non-empty. Furthermore, the definition of the sender $S$ is data-independent—once it is ready to accept some input, it is ready to accept any.

The system can be seen to be deadlock-free by considering a stable state. If *in* is refused in this state, then $S$ is live on both $c_1$ and $d_2$. If these are both refused by their respective media, then both $c_2$ and $d_1$ must be enabled within the media. The receiver $R$ cannot be willing to interact on either of these internal events, otherwise the state would not be stable, so it must be ready to provide output

Hence the required result is obtained: that $ABP$ **sat** $Buff_{SF}(tr, X)$. □

## 7.4 PROCESS-ORIENTED SPECIFICATION

As in the traces model, the refinement relation on processes $(T, SF) \sqsubseteq_{SF} (T, SF)$ holds when the second process has fewer possible behaviours than the first.

$$(T_1, SF_1) \sqsubseteq_{SF} (T_2, SF_2) \quad \Leftrightarrow \quad T_2 \subseteq T_1 \wedge SF_2 \subseteq SF_1$$

The subscript *SF* emphasizes the fact that the relationship is defined on the stable failures model.

Refinement holds between two process expressions $P_1$ and $P_2$ whenever it holds between their sets of traces and stable failures. Another way of characterizing the relationship $P_1 \sqsubseteq_{SF} P_2$ is as $P_1 =_{SF} P_1 \sqcap P_2$. The introduction of the traces and stable failures of $P_2$ does not introduce any new behaviours to $P_1$. The subscript *SF* will be elided if it is clear from the context.

The refinement relation $P_1 \sqsubseteq_{SF} P_2$ supports a process-oriented approach to specification. As in the traces model, a specification describes behaviours that are acceptable in a particular situation, and these behaviours can be described either by use of predicates, or else by means of a CSP process expression itself. A process description *SPEC* will have particular traces and stable failures associated with it, and these are taken to be all the acceptable behaviours. An implementation process *IMP* meets this specification if all of its possible behaviours are allowed by *SPEC*, or in other words, if $SPEC \sqsubseteq_{SF} IMP$.

Many common specifications can be captured in a process-oriented style, where the specification is the process with the most behaviours which meets the requirement. This means that *SPEC* should allow all possibilities that are not expressly forbidden.

The process $ALT = a \to b \to ALT$ expresses the requirement that the performance of $a$'s and $b$'s should alternate. It also contains the requirement that these events should be available, and that no other events are possible, since none appear as possibilities in $ALT$. A weaker specification which places no constraint on any other events would be $ALT \mid\mid\mid CHAOS_{\Sigma \setminus \{a,b\}}$, which allows arbitrary behaviour on all other events, but still requires that $a$ and $b$ must be available when they are next in the alternating sequence. An even weaker specification, which also allows the possibility of deadlock, would be $ALT \underset{\{a,b\}}{\parallel} CHAOS$. Any sequence of $a$'s and $b$'s must still be alternating, but no liveness conditions on them are present.

EXAMPLE 7.8 (BUFFERS) The property of being a buffer of type $T$ is expressible in a process-oriented way, by means of a mutual recursion. Internal choice is used to describe the various possibilities:

$$\begin{aligned}
NBUFF_T(\langle\rangle) &= in?x : T \to NBUFF_T(\langle x \rangle) \\
NBUFF_T(s \frown \langle y \rangle) &= out!y \to NBUFF_T(s) \\
&\quad \Box\ (STOP \sqcap in?x : T \to NBUFF_T(\langle x \rangle \frown s \frown \langle y \rangle))
\end{aligned}$$

The parameter to *NBUFF* consists of the sequence of messages that the buffer currently contains. If this sequence is the empty sequence $\langle\rangle$, then the buffer is empty and must be ready for input. If the sequence contains some messages, then the buffer must be ready to output the next message required. It is also possible that it will accept further input, but it does not have to. These possibilities are represented by the internal choice between *STOP* and a further input.

The buffer specification $NBUFF_T = NBUFF_T(\langle\rangle)$ also specifies that the alphabet of the buffer is restricted to its input and output channels. It encapsulates the buffer specification given on Page 194:

$$NBUFF \sqsubseteq_{SF} IMP \quad \Leftrightarrow \quad IMP \textbf{ sat } (Buff_T(tr), Buff_{SF}(tr, X))$$

If further events are to be possible (such as a channel which can report on whether or not the buffer is empty), then the appropriate specification will be $NBUFF_T \ ||| \ CHAOS_A$, where $A$ are the other possible events. The most general specification, that corresponds to $Buff_T(tr, X)$, is given as follows:

$$\begin{array}{l} NBUFF_T \ ||| \ CHAOS_{\Sigma \setminus (in.T \cup out.T)} \sqsubseteq_{SF} IMP \\ \quad \Leftrightarrow \quad IMP \textbf{ sat } (Buff_T(tr), Buff_T(tr, X)) \end{array}$$

However, in general it will be more appropriate to restrict $A$ to the particular set of events which are allowed for the buffer. □

One of the benefits of the process-oriented approach is provided by the availability of model-checking tools which permit automatic checking of (finite state) specifications against implementations. The FDR tool allows processes to be checked against process-oriented specifications with regard to their stable failures.

The two styles of specification can often be combined within verification. If $SPEC \sqsubseteq_{SF} IMP$ and $SPEC$ **sat** $Spec_{SF}(tr, X)$, then $IMP$ **sat** $Spec_{SF}(tr, X)$, and this result can be used within the application of a proof rule.

A process-oriented version of the proof rule for recursion induction will use $SPEC \sqsubseteq_{SF} Y$ in place of $Y$ **sat** $S_{SF}(tr, X)$, resulting in the following antecedent:

$$\forall Y \bullet (SPEC \sqsubseteq_{SF} Y \Rightarrow SPEC \sqsubseteq_{SF} F(Y))$$

This is equivalent to the assertion that $SPEC \sqsubseteq_{SF} F(SPEC)$. The rule becomes

$$\frac{SPEC \sqsubseteq_{SF} F(SPEC)}{SPEC \sqsubseteq_{SF} N} \quad [\ N = F(N)\ ]$$

Even if the relation $SPEC \sqsubseteq_{SF} N$ cannot be checked directly by mechanical means, for example if $N$ has infinitely many states, it can still be verified via the proof rule by checking that $SPEC \sqsubseteq_{SF} F(SPEC)$.

EXAMPLE 7.9 The bag process $BAG$ takes messages as input, and makes them available for output. It is given by a guarded recursive definition.

$$BAG \quad = \quad in?x : T \rightarrow (BAG \ ||| \ (out!x \rightarrow STOP))$$

One property that this process satisfies is that it is always ready for input. This specification can be captured as the process

$$INS \quad = \quad RUN_{in.T} \; ||| \; CHAOS_{out.T}$$

It is not possible to check $INS \sqsubseteq_{SF} BAG$ directly using a model-checker, since the process $BAG$ has an infinite number of states. However,

$$INS \quad \sqsubseteq_{SF} \quad in?x : T \rightarrow (INS \; ||| \; (out!x \rightarrow STOP))$$

which can be automatically checked, since $INS$ has a finite (and extremely small) number of states. This single refinement check establishes the antecedent to the inference rule for guarded recursion given above, establishing the result

$$INS \sqsubseteq_{SF} BAG$$

The process $BAG$ is always ready for input. $\square$

## 7.5 CASE STUDY: DISTRIBUTED SUM

The functional correctness of the distributed sum algorithm was established in Chapter 5. Only traces need to be considered in order to establish that any answer provided by the system of nodes must be the correct one.

However, the trace analysis does not provide any guarantees that an answer will eventually be output. This is a liveness property, so an analysis in the stable failures model is required. The main aim will be to establish deadlock-freedom of the network; more specific liveness properties will follow from this. It will also ultimately be necessary to establish that the network is free from divergence. This will be discussed in Chapter 8.

The liveness of the network as a whole will rely on the liveness properties of the individual nodes. The particular properties used to prove deadlock-freedom will be $I1$ and $I2$ concerning liveness on inputs, $O1$ and $O2$ concerning liveness on outputs, and $T1$ concerning liveness on termination. A subsidiary result $N1$ is also useful: it can be established in the traces model.

### Liveness on input

All nodes will be initially live on all of their input channels. This is specified for each input channel $c_{ij}$ as follows:

$$I1_{ij} \qquad NODE(j) \textbf{ sat } tr \upharpoonright A_j^{\checkmark} = \langle\rangle \Rightarrow c_{ij}.0 \notin X$$

Given a particular node $j$ and edge $(i,j) \in E$, if node $j$ has not yet communicated with any neighbour, then it must be ready for an initiating input from $i$.

Furthermore, any node $j$ on any particular input channel $c_{ij}$ will remain willing to accept input until it occurs (see Exercise 7.11).

$$I2_{ij} \qquad NODE(j)\ \mathbf{sat}\ (tr \upharpoonright A_j^{\checkmark} \neq \langle\rangle \wedge tr \Downarrow c_{ij} = \langle\rangle) \Rightarrow c_{ij}.\mathbb{N} \cap X = \{\}$$

Given a particular node $j$ and edge $(i,j) \in E$, if node $j$ has performed some communication but has not yet received any input from its neighbour $i$, then it must be ready to input any possible value.

## Liveness on output

The first liveness property on output is that, once some message has been received along a channel $c_{ki}$, then output is available along any channel $c_{ij}$ other than the one matching the initial input. This means that a node is ready to provide output to any of its neighbours with the possible exception of the neighbour it first interacted with. The relationship between two neighbours $i$ and $j$ is expressed as follows:

$$\begin{array}{lll} O1_{ij} & NODE(i)\ \mathbf{sat} & tr \neq \langle\rangle \wedge \mathsf{channel}(head(tr)) \neq c_{ji} \\ & & \quad \Rightarrow (tr \Downarrow c_{ij} = \langle\rangle \Rightarrow c_{ij}.0 \notin X) \end{array}$$

The process $c_{ij}!0 \rightarrow SKIP$ is live on channel $c_{ij}$ until it occurs:

$$c_{ij}!0 \rightarrow SKIP \quad \mathbf{sat} \quad (tr \Downarrow c_{ij} = \langle\rangle \Rightarrow c_{ij}.0 \notin X)$$

Furthermore, the channel $c_{ij}$ is not in the alphabet of any of the processes $c_{ik}!0 \rightarrow SKIP$ where $k \neq j$, nor of $TOT(i,k,adj(i) \setminus \{k\}, w_i)$ where $k \neq j$. It follows that the same specification is met by the parallel combination (provided $k \neq j$):

$$\begin{array}{l} (TOT(i,k,adj(i) \setminus \{k\}, w_i) \parallel (\big\|^{l \in adj(i) \setminus \{k\}}\ c_{ij}!0 \rightarrow SKIP)) \\ \quad \mathbf{sat} \quad (tr \Downarrow c_{ij} = \langle\rangle \Rightarrow c_{ij}.0 \notin X) \end{array}$$

Prefixing this process with an input $c_{ki}.0$ will yield the required specification when $j \neq k$

$$\begin{array}{l} c_{ki}.0 \rightarrow (TOT(i,k,adj(i) \setminus \{k\}, w_i) \parallel (\big\|^{l \in adj(i) \setminus \{k\}}\ c_{ij}!0 \rightarrow SKIP)) \\ \quad \mathbf{sat} \quad tr \neq \langle\rangle \wedge \mathsf{channel}(head(tr)) \neq c_{ji} \Rightarrow (tr \Downarrow c_{ij} = \langle\rangle \Rightarrow c_{ij}.0 \notin X) \end{array}$$

In the case where $j = k$, then the specification is vacuously satisfied since whenever $tr \neq \langle\rangle$ then $\mathsf{channel}(head(tr)) = c_{ji}$. Hence the specification is met in all cases, and $O1_{ij}$ follows from the fact that $NODE(i)$ is an indexed choice between all of these processes.

The other liveness property required on output is that once a node has received inputs from all of its neighbours, it is ready to output to the neighbour that it first communicated with. The set of input possibilities to a node $i$ will be defined as $A_i^{in}$:

$$A_i^{in} \quad = \quad \{c_{ji}.n \mid c_{ji}.n \in A_i\}$$

If all the input channels are mentioned in the trace, then the node has the required liveness property:

$$\begin{array}{lll} O2_{ij} & NODE(i) \ \mathbf{sat} & \mathsf{channels}(A_i^{in}) \subseteq \mathsf{channels}(tr) \wedge \mathsf{channel}(head(tr)) = c_{ji} \\ & & \quad \Rightarrow ((tr \Downarrow c_{ij} = \langle\rangle \Rightarrow c_{ij}.\mathbb{N} \not\subseteq X)) \end{array}$$

The proof of this property is left as an exercise (see Exercise 7.13).

## Liveness on termination

When a node has had some communication along each of its channels, it is either ready to terminate or else already terminated:

$$T1_i \qquad NODE(i) \ \mathbf{sat} \ \mathsf{channels}(A_i) \subseteq \mathsf{channels}(tr) \Rightarrow (\checkmark \ \mathbf{in} \ tr \vee \checkmark \notin X)$$

The proof of this property is left as Exercise 7.14.

## Safety on the nodes

The safety property is simply that the first event of any process must be on one of its input channels. This may be established in the traces model.

$$N1_i \qquad NODE(i) \ \mathbf{sat} \ tr \neq \langle\rangle \Rightarrow \mathsf{channel}(head(tr \upharpoonright A_i^{\checkmark})) \in A_i^{in}$$

A single application of the proof rule for input (using simply $P(x)$ **sat** $true(tr)$ as the antecedent) establishes that this specification holds for any process of the form $c?x : T \rightarrow P(x)$ for which $c \in A_i^{in}$. This is indeed the case for all channels of the form $c_{ji}$ where $j \in adj(i)$, so the proof rule for indexed external choice yields that any process of the form

$$\Box_{j \in adj(i)} \ c_{ji}?x : T \rightarrow P(x, j)$$

must also satisfy this specification, since each of its components do. The description of $NODE(i)$ is of this form, so $N1_i$ follows without any need to consider the behaviour following the first event.

## Deadlock-freedom for the network

The properties of the individual nodes given above are sufficient to establish that the process $NETWORK = \big\|_{A_i} NODE(i)$ is deadlock-free.

Consider a failure $(tr, X)$ of $NETWORK$. Then the refusal set is made up of a family of refusal sets $\langle X_i \rangle_{i \in N}$, one for each $i \in N$, where $X = \bigcup_{i \in N}(X_i \cap A_i^{\checkmark})$, and $(tr \upharpoonright A_i^{\checkmark}, X_i) \in \mathcal{SF}\,[\![NODE(i)]\!]$. The special channels $c_{0\infty}$ and $c_{\infty 0}$ appear only in $A_0$. All other channels $c_{ij}$ appear in only two alphabets, $A_i$ and $A_j$. This means that if there is some value $v$ for which $c_{ij}.v \notin X_j$ and $c_{ij}.v \notin X_i$, then $c_{ij}.v \notin X$. If it is offered by both $NODE(i)$ and $NODE(j)$, then it cannot be blocked by any other component.

To establish deadlock-freedom for $NETWORK$, all the possible cases for the trace $tr$ will be considered, and in each case (before termination) the liveness properties will be enough to show that there is some communication which does not appear in the refusal set $X$.

There are essentially two cases to consider: whether or not any of the nodes are still in their initial state. The two cases each split into a number of subcases.

Case $\exists\, i \bullet tr \upharpoonright A_i^{\checkmark} = \langle\rangle$:

Subcase $tr \Downarrow c_{\infty,0} = \langle\rangle$: In this case $(tr \upharpoonright A_0^{\checkmark}) \Downarrow c_{\infty,0} = \langle\rangle$, and so $(\langle\rangle, X_0)$ is a failure of $NODE(0)$. It follows from $I1_{\infty 0}$ that $c_{\infty,0}.0 \notin X_0$, and so $c_{\infty 0}.0 \notin X$.

Subcase $tr \Downarrow c_{\infty,0} \neq \langle\rangle \wedge \exists j \bullet tr \upharpoonright A_j^{\checkmark} = \langle\rangle$: In this case the set of nodes can be partitioned into the nodes that have engaged in some event, and those that have not. Both sets will be non-empty:

$$
\begin{aligned}
S_1 &= \{i \in N \mid tr \upharpoonright A_i^{\checkmark} \neq \langle\rangle\} \\
S_2 &= N \setminus S_1
\end{aligned}
$$

Connectedness of the graph $(N, E)$ implies that there must be some edge $(i, j) \in E$ connecting the two sets: $i \in S_1$ and $j \in S_2$. Then $c_{ij} \in A_j$, so $tr \Downarrow c_{ij} = \langle\rangle$ because $j \in S_2$, so $I1_{ij}$ yields that $c_{ij}.0 \notin X_j$.

Furthermore, $tr \upharpoonright A_1^{\checkmark} \neq \langle\rangle$ since $i \in S_1$, and $tr \Downarrow c_{ji} = \langle\rangle$ because $j \in S_2$, so from $O1_{ij}$ it follows that $c_{ij}.0 \notin X_i$. Thus $c_{ij}.0$ does not appear in $X$, which establishes the case.

Case $\forall\, i \bullet tr \upharpoonright A_i^{\checkmark} \neq \langle\rangle$:

In this case, every node has participated in some event in the trace $tr$. In each case, the first channel a node $i$ has interacted on will be $c_{ji} = \mathsf{channel}(head(tr \upharpoonright A_i^{\checkmark}))$ for some other node $j$. The set of all such channels is defined to be the set $T$:

$$
T = \{\mathsf{channel}(head(tr \upharpoonright A_i^{\checkmark})) \mid i \in N\}
$$

These are the channels from parent to child nodes. They form a tree.

Property $N1_i$ means that for any $i$ the set $T \cap A_i^{in}$ contains exactly one channel: each node has exactly one parent.

If $c_{ji} \in T$ then node $i$ will pass the sum of its inputs, together with its own weight $w_i$, along the complementary channel $c_{ij}$. The set of all such channels is defined to be $T'$:

$$T' \quad = \quad \{c_{ij} \mid c_{ji} \in T\}$$

This is the set of channels from child nodes to their parent nodes. These are the channels that carry values rather than initiating messages.

The set of all channels apart from those in $T'$ is given by

$$\overline{T'} \quad = \quad (\bigcup_{i \in N} \mathsf{channels}(A_i)) \setminus T'$$

These are the channels that carry initiating messages.

Subcase $\overline{T'} \not\subseteq \mathsf{channels}(tr)$: The first subcase to consider is where not all initiating messages have yet occurred: not all channels in $\overline{T'}$ appear in the trace.

In this case there is a channel $c_{ij} \in \overline{T'}$ such that $tr \Downarrow c_{ij} = \langle\rangle$. This means that $c_{ji} \neq channel(head(tr \upharpoonright A_i^{\checkmark}))$. The property $O1_{ij}$ implies that $c_{ij}.0 \notin X_i$, and $I2_{ij}$ means that $c_{ij}.0 \notin X_j$. Hence $c_{ij}.0$ cannot appear in the refusal set $X$.

Any channel in $\overline{T'}$ that has not appeared in the trace cannot have $0$ refused.

Subcase $\overline{T'} \subseteq \mathsf{channels}(tr) \wedge T' \not\subseteq \mathsf{channels}(tr)$: In this subcase, all initiating messages have occurred (all channels in $\overline{T'}$ have been used) but not all values have yet been returned: some channels in $T'$ have not yet been used. Then the set of channels $U' \subseteq T'$ that have not been used by a child node to return its value to its parent node is non-empty:

$$U' \quad = \quad \{c_{ij} \in T' \mid tr \Downarrow c_{ij} = \langle\rangle\}$$

The channels in the opposite direction to those in $U'$, from parent to child, are given by $U$:

$$U \quad = \quad \{c_{ji} \mid c_{ij} \in U'\}$$

Thus $U \subseteq T$. The sets $T$, $T'$, $U'$ and $U$ corresponding to the point reached in Diagram 3 of Figure 5.2 are illustrated in Figure 7.2. The set $U$ consists of those channels $c_{ij}$ which are the first channels used by some node $i$ such that $i$ has not communicated its output along the corresponding channel $c_{ji}$. The last of those channels to have appeared in the trace is given by $\mathsf{channel}(foot(tr \upharpoonright U)) = c_{lk}$ for some $l$ and $k$. The aim is to show that $c_{kl}.\mathbb{N}$ cannot be refused.

To establish this, it is sufficient to show that all of the inputs channels $A_k^{in}$ to node $k$ must appear in the trace $tr$:

1. Firstly, all of those in $\overline{T'}$ must have occurred, since all channels in $\overline{T'}$ appear in $tr$ in this subcase.

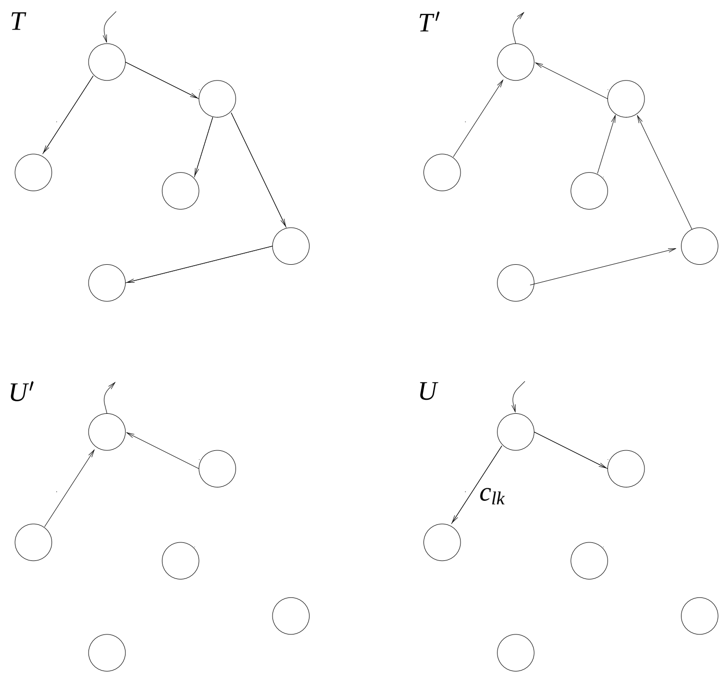


***Fig. 7.2*** Example edge sets $T$, $T'$, $U'$, and $U$

2. Secondly, all of $k$'s input channels in $T'$ must also appear in $tr$. If any of $k$'s input channels do not appear in $tr$, then there is some input channel $c_{mk} \in U'$, since $U'$ consists of those channels of $T'$ that do not appear in $tr$. This means that the corresponding channel $c_{km} \in U$. Both $c_{km}$ and $c_{lk}$ are in the alphabet $A_i$ of $NODE(i)$. The fact that $c_{lk} \in A_k^{in}$ means that $c_{lk} = \mathsf{channel}(head(tr \upharpoonright A_k^{\checkmark}))$, and so $c_{km}$ must appear after $c_{jk}$ in the trace $tr$. But this is impossible, since $\mathsf{channel}(foot(tr \upharpoonright U)) = c_{lk}$, so no other channel in $U$ can appear in the trace after $c_{lk}$.

Thus all of $k$'s input channels appear in $tr$, so $\mathsf{channels}(A_k^{in}) \subseteq \mathsf{channels}(tr \upharpoonright A_k^{\checkmark})$. Since $tr \Downarrow c_{kl} = \langle\rangle$, property $O2_{kl}$ yields that $c_{kl}.\mathbb{N} \nsubseteq X_k$, and so there is some number $v$ for which $c_{kl}.v \notin X_k$. If $l = \infty$ (and $k = 0$) then this is sufficient to establish that $c_{kl}.v \notin X$. Otherwise, property $I2_{kl}$ yields that $c_{kl}.v \notin X_j$. Hence in either case $c_{kl}.v$ cannot appear in the refusal set $X$.

Subcase $\overline{T'} \subseteq \mathsf{channels}(tr) \wedge T \subseteq \mathsf{channels}(tr)$: In this case all of the channels have been used, so for any node $i$

$$\mathsf{channels}(A_i) \subseteq \mathsf{channels}(tr \upharpoonright A_i^{\checkmark})$$

The property $T1$ on each node yields in each case that either $\checkmark$ **in** $tr \upharpoonright A_i^{\checkmark}$ or that $\checkmark \notin X_i$.

1. If $\checkmark \notin tr$, then $\checkmark \notin tr \upharpoonright A_i^{\checkmark}$ for any node $i$, and so $\checkmark \notin X_i$ for any $i$, and hence $\checkmark \notin X$.
2. If $\checkmark \in tr$, then the execution has terminated and deadlock is no longer a concern.

Hence no possible trace of *NETWORK* is associated with a deadlock before termination, so *NETWORK* is deadlock-free.

The deadlock-freedom of *DISTSUM* allows further conclusions to be drawn about its behaviour. The alphabet of *DISTSUM* is simply $c_{0\infty}.\mathbb{N} \cup c_{\infty 0}.\mathbb{N} \cup \{\checkmark\}$. This is a subset of the alphabet of node $0$, so any constraints imposed by that node on the order of these events must be respected by *DISTSUM*.

In fact (see Exercise 5.15) $NODE(0)$ satisfies the following safety specifications:

$$\begin{array}{l} tr \Downarrow c_{0\infty} \neq \langle\rangle \Rightarrow tr \Downarrow c_{\infty 0} \neq \langle\rangle \\ tr \upharpoonright \checkmark \neq \langle\rangle \Rightarrow tr \Downarrow c_{0\infty} \neq \langle\rangle \\ (tr \Downarrow c_{\infty 0}) \leqslant \langle 0 \rangle \\ (tr \downarrow c_{0\infty}) \leqslant 1 \end{array}$$

These together mean that any first event of *DISTSUM* must be $c_{0\infty}.0$, any second event must be a communication along the channel $c_{0\infty}$, and any third event must be termination. Deadlock-freedom means that each of these events must be available in turn, so the behaviour of *DISTSUM* is equivalent to a process

$$c_{\infty 0}.0 \rightarrow c_{0\infty}!v \rightarrow SKIP$$

for some value $v$. The safety specification proven in the previous chapter ensures that the value $v$ is the sum of all the weights of the nodes, and so

$$DISTSUM \quad =_{SF} \quad c_{\infty 0}.0 \rightarrow c_{0\infty}!(\Sigma_{i \in N} w_i) \rightarrow SKIP$$

## Exercises

EXERCISE 7.1 A cheese shop sells Stilton, Brie, Gouda, and Jarlsberg cheese. Specify that

1. All of these will always be available;
2. All of these will initially be available;
3. At any time at least three are available;
4. At any time some cheese is available;

5. After a delivery there is at least one cheese available.

EXERCISE 7.2 If $P$ is deadlock-free, then prove that $P \setminus A$ is also deadlock-free.

EXERCISE 7.3 If $P_1$ and $P_2$ are strongly deadlock-free, then prove that so too is $P_1 \underset{\{a\}}{\|} P_2$. Must $P_1 \underset{\{a,b\}}{\|} P_2$ be deadlock-free?

EXERCISE 7.4 Prove that if $P_1$ and $P_2$ are always live on an input *in*, then so too is $P_1 \underset{in.T}{\|} P_2$.

EXERCISE 7.5 Prove that

$$\forall Y \bullet (SPEC \sqsubseteq_{SF} Y \Rightarrow SPEC \sqsubseteq_{SF} F(Y))$$

is equivalent to the assertion that $SPEC \sqsubseteq_{SF} F(SPEC)$.

EXERCISE 7.6 Prove that $P = on \rightarrow off \rightarrow P$ **sat** $foot(tr) = off \Rightarrow on \notin X$. [You will have to strengthen the specification before the recursion rule can be applied.]

EXERCISE 7.7 Prove that $P_1 \sqcap P_2 \sqsubseteq_{SF} P_1 \Box P_2$ for any $P_1$ and $P_2$.

EXERCISE 7.8 A stack process (or last-in-first-out queue) of type $T$ must always be receptive to input elements of $T$ along channel *push* when empty, and is always willing to output along channel *pop* when non-empty. It can accept further input when nonempty, but is not obliged to. Its output is always the item most recently input that has not yet been output. The process defined in Example 1.23 may be considered a process-oriented specification for the traces model.

1. Give a process-oriented specification of a stack for the stable failures model, which allows for the possibility of non-empty stacks being full.
2. Prove that $STACK_1 = push?x : \mathbb{N} \rightarrow pop!x \rightarrow STACK_1$ is a stack.
3. Prove that $push?x : T \rightarrow STACK_2(x)$ is a stack, where

$$\begin{aligned} STACK_2(x) \quad = \quad & pop!x \rightarrow push?y : T \rightarrow STACK_2(y) \\ & \mid push?y : T \rightarrow pop!y \rightarrow STACK_2(x) \end{aligned}$$

EXERCISE 7.9 (HARD) Prove that the alternating bit protocol of Example 7.7 meets its trace specification:

$$ABP \quad \textbf{sat} \quad tr \Downarrow out \leqslant_1 tr \Downarrow in$$

EXERCISE 7.10 If the graph of nodes in the distributed sum *NETWORK* process is not connected, which part of the liveness proof breaks down? What will be the behaviour of the network in this case, and at what point will it deadlock?

EXERCISE 7.11 Prove that

$$I2_{ij} \qquad NODE(j) \ \mathbf{sat} \ (tr \upharpoonright A_j^{\checkmark} \neq \langle\rangle \wedge tr \Downarrow c_{ij} = \langle\rangle) \Rightarrow c_{ij}.\mathbb{N} \cap X = \{\}$$

Hint: a useful starting point is to establish by a mutual recursion induction that the *TOT* family of processes meet a corresponding family of specifications:

$$\begin{array}{lll} TOT(j,k,M,t) & \mathbf{sat} & i \notin M \vee \\ & & (tr \Downarrow c_{ij} = \langle\rangle \Rightarrow c_{ij}.\mathbb{N} \cap X = \{\} \end{array}$$

EXERCISE 7.12 Use the traces model to prove $N1_i$ on Page 212.

EXERCISE 7.13 [Harder] Prove $O2_{ij}$ on Page 212.

EXERCISE 7.14 Prove $T1_i$ on Page 212.

# 8 Failures, divergences, and infinite traces

## 8.1 OBSERVING PROCESSES

The stable failures model records the occurrence of events as processes perform them, and their refusal after processes stabilize. This approach is effective for processes that cannot diverge. When divergence is a possibility then the stable failures model is not discriminating enough, since it completely ignores any divergent behaviour that a process might have.

In order to analyze processes for the possibility of divergence, it is necessary to introduce the appropriate observations into the model. There are two kinds of behaviour that have a bearing on divergence: traces that lead to a divergent state, and infinite traces that might give rise to divergence. These are introduced alongside failures information to yield the *Failures/Divergences/Infinite Traces* model, which is the concern of this chapter.

### Divergence

When a process executes, it may pass through a sequence of process states by means of internal $\tau$ transitions. In an unstable state, external events might be possible as well as the internal event, but there can be no assurance that they will be available to the environment of the process, since there is no way the $\tau$ transition can be prevented from firing. It is appropriate to consider guarantees on event offers only for those states which have no internal transitions leading from them.

If a process $P$ is able to perform an infinite sequence of internal events, then there is no guarantee that it will ever reach a stable state, and in fact there is no guarantee that it will ever

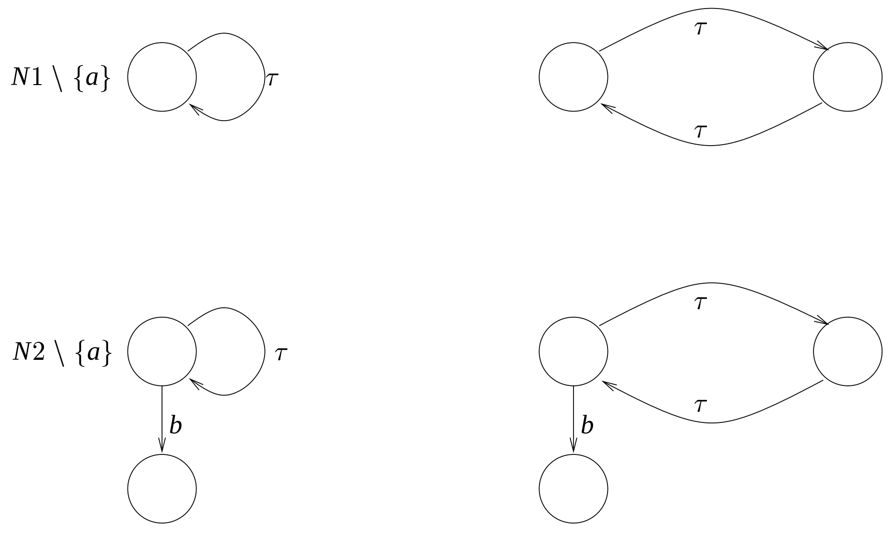


***Fig. 8.1*** Divergent processes

respond to any offer that its environment can make to it. As defined on Page 173, $P$ is said to be *divergent*, written $P \uparrow$ .

Divergence is the worst possible behaviour that a component of a system may exhibit, since any other component waiting to synchronize with it might remain waiting for ever, in anticipation that the process will eventually reach a point where it will synchronize. Infinite computing resources will be consumed during a divergent execution. Figure 8.1 illustrates some divergent processes.

EXAMPLE 8.1 The process $N_1 \setminus \{a\}$ has a divergent execution, where $N_1$ is defined recursively as $N_1 = a \rightarrow N_1$. Thus $N_1 \setminus \{a\} \uparrow$ . This is pictured in Figure 8.1. $\square$

EXAMPLE 8.2 The process $N_2 \setminus \{a\}$ has a divergent execution, where $N_2$ is defined recursively as $N_2 = (a \rightarrow N_2 \mid b \rightarrow STOP)$. There is an infinite sequence of internal transitions that can be performed. Thus $N_2 \setminus \{a\} \uparrow$ . This is pictured in Figure 8.1. $\square$

If a process $P$ is unable to diverge, this means that there are no infinite sequences of internal transitions starting from $P$, and so every sequence of internal transitions must eventually reach a stable state $P' \downarrow$ from which no further internal events are possible.

## Divergent traces

If a divergent state is reached after a finite sequence of events, then the sequence $tr$ is recorded as a *divergent trace* of the process. No guarantees about the behaviour of the process can be made subsequent to a divergent trace. The trace $tr$ is a divergent trace of $P$ if there is some divergent process $P'$ such that $P \overset{tr}{\Longrightarrow} P' \wedge P' \uparrow$ . For example, the process $N_2 \setminus \{a\}$ of Example 8.2 has $\langle\rangle$ as a divergent trace.

### Infinite traces

If an infinite sequence of offers are all accepted, then an infinite trace $u$ will be recorded. No refusal information will appear since there is no point at which offers are refused. If the trace $u = \langle a_0, a_1, a_2 \dots \rangle$ then $u$ will be an infinite trace of $P$ if there is an infinite sequence $\langle P_i \rangle_{i \in \mathbb{N}}$ such that

$$P = P_0 \wedge \forall\, i \bullet P_i \overset{\langle a_i \rangle}{\Longrightarrow} P_{i+1}$$

For example, the process $P = a \to P$ has $\langle a, a, a, \dots \rangle$ as an infinite trace—in fact its only one.

The set of all possible infinite traces is denoted *ITRACE*, and the variable $u$ is used to range over *ITRACE*. It is the set of all infinite sequences of events from $\Sigma$. Such sequences cannot contain the termination event ✓—if termination occurs at some point in a trace then it must be last.

### Failures

The stable failures of a process will continue to be recorded. However, a divergent process is also able to refuse any set of offered events by default, since it is able to ignore permanently any offers made to it simply by following the divergent execution. Observations of such behaviour will also be recorded as failures.

For example, the process $N_2 \setminus \{a\}$ of Example 8.2 has $(\langle\rangle, \{a, b\})$ as a possible (unstable) failure, since it can engage in a divergent execution after the empty trace, and hence refuse $\{a, b\}$.

The inclusion of both stable and unstable failures as observations means that the traces of a process no longer need to be recorded separately. Unlike the stable failures model, all traces of a process will appear in the failure set. Every trace of a process will either be divergent, or will lead to a stable state. In either case, it will be associated with the refusal $\{\}$, either from a stable refusal of the empty set (which is possible for any stable state, since no events are offered), or else for an unstable refusal. Hence whenever $tr$ is a trace of a process, then $(tr, \{\})$ will be a failure of that process.

EXAMPLE 8.3 The process $N_3$ pictured in Figure 8.2 has $\langle c \rangle$ and $\langle a, b, c \rangle$ as two of its possible divergent traces. It has $(\langle a, b \rangle, \{b\})$ as a possible failure, and $\langle a, b, a, b, \dots \rangle$ as an infinite trace. □

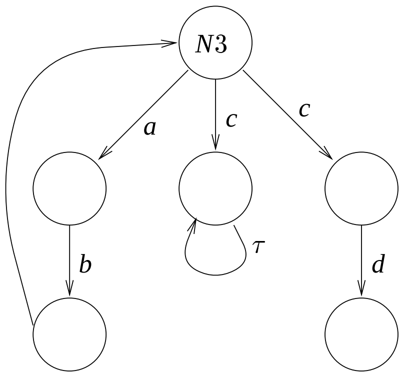


***Fig. 8.2*** Divergent processes

## Semantic model

The failures, divergences, and infinite traces model (abbreviated as the *FDI* model) for CSP identifies a process *P* with the failures, divergences, and infinite traces that may be observed of it. This model is even more discriminating than the stable failures model, as it takes account of divergent and infinite behaviour as well as stable failures, but the underlying approach taken to the semantics and to specification and verification is the same as in all the models for CSP.

If three sets *F*, *D*, and *I* are to correspond respectively to the possible failures, divergences, and infinite traces of some process, there are some consistency conditions that they should meet, both within and between the sets. These are properties that must hold of any triple of sets which describe some process.

Firstly, the set *F* should not be empty: there must always be some observation. Although each of *D* and *I* can be empty (since it is reasonable for processes to have no divergent or infinite traces) any experiment on a process will give some response. In fact, it is possible to be more specific: if the empty set of events is initially offered to any process, then it will be refused, since the empty set can always be refused. This is described by the failure $(\langle\rangle, \{\})$, which must therefore appear in any process' set of failures.

$F1 \qquad (\langle\rangle, \{\}) \in F$

The second property is also inherited from the traces model, although it is formulated differently because of the presence of refusal sets. If a failure $(tr, X)$ is in the set *F* of failures associated with a process, then that process must be able to perform any prefix $tr'$ of the trace $tr$. Although the failure gives no information about possible refusals after the trace $tr'$, the empty refusal set $\{\}$ must always be possible after $tr'$.

$F2 \qquad (tr, X) \in F \wedge tr' \leqslant tr \Rightarrow (tr', \{\}) \in F$

There is also a property of subset closure in the refusal component of a behaviour: if a set $X$ can be refused after a trace $tr$, then any subset $X'$ of $X$ can also be refused after that trace.

$$F3 \qquad (tr, X) \in F \wedge X' \subseteq X \Rightarrow (tr, X') \in F$$

The final property of the failures component $F$ is concerned with the relationship between refusals and events that are not possible. If $(tr, X) \in F$, and no event from $X'$ can follow the trace $tr$, then the set $X'$ can augment the refusal set. Events are either possible or refusable.

$$F4 \qquad (tr, X) \in F \wedge \forall\, a \in X' \bullet (tr \frown \{a\}, \{\}) \notin F \Rightarrow (tr, X \cup X') \in F$$

Since $\checkmark$ is always the last event in a trace, $F4$ means that any set of events can be refused after $\checkmark$ has occurred, since the trace cannot be extended beyond $\checkmark$. A terminating trace can be associated with all refusal sets.

The FDI model takes a pessimistic approach to recording process behaviours. In order to guarantee that certain behaviours are not possible, which will be required for verification, it is necessary to allow for the possibility of all behaviours that cannot be definitely excluded. Behaviours are known to be possible if they can actually be observed during some interaction with the process, and correspond directly to some execution. Certainly these behaviours cannot be excluded. However, divergence complicates the picture, because an extremely negative view is taken of divergent processes.

Any process $P$ which is attempting to interact with a divergent process $P_D$ can guarantee nothing about the results of any attempted interaction, or even about its own independent progress, since the divergent execution might take precedence over any activity that $P$ would prefer, and the parallel combination of $P$ and $P_D$ is also divergent. Since nothing can be guaranteed about any future behaviour, nothing can be excluded. The possibility of divergence masks all other possible executions, so once a process is divergent then all other behaviours should be allowed as possible observations. Any process or environment interacting with a purely divergent process which can only loop has no guarantees about further interactions, just as if it were interacting with a divergent process that has some other well-defined executions. Even the possibility of divergence is considered to be catastrophic. This is true in a precise testing sense, as will be discussed at the end of this chapter. The reason for recording divergence within the FDI model is because it is a phenomenon that may arise, so the possibility must be incorporated in order to provide a framework which can establish that it has not arisen in particular systems.

Hence once a divergence has occurred, then no behaviour can be excluded from the possibilities associated with a process, so any behaviour prefixed by a divergent trace will be included in the semantics of a process. This means that any trace which contains a divergent trace as a prefix will also be in the set of possible divergences:

$$D1 \qquad tr \in D \wedge tr \leqslant tr' \Rightarrow tr' \in D$$

Any possible divergence can be associated with any refusal set $X$, and appear in the set of possible refusals:

$$D2 \qquad tr \in D \wedge X \subseteq \Sigma^{\checkmark} \Rightarrow (tr, X) \in F$$

Any infinite trace which has a divergent trace as a prefix cannot be excluded as a possible behaviour:

$$D3 \qquad tr \in D \wedge tr \leqslant u \Rightarrow u \in I$$

Once a process has diverged then nothing can be ruled out.

Finally, the relationship between divergence and termination is as follows:

$$D4 \qquad tr \frown \langle \checkmark \rangle \in D \Rightarrow tr \in D$$

A process cannot diverge on termination—if it is divergent after termination, then this must be because it was already divergent.

## Determinism

In sequential programming, determinism is associated with programs which give the same result each time they are executed from the same initial state. This characterization is not entirely appropriate for concurrent programs, since their execution is dependent to some extent on their environment. Furthermore, such components are often non-terminating, since they are designed to provide some ongoing service rather than to perform a specific computation and return a result. Rather than focus on initial and final states, it is necessary to consider the interactions that a process can perform, and whether an environment will achieve the same interactions every time it offers events.

There are three possible results that may be obtained from offering an event to a process: it might accept the event, it might refuse it, or it may diverge and not give a response at all. If the same response is guaranteed every time the offer is made, then the process will be considered to be deterministic on this event. Hence a process will be deterministic if it only ever has one possible response to an offer, after any given previous interaction. This means that if a trace can be extended with some event, then that event cannot also be refused. A process $(F, D, I)$ is *deterministic* if

$$\forall\, tr, a \bullet ((tr \frown \langle a \rangle, \{\}) \in F \Rightarrow (tr, \{a\}) \notin F)$$

It follows from this definition that no divergent process can be deterministic, since after divergence all failures are possible.

Deterministic processes are completely characterized by their traces: once their traces are known, then their failures can be deduced from the fact that they are deterministic, together with (the contrapositive of) the property $F4$. The refusals associated with a trace *tr* will be precisely those events that do not extend *tr*. All the possible infinite traces will be present: the infinite traces will be the limits of all the chains of finite traces.

If $T$ is the set of traces of a deterministic process, then its failures, divergences, and infinite traces will be given by

$$\begin{array}{rcl} F_T & = & \{(tr, X) \mid tr \in T \wedge \forall\, a \in X \bullet tr \frown \langle a \rangle \notin T\} \\ D_T & = & \{\} \\ I_T & = & \{u \mid \forall\, tr \leqslant u \bullet \#tr < \infty \Rightarrow tr \in T\} \end{array}$$

Each process in the traces model corresponds to a deterministic process in the FDI model.

For example, if

$$F = \{(\langle a \rangle^n, X) \mid a \notin X\}$$

then $F$ gives the failures of a process which can only ever perform $a$ events, and can never refuse them. The only trace set $T$ with $F_T \subseteq F$ is

$$T = \{\langle a \rangle^n \mid n \in \mathbb{N}\}$$

Thus $I_T = \{\langle a \rangle^\omega\}$.

## Infinite traces

The relationship between the infinite traces and the finite traces of a process has implications in both directions. Firstly, the presence of an infinite trace $u$ requires that the finite prefixes of $u$, must appear as traces in the set of failures.

$$I1 \qquad u \in I \wedge tr < u \Rightarrow (tr, \{\}) \in F$$

For example, if $\langle a \rangle^\omega \in I$, then $(\langle a \rangle^n, \{\}) \in F$ for any $n$.

Secondly, the presence of some infinite traces may be deduced from failures information about sequences of events that can be forced from a process.

The *closure* $\overline{(F, D, I)}$ of a set of behaviours $(F, D, I)$ includes all the infinite traces that are consistent with the finite traces which appear in the refusal set.

$$\overline{(F, D, I)} \quad = \quad (F, D, \{u \mid \forall\, tr \leqslant u \bullet (tr, \{\}) \in F\})$$

A set of behaviours is *closed* if it is equal to its closure.

If a process is closed then that means that any set of finite approximations to an infinite trace could have come from the same execution. Not all processes are closed; for instance, those with infinite choice such as $\bigsqcap_{i \in \mathbb{N}} P_n$ are able to perform arbitrary length finite sequences of $a$ events, but no infinite sequences of $a$'s. (Recall that $P_0 = STOP$, and $P_{i+1} = a \rightarrow P_i$.)

In this case the $a$ can be refused at any stage, so an infinite sequence cannot be forced. Deterministic processes are closed. This means that their infinite traces $I_T$ can be deduced from their set of finite traces $T$.

Any process $P$ has a deterministic refinement, since in principle all of its internal choices can be resolved in advance of execution. Hence at any point of a process' execution, having exhibited a trace $tr$ and reached a process state $P$, there is a deterministic refinement of $P$ with set of traces $T$.

$$\begin{aligned} I2 \qquad & \forall (tr, X) \in F, \exists\, T \bullet \\ & \quad \{(tr \frown tr', X') \mid (tr', X') \in F_T\} \subseteq F \wedge \{tr \frown u \mid u \in I_T\} \subseteq I \end{aligned}$$

This property allows the inference of infinite traces from finite ones, since it asserts that a particular set of infinite traces are contained in $I$. For example, a process whose set of failures is $\{(\langle a \rangle^n, X) \mid n \in \mathbb{N} \wedge a \notin X\}$ has the set $T = \{\langle a \rangle^n \mid n \in \mathbb{N}\}$ as the only possible deterministic refinement (after the empty trace). If it has any fewer traces, then the set $F_T$ will allow the refusal of $a$ at some point, and this contradicts $I2$'s requirement that $F_T \subseteq F$. Since it has all possible traces of $a$, the set $I_T = \{\langle a \rangle^\omega\}$, and so $\langle a \rangle^\omega$ must be a trace of $I$. This follows purely from the set of finite failures and $I2$.

There are different degrees of nondeterminism that it is useful to distinguish. Processes that only ever make internal choices between finitely many alternatives are said to be *finitely nondeterministic*, and they will be characterized by their finite behaviours. This means that once their finite traces and divergences are known, and their refusal behaviour on finite sets, then their remaining behaviours can be derived. Such processes will be closed: their infinite traces will be the limits of all the infinite chains of their traces. Furthermore, their refusal behaviour on infinite sets will be deduced from the refusal behaviour on finite sets. If they are able to refuse all finite subsets of a set, then they will also be able to refuse the entire set. Such a property is called *compactness*: a process $(F, D, I)$ is said to be *compact* if

$$(\forall X' \subseteq^{\mathit{fin}} X \bullet (tr, X') \in F) \Rightarrow (tr, X) \in F$$

Processes which are both closed and compact correspond to finitely nondeterministic processes. Together with closedness, a property *FFN* characterises finite nondeterminism:

$$\forall (tr, \{\}) \in F, \exists\, FIN \in \mathbb{F}\,(\mathbb{P}(\Sigma^{\checkmark})) \bullet (tr, X) \in F \Leftrightarrow \exists\, Y \in FIN \bullet X \subseteq Y$$

This states that after any trace $tr$, there are only a finite number of maximal refusals, given by the finite set *FIN*. All refusals of the process $(F, D, I)$ after $tr$ must be contained in one of these refusals.

Processes that are not closed, or which do not have the property *FFN*, are *infinitely nondeterministic*: at some stage they will contain some internal choice over an infinite number of possibilities.

The property *FFN* is stronger than compactness, so any process which exhibits only finite nondeterminism will be compact.

For example, the process $\bigsqcap_{i \in \mathbb{N}} (i \to STOP)$ will internally choose a single number $i$ to offer, and will refuse all others. This process can refuse any finite subset of $\mathbb{N}$, but will be unable to refuse all of $\mathbb{N}$, so it is not compact, and hence not *FFN*. However, it is closed.

On the other hand, the process $\bigsqcap_{i \in \mathbb{N}} P_i$ described earlier, where

$$\begin{aligned} P_0 &= STOP \\ P_{n+1} &= a \to P_n \end{aligned}$$

does meet *FN*, since after any trace it will always be able to refuse everything apart from $a$, and may refuse $a$ as well. In this case, $FIN = \{\Sigma^{\checkmark}\}$. However, it is not closed, since it can perform any finite number of $a$ events, but not an infinite sequence. This process is also infinitely nondeterministic.

## 8.2 PROCESS SEMANTICS

Each CSP process expression will be associated with appropriate failures, divergences, and infinite traces in the FDI model. These are defined compositionally, so the behaviours associated with a composite process will be defined in terms of the behaviours of its components. The failures associated with a CSP process expression $P$ will be given by $\mathcal{F}[\![P]\!]$, the divergences by $\mathcal{D}[\![P]\!]$, and the infinite traces by $\mathcal{I}[\![P]\!]$.

There is a close relationship between the traces semantics, the stable failures semantics, and the FDI semantics of any process expression $P$ which does not have any divergences. In this case, the set of traces predicted by the trace semantics $\mathsf{traces}(P)$ is the same as the traces appearing in the failures $\mathcal{F}[\![P]\!]$ of $P$:

$$\mathcal{D}[\![P]\!] = \{\} \quad \Rightarrow \quad \{tr \mid (tr, \{\}) \in \mathcal{F}[\![P]\!]\} = \mathsf{traces}(P)$$

Any trace appearing with some refusal set in $\mathcal{F}[\![P]\!]$ will also appear with the empty refusal set, by property $F3$, so the set of traces appearing with the empty refusal is precisely the set of traces appearing in the failures set.

Furthermore, the set of stable failures predicted in the stable failures is the same as the failures given in the FDI model:

$$\mathcal{D}[\![P]\!] = \{\} \quad \Rightarrow \quad \mathcal{F}[\![P]\!] = \mathcal{SF}[\![P]\!]$$

The definitions of $\mathcal{F}[\![P]\!]$ generally correspond to the definitions of $\mathcal{SF}[\![P]\!]$ with an extra clause which includes the additional failures introduced as a result of divergences of $P$.

## *STOP*

The process *STOP* is a deadlocked process. It is able to perform nothing and can refuse anything. It has no infinite traces (or finite traces apart from $\langle\rangle$), and it does not diverge.

$$\begin{aligned}\mathcal{F}[\![STOP]\!] &= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}\\ \mathcal{D}[\![STOP]\!] &= \{\}\\ \mathcal{I}[\![STOP]\!] &= \{\}\end{aligned}$$

### Prefixing

In a failure of the process $a \to P$, there are two possibilities: either the event $a$ has not occurred, in which case the trace must be $\langle\rangle$ and any events other than $a$ can be refused, or else the event $a$ has occurred and the rest of the failure derives from process $P$.

$$\begin{aligned}\mathcal{F}[\![a \to P]\!] &= \{(\langle\rangle, X) \mid a \notin X\}\\ &\quad \cup\\ &\quad \{(\langle a\rangle \frown tr, X) \mid (tr, X) \in \mathcal{F}[\![P]\!]\}\end{aligned}$$

It does not diverge initially, so any divergence will be a divergence of $P$, prefixed with the initial $a$:

$$\mathcal{D}[\![a \to P]\!] = \{\langle a\rangle \frown tr \mid tr \in \mathcal{D}[\![P]\!]\}$$

Its infinite traces will be those of $P$, prefixed with the initial $a$:

$$\mathcal{I}[\![a \to P]\!] = \{\langle a\rangle \frown u \mid u \in \mathcal{I}[\![P]\!]\}$$

Prefixing preserves determinism: if $P$ is deterministic, then so too is $a \to P$.

### Prefix choice

A failure of the process $x : A \to P(x)$ is again one of two possibilities. Either no event has yet occurred, in which case no event from $A$ can be refused; or else an event $a$ in $A$ has occurred, and the subsequent behaviour is that of the corresponding process $P(a)$.

$$\begin{aligned}\mathcal{F}[\![x : A \to P(x)]\!] &= \{(\langle\rangle, X) \mid A \cap X = \{\}\}\\ &\quad \cup\\ &\quad \{(\langle a\rangle \frown tr, X) \mid a \in A \wedge (tr, X) \in \mathcal{F}[\![P(a)]\!]\}\end{aligned}$$

The prefix choice does not initially diverge, so any divergence will arise from behaviours of the $P(a)$ components; and infinite traces will also be generated by the $P(a)$ processes.

$$\begin{array}{lcl}
\mathcal{D}\,[\![x : A \rightarrow P(x)]\!] & = & \{a \frown tr \mid a \in A \land tr \in \mathcal{D}\,[\![P(a)]\!]\} \\
\mathcal{I}\,[\![x : A \rightarrow P(x)]\!] & = & \{a \frown u \mid a \in A \land u \in \mathcal{I}\,[\![P(a)]\!]\}
\end{array}$$

Prefix choice also preserves determinism.

## *SKIP*

The atomic process *SKIP* is used to denote successful termination, and it signals this by means of the termination event ✓. This is the only event it can perform. It cannot diverge, and it has no infinite traces. All other events will be refused before termination, and all events will be refused after termination.

$$\begin{array}{lcl}
\mathcal{F}\,[\![SKIP]\!] & = & \{(\langle\rangle, X) \mid \checkmark \notin X\} \\
 & & \cup\,\{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\
\mathcal{D}\,[\![SKIP]\!] & = & \{\} \\
\mathcal{I}\,[\![SKIP]\!] & = & \{\}
\end{array}$$

All of the laws given earlier concerning the behaviour of *SKIP* in parallel combinations remain valid in the FDI model.

## *DIV*

The process that can immediately diverge, and hence provides no guarantees about any behaviour, is denoted *DIV*. This process has all possible failures, divergences, and infinite traces.

$$\begin{array}{lcl}
\mathcal{F}\,[\![DIV]\!] & = & \{(tr, X) \mid tr \in TRACE \land X \subseteq \Sigma^{\checkmark}\} \\
 & = & TRACE \times \mathbb{P}(\Sigma^{\checkmark}) \\
\mathcal{D}\,[\![DIV]\!] & = & TRACE \\
\mathcal{I}\,[\![DIV]\!] & = & ITRACE
\end{array}$$

## *CHAOS*

The process which can do absolutely anything except diverge is *CHAOS*. This is able to accept or refuse any events, but it is at least guaranteed to stabilize. It has all possible failures and infinite traces, but no divergences.

$$\begin{aligned}
\mathcal{F}\,[\![\mathit{CHAOS}]\!] &= \mathit{TRACE} \times \mathbb{P}(\Sigma^{\checkmark}) \\
\mathcal{D}\,[\![\mathit{CHAOS}]\!] &= \{\} \\
\mathcal{I}\,[\![\mathit{CHAOS}]\!] &= \mathit{ITRACE}
\end{aligned}$$

Chaotic behaviour may be restricted to a particular set of events $A \subseteq \Sigma^{\checkmark}$. The process $\mathit{CHAOS}_A$ allows any events in the set $A$ to be performed or refused, but cannot perform any events outside the set $A$.

$$\begin{aligned}
\mathcal{F}\,[\![\mathit{CHAOS}_A]\!] &= \{(tr, X) \mid \sigma(tr) \subseteq A\} \\
\mathcal{D}\,[\![\mathit{CHAOS}_A]\!] &= \{\} \\
\mathcal{I}\,[\![\mathit{CHAOS}_A]\!] &= \{u \mid \sigma(u) \subseteq A\}
\end{aligned}$$

### *RUN*

Both *CHAOS* and *DIV* have the same traces as *RUN*, so there was no need to introduce them in the traces model. In the FDI model, it is *RUN* that is the best behaved, always willing to interact, and never refusing any interaction, before termination.

$$\begin{aligned}
\mathcal{F}\,[\![\mathit{RUN}]\!] &= \{(tr, X) \mid X = \{\} \lor \checkmark \in \sigma(tr)\} \\
\mathcal{D}\,[\![\mathit{RUN}]\!] &= \{\} \\
\mathcal{I}\,[\![\mathit{RUN}]\!] &= \mathit{ITRACE}
\end{aligned}$$

This process is deterministic.

The process $\mathit{RUN}_A$ parameterized by a particular set $A$ is able to perform events in that set, and to refuse all others.

$$\begin{aligned}
\mathcal{F}\,[\![\mathit{RUN}_A]\!] &= \{(tr, X) \mid \sigma(tr) \subseteq A \land (X \cap A = \{\} \lor \checkmark \in \sigma(tr))\} \\
\mathcal{D}\,[\![\mathit{RUN}_A]\!] &= \{\} \\
\mathcal{I}\,[\![\mathit{RUN}_A]\!] &= \{u \mid \sigma(u) \subseteq A\}
\end{aligned}$$

If $\checkmark \notin A$ then $\mathit{RUN}_A$ cannot terminate.

### External choice

An observer of the choice construct $P_1 \;\Box\; P_2$ might observe an execution of $P_1$, or of $P_2$; there are no other possibilities. Before any events are performed and the choice resolved, any refused set must be refused by both $P_1$ and $P_2$, unless the choice has already diverged in which case any refusal is possible. After the choice is resolved, any refusal need only be possible for the process in whose favour the choice was resolved.

$$\begin{array}{rcl} \mathcal{F}[\![P_1 \mathbin{\Box} P_2]\!] & = & \{(\langle\rangle, X) \mid ((\langle\rangle, X) \in \mathcal{F}[\![P_1]\!] \cap \mathcal{F}[\![P_2]\!]) \vee \langle\rangle \in \mathcal{D}[\![P_1 \mathbin{\Box} P_2]\!]\} \\ & & \cup \\ & & \{(tr, X) \mid tr \neq \langle\rangle \wedge (tr, X) \in \mathcal{F}[\![P_1]\!] \cup \mathcal{F}[\![P_2]\!]\} \end{array}$$

The divergences and infinite traces of a choice are simply the unions of the component behaviours.

$$\begin{array}{rcl} \mathcal{D}[\![P_1 \mathbin{\Box} P_2]\!] & = & \mathcal{D}[\![P_1]\!] \cup \mathcal{D}[\![P_2]\!] \\ \mathcal{I}[\![P_1 \mathbin{\Box} P_2]\!] & = & \mathcal{I}[\![P_1]\!] \cup \mathcal{I}[\![P_2]\!] \end{array}$$

The properties of idempotence, associativity, and commutativity still hold for external choice in the FDI model. Furthermore, *STOP* is still a unit, though *RUN* is no longer a zero because *P* might diverge whereas *RUN* does not. In the FDI model, the zero of external choice is *DIV*, which has the same traces as *RUN* but minimal guaranteed behaviour.

$$P \mathbin{\Box} DIV =_{FDI} DIV \qquad \langle \Box_{FDI}\text{-zero} \rangle$$

In fact the zero in the stable failures model, *RUN* $\Box$ *DIV*, is also a zero in this model since here it is equivalent to *DIV*.

The executions of the indexed external choice $\Box_{i \in I} P_i$ are the executions of all of its components. Its failures, divergences, and infinite traces will be those of its components:

$$\begin{array}{rcl} \mathcal{F}[\![\Box_{i \in I} P_i]\!] & = & \{(\langle\rangle, X) \mid ((\langle\rangle, X) \in \bigcap_{i \in I} \mathcal{F}[\![P_i]\!]) \vee \langle\rangle \in \mathcal{D}[\![\Box_{i \in I} P_i]\!]\} \\ & & \cup \\ & & \{(tr, X) \mid tr \neq \langle\rangle \wedge (tr, X) \in \bigcup_{i \in I} \mathcal{F}[\![P_i]\!]\} \\ \mathcal{D}[\![\Box_{i \in I} P_i]\!] & = & \bigcup_{i \in I} \mathcal{D}[\![P_i]\!] \\ \mathcal{I}[\![\Box_{i \in I} P_i]\!] & = & \bigcup_{i \in I} \mathcal{I}[\![P_i]\!] \end{array}$$

In the case where the choice is over the empty set of processes, the intersection $\bigcap_{i \in I} \mathcal{F}[\![P_i]\!]$ is taken to include all possible failures, since all of them are vacuously in each of the $\mathcal{F}[\![P_i]\!]$. This means that in this case, any refusal is possible on the empty trace. Furthermore, no events are possible, and there are no divergences or infinite traces. As in the traces model, an empty choice is equivalent to *STOP*.

### Internal choice

The internal choice $P_1 \sqcap P_2$ behaves either as $P_1$ or as $P_2$, and its environment exercises no control over which, at any point. The possible observations are precisely those that either $P_1$ or $P_2$ are able to exhibit.

$$\begin{aligned}
\mathcal{F}[\![P_1 \sqcap P_2]\!] &= \mathcal{F}[\![P_1]\!] \cup \mathcal{F}[\![P_2]\!] \\
\mathcal{D}[\![P_1 \sqcap P_2]\!] &= \mathcal{D}[\![P_1]\!] \cup \mathcal{D}[\![P_2]\!] \\
\mathcal{I}[\![P_1 \sqcap P_2]\!] &= \mathcal{I}[\![P_1]\!] \cup \mathcal{I}[\![P_2]\!]
\end{aligned}$$

The indexed internal choice $\bigsqcap_{i \in J} P_i$ is able to behave as any of its component processes, and its behaviours will be the union of those of its constituents:

$$\begin{aligned}
\mathcal{F}[\![\bigsqcap_{i \in J} P_i]\!] &= \bigcup_{i \in J} \mathcal{F}[\![P_i]\!] \\
\mathcal{D}[\![\bigsqcap_{i \in J} P_i]\!] &= \bigcup_{i \in J} \mathcal{D}[\![P_i]\!] \\
\mathcal{I}[\![\bigsqcap_{i \in J} P_i]\!] &= \bigcup_{i \in J} \mathcal{I}[\![P_i]\!]
\end{aligned}$$

Indexed internal choice over an infinite set is one of the few operators which can introduce infinite nondeterminism into a process description.

### Alphabetized Parallel

An alphabetized parallel combination $P_1 \;{}_A\|_B\; P_2$ consists of $P_1$ performing events in $A^{\checkmark}$, and $P_2$ performing events in $B^{\checkmark}$. Processes $P_1$ and $P_2$ synchronize on events in $(A \cap B)^{\checkmark}$, and perform the other events independently.

The definition of the failures of a parallel combination resembles that of the stable failures model, though divergences must also be included:

$$\begin{aligned}
\mathcal{F}[\![P_1 \;{}_A\|_B\; P_2]\!] = \; & \{(tr, X) \mid \; \exists X_1, X_2 : \mathbb{P}(\Sigma^{\checkmark}) \bullet \\
& \qquad X \cap (A \cup B)^{\checkmark} = (X_1 \cap A^{\checkmark}) \cup (X_2 \cap B^{\checkmark}) \\
& \qquad \wedge (tr \upharpoonright A^{\checkmark}, X_1) \in \mathcal{F}[\![P_1]\!] \\
& \qquad \wedge (tr \upharpoonright B^{\checkmark}, X_2) \in \mathcal{F}[\![P_2]\!]) \\
& \qquad \wedge \sigma(tr) \subseteq (A \cup B)^{\checkmark} \} \\
& \cup \{(tr, X) \mid tr \in \mathcal{D}[\![P_1 \;{}_A\|_B\; P_2]\!]\}
\end{aligned}$$

Failures will also be present as a result of divergence of the combination.

When one of the components has reached a divergent state, then the entire combination is divergent. In order to reach a divergent state, co-operation may be required from the other component, though once the divergence is reached then no further co-operation is required.

$$\begin{array}{lcll} \mathcal{D}[\![P_1 \ {}_A\|_B \ P_2]\!] & = & \{tr \frown tr' \mid & \sigma(tr) \subseteq (A \cup B)^{\checkmark}) \wedge tr' \in \mathit{TRACE} \wedge \\ & & & (tr \upharpoonright A^{\checkmark} \in \mathcal{D}[\![P_1]\!] \wedge (tr \upharpoonright B^{\checkmark}, \{\}) \in \mathcal{F}[\![P_2]\!] \\ & & & \vee\ tr \upharpoonright B^{\checkmark} \in \mathcal{D}[\![P_2]\!] \wedge (tr \upharpoonright A^{\checkmark}, \{\}) \in \mathcal{F}[\![P_1]\!])\} \end{array}$$

The infinite traces of $P_1 \ {}_A\|_B \ P_2$, apart from those arising as a result of divergence, will be those whose projections onto the interface sets $A^{\checkmark}$ and $B^{\checkmark}$ are behaviours of $P_1$ and $P_2$. The projections could individually be finite, in which case they will appear in the failure of the corresponding process, or infinite, appearing as an infinite trace.

$$\begin{array}{lcll} \mathcal{I}[\![P_1 \ {}_A\|_B \ P_2]\!] & = & \{u \mid & \sigma(u) \subseteq A \cup B \wedge \\ & & & u \downarrow A = \infty \Rightarrow u \upharpoonright A \in \mathcal{I}[\![P_1]\!] \wedge \\ & & & u \downarrow A < \infty \Rightarrow (u \upharpoonright A, \{\}) \in \mathcal{F}[\![P_1]\!] \wedge \\ & & & u \downarrow B = \infty \Rightarrow u \upharpoonright B \in \mathcal{I}[\![P_2]\!] \wedge \\ & & & u \downarrow B < \infty \Rightarrow (u \upharpoonright B, \{\}) \in \mathcal{F}[\![P_2]\!]\} \\ & & \multicolumn{2}{l}{\cup\ \{tr \frown u \mid tr \in \mathcal{D}[\![P_1 \ {}_A\|_B \ P_2]\!]\}} \end{array}$$

All of the laws for the parallel operator given in Figure 4.5, with the exception of the law ||-idempotence, also hold for the FDI model.

If both $P_1$ and $P_2$ are divergence-free, then so too is their parallel combination. Furthermore, if both $P_1$ and $P_2$ are deterministic, then so is their parallel combination: synchronized parallel combination preserves determinism.

### Interleaving

An interleaving of two processes $P_1 \ ||| \ P_2$ executes each component entirely independently of the other. Traces of the combination appear as interleavings of traces of the two component processes. Since they do not synchronize, an event (other than termination) will be refused by the combination only when it is refused by both processes independently—if only one of the processes is able to refuse the event, then the combination will still perform it when offered the opportunity.

$$\begin{array}{lcll} \mathcal{F}[\![P_1 \ ||| \ P_2]\!] & = & \{(tr, X_1 \cup X_2) \mid \exists\, tr_1, tr_2 \bullet & tr \text{ interleaves } tr_1, tr_2 \\ & & & \wedge\ X_1 \upharpoonright \Sigma = X_2 \upharpoonright \Sigma \\ & & & \wedge\ (tr_1, X_1) \in \mathcal{F}[\![P_1]\!] \\ & & & \wedge\ (tr_2, X_2) \in \mathcal{F}[\![P_2]\!]\} \\ & & \multicolumn{2}{l}{\cup\ \{(tr, X) \mid tr \in \mathcal{D}[\![P_1 \ ||| \ P_2]\!]} \end{array}$$

An interleaved combination diverges as soon as one of its components does:

$$\begin{array}{lcll} \mathcal{D}[\![P_1 \ ||| \ P_2]\!] & = & \{tr \frown tr' \mid \exists\, tr_1, tr_2 \bullet & tr \text{ interleaves } tr_1, tr_2 \wedge \\ & & & (tr_1 \in \mathcal{D}[\![P_1]\!] \wedge (tr_2, \langle\rangle) \in \mathcal{F}[\![P_2]\!] \\ & & & \vee\ (tr_2 \in \mathcal{D}[\![P_2]\!] \wedge (tr_1, \langle\rangle) \in \mathcal{F}[\![P_1]\!])\} \end{array}$$

The infinite traces are the infinite interleavings of finite or infinite traces of the two components, provided at least one of the components makes an infinite contribution. Infinite traces may also be present as a consequence of the combination's divergence.

$$\begin{array}{rcl}\mathcal{I}\,[\![P_1 \mathbin{|||} P_2]\!] & = & \{u \mid \ \exists\, u_1, u_2 \bullet \ u \text{ interleaves } u_1, u_2 \\ & & \qquad\qquad \wedge\, u_1 \in \mathcal{I}\,[\![P_1]\!] \wedge u_2 \in \mathcal{I}\,[\![P_2]\!] \\ & & \quad \vee\, \exists\, u_1, tr_2 \bullet \ u \text{ interleaves } u_1, tr_2 \\ & & \qquad\qquad \wedge\, u_1 \in \mathcal{I}\,[\![P_1]\!] \wedge (tr_2, \{\}) \in \mathcal{F}\,[\![P_2]\!] \\ & & \quad \vee\, \exists\, tr_1, u_2 \bullet \ u \text{ interleaves } tr_1, u_2 \\ & & \qquad\qquad \wedge\, (tr_1, \{\}) \in \mathcal{F}\,[\![P_1]\!] \wedge u_2 \in \mathcal{I}\,[\![P_2]\!]\} \\ & & \cup\, \{tr \frown u \mid tr \in \mathcal{D}\,[\![P_1 \mathbin{|||} P_2]\!]\}\end{array}$$

The interleaving condition involving infinite traces is defined as a limit of interleaving on finite traces. If there are three sequences of traces, $\langle tr_i \rangle_{i\in\mathbb{N}}$, $\langle tr_i' \rangle_{i\in\mathbb{N}}$, $\langle tr_i'' \rangle_{i\in\mathbb{N}}$, whose limits are $w$, $w'$, and $w''$ respectively, then $w$ interleaves $w', w''$ if $tr_i$ interleaves $tr_i', tr_i''$ for each $i \in \mathbb{N}$. This definition is applicable both for finite and infinite $w$, $w'$ and $w''$. If all sequences are infinite then it ensures that all of the events in both $w'$ and $w''$ appear in $w$.

The laws given in Figure 4.9 are all true for the FDI model as well, with the exception of $|||$-zero. Although all of the *traces* of $RUN_\Sigma$ will be possible for $P \mathbin{|||} RUN_\Sigma$, if $P$ is divergent then it will be able to refuse arbitrary offers after it has diverged and hence will not be equivalent to $RUN_\Sigma$. The zero for interleaving in the FDI model is the immediately divergent process $DIV$, and the law is

$$P \mathbin{|||} DIV =_{FDI} DIV \qquad \langle |||_{FDI}\text{-zero}\rangle$$

## Interface parallel

The process $P_1 \underset{A}{\|} P_2$ is a combination of synchronous and interleaved parallel, synchronizing on events in the set $A^{\checkmark}$ and interleaving outside that set.

Any failure of the parallel process $P_1 \underset{A}{\|} P_2$ will be a combination of failures of its two components.

$$\begin{array}{rcl}\mathcal{F}\,[\![P_1 \underset{A}{\|} P_2]\!] & = & \{(tr, X_1 \cup X_2) \mid \ \exists\, tr_1, tr_2 \bullet \\ & & \qquad\qquad tr \text{ synch}_A\ tr_1, tr_2) \\ & & \qquad\qquad \wedge\, X_1 \setminus A^{\checkmark} = X_2 \setminus A^{\checkmark} \\ & & \qquad\qquad \wedge\, (tr_1, X_1) \in \mathcal{F}\,[\![P_1]\!] \\ & & \qquad\qquad \wedge\, (tr_2, X_2) \in \mathcal{F}\,[\![P_2]\!]\} \\ & & \cup\, \{(tr, X) \mid tr \in \mathcal{D}\,[\![P_1 \underset{A}{\|} P_2]\!]\}\end{array}$$

Divergences will arise from divergences of either component:

$$\begin{aligned}\mathcal{D}\,[\![P_1 \underset{A}{\|} P_2]\!] \quad = \quad \{tr \frown tr' \mid \exists\, tr_1, tr_2 \bullet\ & tr \ \mathsf{synch}_A\ tr_1, tr_2\ \wedge \\ & (tr_1 \in \mathcal{D}\,[\![P_1]\!] \wedge (tr_2, \{\}) \in \mathcal{F}\,[\![P_2]\!] \\ & \vee\ tr_2 \in \mathcal{D}\,[\![P_2]\!] \wedge (tr_1, \{\}) \in \mathcal{F}\,[\![P_1]\!]\}\end{aligned}$$

Infinite traces will arise from infinite traces of the components, or from divergences:

$$\begin{aligned}\mathcal{I}\,[\![P_1 \underset{A}{\|} P_2]\!] \quad = \quad \{u \mid\ & \exists\, u_1, u_2 \bullet\ u \ \mathsf{synch}_A\ u_1, u_2 \\ & \qquad \wedge\ u_1 \in \mathcal{I}\,[\![P_1]\!] \wedge u_2 \in \mathcal{I}\,[\![P_2]\!] \\ & \vee\ \exists\, u_1, tr_2 \bullet\ u \ \mathsf{synch}_A\ u_1, tr_2 \\ & \qquad \wedge\ u_1 \in \mathcal{I}\,[\![P_1]\!] \wedge (tr_2, \{\}) \in \mathcal{F}\,[\![P_2]\!] \\ & \vee\ \exists\, tr_1, u_2 \bullet\ u \ \mathsf{synch}_A\ tr_1, tr_2 \\ & \qquad \wedge\ (tr_1, \{\}) \in \mathcal{F}\,[\![P_1]\!] \wedge u_2 \in \mathcal{I}\,[\![P_2]\!]\} \\ \cup\ & \{tr \frown u \mid tr \in \mathcal{D}\,[\![P_1 \underset{A}{\|} P_2]\!]\}\end{aligned}$$

The $\mathsf{synch}_A$ operator for an infinite trace $w$ is defined in a similar way to the corresponding operation for interleaving. If there are three sequences of traces, $\langle tr_i \rangle_{i \in \mathbb{N}}$, $\langle tr'_i \rangle_{i \in \mathbb{N}}$, $\langle tr''_i \rangle_{i \in \mathbb{N}}$, whose limits are $w$, $w'$, and $w''$ respectively (where $w'$ and $w''$ can be finite or infinite, though at least one of them will be infinite) then $w \ \mathsf{synch}_A\ w', w''$ if $tr \ \mathsf{synch}_A\ tr'_i, tr''_i$ for each $i \in \mathbb{N}$. The predicate $w \ \mathsf{synch}_A\ w', w''$ will hold for precisely those traces which have an appropriate sequence of approximations.

The laws for interface parallel given in Figure 4.10 all hold in the FDI model with the exception of $\underset{A_T}{\|}$ -zero which does not hold for divergent process, although $P \underset{A}{\|} STOP = STOP$ for any non-divergent process $P$ with $\alpha(P) \subseteq A$. The general zero for interface parallel is $DIV$, since immediate divergence is propagated:

$P \underset{A}{\|} DIV =_{FDI} DIV$ $\langle \underset{A_{FDI}}{\|}$ -zero$\rangle$

## Hiding

The process $P \setminus A$ will undergo the same executions as $P$, but with all events in the set $A$ as internal events rather than external synchronizations. This means that any stable refusal $X$ of $P \setminus A$ will correspond to a refusal of $P$ in which not only internal events but also all events in $A$ are refused. The failures of $P \setminus A$ are constructed around this possibility:

$$\begin{aligned}\mathcal{F}\,[\![P \setminus A]\!] \quad = \quad & \{(tr \setminus A, X) \mid (tr, X \cup A) \in \mathcal{F}\,[\![P]\!]\} \\ & \cup\ \{(tr, X) \mid tr \in \mathcal{D}\,[\![P \setminus A]\!]\}\end{aligned}$$

Failures arising as a result of divergence must also be included in the failure set.

The process $P \setminus A$ may diverge because $P$ does, but the abstraction of $A$ may also result in some fresh divergent behaviour. If $P$ may perform an infinite sequence of events from the set $A$, then once those events are internalized the process $P$ is able to perform a divergent trace. The infinite traces of $P$ contain the requisite information.

$$\begin{aligned}\mathcal{D}[\![P \setminus A]\!] &= \{(tr \setminus A) ^\frown tr' \mid tr \in \mathcal{D}[\![P]\!]\}\\ &\quad \cup \{(u \setminus A) ^\frown tr' \mid u \in \mathcal{I}[\![P]\!] \wedge \#(u \setminus A) < \infty\}\end{aligned}$$

If the trace $u \setminus A$ is finite when $u \in \mathcal{I}[\![P]\!]$ is infinite, then $u$ must end with an infinite sequence of events from $A$, and this becomes a divergent sequence of $P \setminus A$. Hiding is the only operator, apart from recursion, that is able to introduce a divergence when applied to a non-divergent process.

The infinite traces of $P \setminus A$ are those infinite traces of $P$ that are still infinite when $A$ is hidden:

$$\mathcal{I}[\![P \setminus A]\!] = \{u \setminus A \mid u \in \mathcal{I}[\![P]\!] \wedge \#(u \setminus A) = \infty\}$$

All of the laws for hiding given in Figure 4.12 are also true in the FDI model.

Together with infinite choice and infinite-to-one alphabet renaming, hiding of an infinite set is one of the few ways in which infinite nondeterminism can be introduced into a process description. For example, the process $CH$ defined below offers the choice of any natural number, and then performs that number of $a$ events before stopping. This process is deterministic, and so finitely nondeterministic.

$$\begin{aligned}CH &= n : \mathbb{N} \rightarrow P_n\\ P_0 &= STOP\\ P_{n+1} &= a \rightarrow P_n\end{aligned}$$

If all of the initial events $\mathbb{N}$ are hidden, then the choice becomes internal, and the resulting process $CH \setminus \mathbb{N} = \bigsqcap_{i \in \mathbb{N}} P_i$ is infinitely nondeterministic: it can perform any finite sequence of $a$ events, but not an infinite sequence, so it is not closed.

EXAMPLE 8.4 Consider a one-time transmission process which polls two input channels repeatedly until it receives an input, upon which it outputs the value received and terminates. This might be described as follows:

$$\begin{aligned}POLL1 = \;& in_1 \rightarrow out \rightarrow STOP\\ & \Box\; switch \rightarrow \; in_2 \rightarrow out \rightarrow STOP\\ & \qquad\qquad\quad \Box\; switch \rightarrow POLL1\end{aligned}$$

Its FDI semantics will be:

$$\begin{array}{rcl}
\mathcal{F}[\![POLL1]\!] & = & \{(tr, X) \mid \ \exists\, n \bullet tr = \langle switch \rangle^{2n} \wedge \{switch, in_1\} \cap X = \{\} \\
& & \qquad \vee \exists\, n \bullet tr = \langle switch \rangle^{2n+1} \wedge \{switch, in_2\} \cap X = \{\} \\
& & \qquad \vee \exists\, n \bullet tr = \langle switch \rangle^{2n} \frown \langle in_1 \rangle \wedge out \notin X \\
& & \qquad \vee \exists\, n \bullet tr = \langle switch \rangle^{2n+1} \frown \langle in_2 \rangle \wedge out \notin X \\
& & \qquad \vee \exists\, n \bullet tr = \langle switch \rangle^{2n} \frown \langle in_1, out \rangle \\
& & \qquad \vee \exists\, n \bullet tr = \langle switch \rangle^{2n+1} \frown \langle in_2, out \rangle\} \\
\mathcal{D}[\![POLL1]\!] & = & \{\} \\
\mathcal{I}[\![POLL1]\!] & = & \{\langle switch \rangle^{\omega}\}
\end{array}$$

If the *switch* between channels is abstracted, the result is $POLL1 \setminus \{switch\}$. This process can perform an infinite sequence of internal *switch* events from its initial state, so it is immediately divergent. This is reflected in the semantics by the fact that the infinite trace of $POLL1$ becomes the empty trace when *switch* is hidden:

$$\begin{array}{l}
\mathcal{D}[\![POLL1 \setminus \{switch\}]\!] = \\
\qquad \{(\langle switch \rangle^{\omega}) \setminus \{switch\} \frown tr' \mid \#(\langle switch \rangle^{\omega} \setminus \{switch\}) < \infty\} \\
\quad = \quad TRACE
\end{array}$$

Therefore $POLL1 \setminus \{switch\}$ also contains all infinite traces, and all possible failures, and is thus equivalent to $DIV$. □

## Renaming

The forward renamed process $f(P)$ behaves as $P$, except that $f(a)$ can be performed whenever $P$ could have performed $a$. It can refuse a set $X$ if every event that $f$ maps into $X$ can be refused by $P$. This means that $f^{-1}(X)$ must be a refusal of $P$ for $X$ to be a refusal of $f(P)$.

$$\begin{array}{rcl}
\mathcal{F}[\![f(P)]\!] & = & \{(f(tr), X) \mid (tr, f^{-1}(X)) \in \mathcal{F}[\![P]\!]\} \\
& & \cup \{(tr, X) \mid tr \in \mathcal{D}[\![f(P)]\!]\}
\end{array}$$

Failures arising from divergence are also included.

The divergences will be generated by those divergent traces of $P$, mapped through the renaming function $f$:

$$\mathcal{D}[\![f(P)]\!] \quad = \quad \{f(tr) \frown tr' \mid tr \in \mathcal{D}[\![P]\!]\}$$

Finally, the infinite traces of $f(P)$ will be generated by the infinite traces of $P$, and by the divergences of $f(P)$:

$$\begin{array}{rcl}
\mathcal{I}[\![f(P)]\!] & = & \{f(u) \mid u \in \mathcal{I}[\![P]\!]\} \\
& & \cup \{tr \frown u \mid tr \in \mathcal{D}[\![f(P)]\!]\}
\end{array}$$

Infinite-to-one renaming is one of the few operators that can introduce infinite nondeterminism from a finitely nondeterministic process. For example, the function $f$ which maps each number $n \in \mathbb{N}$ to the same event $b$, and $f(a)$ to $a$, might be applied to the process $CH$ described above. The result $f(CH)$ is a process for which $b$ followed by any finite number of $a$ events is a possible trace of $f(P)$, but $b$ followed by an infinite number of $a$'s is not. The application of $f$ has introduced infinite nondeterminism, since the resulting process $f(P)$ is not closed.

Finite-to-one renaming cannot introduce infinite nondeterminism, though it might introduce some nondeterminism even when applied to a deterministic process. However, if $f$ is a one-one renaming, then it will preserve determinism.

The renaming operator in the FDI model meets all of the laws given in Figure 4.13.

The backward renaming operator $f^{-1}(P)$ also behaves in a similar fashion to $P$, but any event $a$ in $f^{-1}(P)$ corresponds to an event $f(a)$ in $P$.

$$\begin{aligned}\mathcal{F}[\![f^{-1}(P)]\!] &= \{(tr, X) \mid (f(tr), f(X)) \in \mathcal{F}[\![P]\!])\} \\ &\quad \cup \{(tr, X) \mid tr \in \mathcal{D}[\![f^{-1}(P)]\!]\}\end{aligned}$$

The divergences of $f^{-1}(P)$ will be generated by the divergences of $P$:

$$\mathcal{D}[\![f^{-1}(P)]\!] = \{tr \mid f(tr) \in \mathcal{D}[\![P]\!]\}$$

The infinite traces of $f^{-1}(P)$ will be generated by the infinite traces of $P$:

$$\mathcal{I}[\![f^{-1}(P)]\!] = \{u \mid f(u) \in \mathcal{I}[\![P]\!]\}$$

All the laws given in Figure 4.13 for backward renaming remain valid.

### Sequential Composition

The sequential composition $P_1;\ P_2$ behaves as $P_1$ until $P_1$ terminates successfully, at which point it passes control to $P_2$. A failure of $P_1;\ P_2$ will arise either from a failure of $P_1$, whose stability means that it also refuses to transfer control to $P_2$, or else from a terminating trace of $P_1$ followed by a failure of $P_2$.

$$\begin{aligned}\mathcal{F}[\![P_1;\ P_2]\!] &= \{(tr, X) \mid (tr, X \cup \{\checkmark\}) \in \mathcal{F}[\![P_1]\!]\} \\ &\quad \cup \{(tr_1 \frown tr_2, X) \mid (\ tr_1 \frown \langle\checkmark\rangle, \{\}) \in \mathcal{F}[\![P_1]\!] \\ &\qquad\qquad \wedge (tr_2, X) \in \mathcal{F}[\![P_2]\!]\} \\ &\quad \cup \{(tr, X) \mid tr \in \mathcal{D}[\![P_1;\ P_2]\!]\}\end{aligned}$$

A divergence of $P_1;\ P_2$ arises either from a divergence of $P_1$, or from a trace of $P_1$ followed by a divergence of $P_2$:

$$\mathcal{D}[\![P_1;\ P_2]\!] = \mathcal{D}[\![P_1]\!] \cup \{tr \frown tr' \mid (tr \frown \langle\checkmark\rangle, \{\}) \in \mathcal{F}[\![P_1]\!] \wedge tr' \in \mathcal{D}[\![P_2]\!]\}$$

An infinite trace of $P_1;\ P_2$ also arises in ways similar to divergences: either from an infinite trace of $P_1$, or else from a trace of $P_1$ followed by an infinite trace of $P_2$:

$$\mathcal{I}\,[\![P_1;\ P_2]\!] \quad = \quad \mathcal{I}\,[\![P_1]\!] \cup \{tr \frown u \mid (tr \frown \langle \checkmark \rangle, \{\}) \in \mathcal{F}\,[\![P_1]\!] \wedge u \in \mathcal{I}\,[\![P_2]\!]\}$$

The same laws for sequential composition are valid in the FDI model as in the stable failures model (see figure 4.14).

### Interrupt

The process $P_1 \bigtriangleup P_2$ executes as $P_1$, but at any stage before termination (or divergence) it can begin executing as $P_2$. Its failures will be given by these behaviours together with those included from divergence.

$$\begin{array}{rcl} \mathcal{F}\,[\![P_1 \bigtriangleup P_2]\!] & = & \{(tr, X) \mid \ (tr, X) \in \mathcal{F}\,[\![P_1]\!] \wedge \\ & & \qquad\qquad (\checkmark \in \sigma(tr) \vee (\langle\rangle, X) \in \mathcal{F}\,[\![P_2]\!])\} \\ & & \cup\, \{(tr_1 \frown tr_2, X) \mid \ (tr_1, \{\}) \in \mathcal{F}\,[\![P_1]\!] \wedge \checkmark \notin \sigma(tr_1) \\ & & \qquad\qquad \wedge\, (tr_2, X) \in \mathcal{F}\,[\![P_2]\!] \\ & & \cup\, \{(tr, X) \mid tr \in \mathcal{D}\,[\![P_1 \bigtriangleup P_2]\!]\} \end{array}$$

The divergences are either divergences of $P_1$, or else traces of $P_1$ followed by divergences of $P_2$:

$$\begin{array}{rcl} \mathcal{D}\,[\![P_1 \bigtriangleup P_2]\!] & = & \mathcal{D}\,[\![P_1]\!] \cup \{tr_1 \frown tr_2 \mid (\ tr_1, \{\}) \in \mathcal{F}\,[\![P_1]\!] \\ & & \qquad\qquad \wedge\, \checkmark \notin \sigma(tr_1) \\ & & \qquad\qquad \wedge\, tr_2 \in \mathcal{D}\,[\![P_2]\!]\} \end{array}$$

Similarly, the infinite traces are either infinite traces of $P_1$, or else finite non-terminating traces of $P_1$ followed by infinite traces of $P_2$:

$$\begin{array}{rcl} \mathcal{I}\,[\![P_1 \bigtriangleup P_2]\!] & = & \mathcal{I}\,[\![P_1]\!] \cup \{tr_1 \frown tr_2 \mid \ (tr_1, \{\}) \in \mathcal{F}\,[\![P_1]\!] \\ & & \qquad\qquad \wedge\, \checkmark \notin \sigma(tr) \\ & & \qquad\qquad \wedge\, tr_2 \in \mathcal{I}\,[\![P_2]\!]\} \end{array}$$

All of the laws concerning the interrupt operator that are presented in Figure 4.15 are also true in the FDI model.

## 8.3 RECURSION

The understanding of recursion in the FDI model requires the same operational treatment of recursion as given for the stable failures model in Chapter 6: that recursions unwind via an internal $\tau$ event.

The failures, divergences, and infinite traces associated with recursively defined process expressions $N = P$ can be obtained directly from the operational semantics, or alternatively by using the denotational semantics. Both of these approaches give the same result.

A recursive definition $N = P$ defines the process $N$ in terms of a process description which may itself contain instances of $N$. The FDI model provides guarantees that any recursive definition equation has a solution. It also provides a way of determining the failures, divergences, and infinite traces of the appropriate solution.

The FDI model is concerned with the guarantees that can be made regarding process behaviour. This means that the more possible behaviours a process has associated with it, the less can be guaranteed about it in any particular context. The process *DIV* has the most possible behaviours of any process, and as a result nothing can be guaranteed about how it will execute. More generally, any process of the form $P_1 \sqcap P_2$ will have more behaviours than $P_1$ alone, and so less can be guaranteed about how it will execute.

As in the traces model, the refinement ordering $(F_1, D_1, I_1) \sqsubseteq_{FDI} (F_2, D_2, I_2)$ between processes holds when the second process has fewer possible behaviours than the first.

$$(F_1, D_1, I_1) \sqsubseteq_{FDI} (F_2, D_2, I_2) \quad \Leftrightarrow \quad F_2 \subseteq F_1 \land D_2 \subseteq D_1 \land I_2 \subseteq I_1$$

The subscript *FDI* signifies that the relationship is defined on the FDI model.

Refinement holds between two process expressions $P_1$ and $P_2$ whenever it holds between their sets of failures, divergences, and infinite traces. Another way of characterizing the relationship $P_1 \sqsubseteq_{FDI} P_2$ is as $P_1 =_{FDI} P_1 \sqcap P_2$. The introduction of the behaviours of $P_2$ does not introduce any new behaviours to $P_1$. The subscript *FDI* will be elided if it is clear from the context.

In the FDI model, refinement amounts to reducing nondeterminism in processes. The minimal process with respect to this ordering, refined by all other processes, is *DIV*. The maximal processes in the refinement ordering will be the deterministic processes: no deterministic process can be further refined. Unlike the traces model and the stable failures model, there is no single maximal process.

Any recursive CSP equation $N = P$ will have a least fixed point: a solution which is refined by any other solutions that might exist. The least solution provides the fewest guarantees, and all guarantees that it does provide are also true for any of the other solutions. It is appropriate to use the least solution of $N = P$ as the semantics of $N$, since the only observations that can be guaranteed of $N$ will be those that follow from the fact that it is a solution of the equation $N = P$. In contrast to the approaches taken in the traces and stable failures model, approximation in the FDI model begins with as many behaviours as possible, and only excludes those behaviours whose absence is guaranteed by unfolding the recursive definition.

EXAMPLE 8.5 The recursive equation $N = N \Box a \to STOP$ has many fixed points in the FDI model, including $a \to STOP$, $a \to STOP \Box b \to STOP$, and *DIV*. The least of these in the FDI model is *DIV*, and so this will be the semantics of the process defined by the recursive equation. The equation does not exclude the possibility of initial divergence, so that possibility must be allowed, resulting in no guarantees at all concerning useful behaviour. The possibility of divergence leads to its semantics being different to its stable failures semantics given in Example 6.7. □

The process $P$ with free variable $N$ corresponds to a function $F(Y) = P[Y/N]$, and successive applications of the function $F$ will give rise to approximations to the fixed point. The first approximation is the weakest process of all, *DIV*, and successive approximations are $F^n(DIV)$ for $n \in \mathbb{N}$. Each of these will be refined by any fixed point, since if $N = F(N)$, then

- $DIV \sqsubseteq_{FDI} N$; and
- if $F^n(DIV) \sqsubseteq_{FDI} N$ then $F(F^n(DIV)) \sqsubseteq_{FDI} F(N) = N$ because all of the CSP operators comprising $F$ are monotonic with respect to the refinement order $\sqsubseteq$.

If the function $F$ does not introduce any infinite nondeterminism—in other words $P$ does not contain any infinite internal choice, infinite hiding, or infinite-to-one renaming—then the sequence of approximations $\langle F^n(DIV)\rangle_{n\in\mathbb{N}}$ will define the fixed point, which will consist of those failures, divergences, and infinite traces that are in the behaviours of all elements of the sequence. The fixed point will then be given by the triple:

$$\left(\bigcap_{n\in\mathbb{N}} \mathcal{F}[\![F^n(DIV)]\!], \bigcap_{n\in\mathbb{N}} \mathcal{D}[\![F^n(DIV)]\!], \bigcap_{n\in\mathbb{N}} \mathcal{I}[\![F^n(DIV)]\!]\right)$$

This process must refine all of the approximations, since it is contained in each of them. Furthermore, any other process which refines all of the approximations will also refine this process, which is therefore the least fixed point of $F$: it will be refined by any other fixed point of $F$.

EXAMPLE 8.6 The process $N = STOP \sqcap a \to N$ is the fixed point of the function $F(Y) = STOP \sqcap a \to Y$. For any $n$, the semantics of $F^{n+1}(DIV)$ can be calculated from the semantics of $F^n(DIV)$, resulting in

$$\begin{array}{rcl}
\mathcal{F}[\![F^n(DIV)]\!] & = & \{(\langle a\rangle^i, X) \mid i < n \wedge X \subseteq \Sigma^{\checkmark}\} \\
 & & \cup\, \{(\langle a\rangle^n \frown tr, X) \mid tr \in TRACE \wedge X \subseteq \Sigma^{\checkmark}\} \\
\mathcal{D}[\![F^n(DIV)]\!] & = & \{\langle a\rangle^n \frown tr \mid tr \in TRACE\} \\
\mathcal{I}[\![F^n(DIV)]\!] & = & \{\langle a\rangle^n \frown u \mid u \in ITRACE\}
\end{array}$$

The only behaviours that are in all of the corresponding sets are those that have only $a$'s in their traces. There is no trace that appears in all of the divergence sets, and the only trace that appears in all the infinite trace sets is the trace $\langle a \rangle^{\omega}$. The intersections of the three components reduces to

$$\begin{array}{rcl} \mathcal{F}[\![N]\!] & = & \{(\langle a \rangle^i, X) \mid i \in \mathbb{N} \wedge X \subseteq \Sigma^{\checkmark}\} \\ \mathcal{D}[\![N]\!] & = & \{\} \\ \mathcal{I}[\![N]\!] & = & \{\langle a \rangle^{\omega}\} \end{array}$$

which is in accordance with the behaviours predicted from the operational semantics.

In fact, the function $F$ has two fixed points:

$$(\{(\langle a \rangle^n, X) \mid n \in \mathbb{N} \wedge X \subseteq \Sigma^{\checkmark}\}, \{\}, \{\})$$

and

$$(\{(\langle a \rangle^n, X) \mid n \in \mathbb{N} \wedge X \subseteq \Sigma^{\checkmark}\}, \{\}, \langle a \rangle^{\omega})$$

Both processes have the same failures: any finite sequence of $a$'s is possible, and any refusal is also possible at any stage. However, one of the fixed points has an infinite sequence of $a$'s possible, whereas the other one does not. If nothing is known about $N$ other than the fact that it satisfies the equation above then it is inappropriate to exclude the infinite trace, since its absence cannot be guaranteed. The appropriate semantics for $N$ defined by the recursive definition $N = STOP \sqcap a \rightarrow N$ will be the second process, with the infinite trace, as calculated. $\Box$

EXAMPLE 8.7 The unguarded process $N = N \, \Box \, a \rightarrow STOP$ corresponds to the function $F(Y) = Y \, \Box \, a \rightarrow STOP$. Since the possibility of divergence is preserved by external choice, it follows from the semantics of the external choice operator that $F(DIV)$ has the same semantics as $DIV$. This means that the entire sequence $\langle F^n(DIV) \rangle_{n \in \mathbb{N}}$ is simply $\langle DIV \rangle_{n \in \mathbb{N}}$, so the limit of this sequence is $DIV$. Hence $N = DIV$, as predicted by the operational semantics. $\Box$

If the function $F$ does contain infinite nondeterminism, then some further work may be required, as the intersection of the approximations might not give the fixed point. If $F$ is guarded, then the intersection will at least give the finite behaviours of the fixed point: the failures and divergences. However, it may be too pessimistic on the infinite traces and further approximations may be required. Hence if $N = F(N)$ for guarded $F$, then the FDI semantics of $N$ satisfy

$$\begin{array}{rcl} \mathcal{F}[\![N]\!] & = & \bigcap_{n \in \mathbb{N}} \mathcal{F}[\![F^n(DIV)]\!] \\ \mathcal{D}[\![N]\!] & = & \bigcap_{n \in \mathbb{N}} \mathcal{D}[\![F^n(DIV)]\!] \\ \mathcal{I}[\![N]\!] & \subseteq & \bigcap_{n \in \mathbb{N}} \mathcal{I}[\![F^n(DIV)]\!] \end{array}$$

EXAMPLE 8.8 The process $N$ defined by

$$\begin{array}{rcl} N &=& (STOP \sqcap a \rightarrow N) \parallel A \\ A &=& \bigsqcap_{n \in \mathbb{N}} A(n) \\ A(0) &=& STOP \\ A(n+1) &=& a \rightarrow A(n) \end{array}$$

is prevented from performing any infinite sequence of $a$'s, though it can perform any arbitrarily long finite sequence of them. Each of the approximations $F^n(DIV)$ have the same behaviours as the approximations in Example 8.6:

$$\begin{array}{rcl} \mathcal{F}\,[\![F^n(DIV)]\!] &=& \{(\langle a\rangle^i, X) \mid i < n \wedge X \subseteq \Sigma^{\checkmark}\} \\ && \cup\,\{(\langle a\rangle^n \frown tr, X) \mid tr \in TRACE \wedge X \subseteq \Sigma^{\checkmark}\} \\ \mathcal{D}\,[\![F^n(DIV)]\!] &=& \{\langle a\rangle^n \frown tr \mid tr \in TRACE\} \\ \mathcal{I}\,[\![F^n(DIV)]\!] &=& \{\langle a\rangle^n \frown u \mid u \in ITRACE\} \end{array}$$

and so the intersection of all the approximations will contain $\langle a\rangle^n$ as an infinite trace. The intersection is in a sense the $\omega$th approximation to the fixed point, and will be denoted $F^\omega(DIV)$. In each approximation $F^n(DIV)$ the infinite trace $\langle a\rangle^\omega$ arises as a result of the divergence $\langle a\rangle^n$.

$$\begin{array}{rcl} \mathcal{F}\,[\![F^\omega(DIV)]\!] &=& \{(\langle a\rangle^i, X) \mid i \in \mathbb{N} \wedge X \subseteq \Sigma^{\checkmark}\} \\ \mathcal{D}\,[\![F^\omega(DIV)]\!] &=& \{\} \\ \mathcal{I}\,[\![F^\omega(DIV)]\!] &=& \{\langle a\rangle^\omega\} \end{array}$$

The limit process has no divergences, and so one further application of $F$ will remove the infinite trace and result in the fixed point (which also happens to be the semantics of the process $A$).

$$\begin{array}{rcl} \mathcal{F}\,[\![N]\!] &=& \{(\langle a\rangle^i, X) \mid i \in \mathbb{N} \wedge X \subseteq \Sigma^{\checkmark}\} \\ \mathcal{D}\,[\![N]\!] &=& \{\} \\ \mathcal{I}\,[\![N]\!] &=& \{\} \end{array}$$

The function $F$ is guarded, and so the sets $\mathcal{F}\,[\![N]\!]$ and $\mathcal{D}\,[\![N]\!]$ must be reached by the $\omega$th approximation, as the intersection of the finite approximations: $\mathcal{F}\,[\![N]\!] = \mathcal{F}\,[\![F^\omega(DIV)]\!]$ and $\mathcal{D}\,[\![N]\!] = \mathcal{D}\,[\![F^\omega(DIV)]\!]$. Only the infinite traces may require more than $\omega$ iterations, but even they must eventually be reached. □

If a function $F$ is not guarded, then the number of approximations required to determine the failures and divergences of the fixed point may be more than $\omega$. (see Exercise 8.12).

Law recursion-unwinding will hold for any recursive definition $N = P$. However, the law UFP does not hold in general, since even guarded recursive definitions can admit more than one fixed point. For example, the (guarded) function $F(Y) = STOP \sqcap a \to Y$ has both the least fixed point calculated in Example 8.6, and the process $A$ of Example 8.8 as solutions to the equation $F(Y) = Y$. However, all solutions to any guarded equation must have the same failures and divergences.

A more restricted unique fixed point law holds in the FDI model. The infinite traces of finitely nondeterministic processes are determined completely by their failures and divergences. This means that if $F(P_1) = P_1$ and $F(P_2) = P_2$ where $P_1$ and $P_2$ are finitely nondeterministic, and $F$ is guarded, then must have exactly the same behaviours: $P_1 =_{FDI} P_2$. The second law for recursion in the FDI model can now be given:

$$\boxed{\begin{array}{ll} (P_1 \text{ finitely nondeterministic} \wedge P_2 \text{ finitely nondeterministic} & \\ \wedge\ F \text{ event guarded} \wedge (F(P_1) =_{FDI} P_1) \wedge (F(P_2) =_{FDI} P_2)) & \\ \qquad\qquad\qquad\qquad \Rightarrow P_1 =_{FDI} P_2 & \langle \mathsf{UFP} \rangle \end{array}}$$

In particular, if a guarded $F$ itself contains no infinite nondeterminism, then the recursively defined process $P_1 = F(P_1)$ will be finitely nondeterministic, and hence equivalent to any other such process $P_2$ for which $P_2 =_{FDI} F(P_2)$.

For example, the finite nondeterminism conditions hold for both $N$ and $P$ of Example 4.24:

$$\begin{array}{rcl} N & = & F(N) \quad = \quad (a \to N) \,\square\, b \to STOP \\ M & = & a \to M \\ P & = & M \triangle (b \to STOP) \end{array}$$

and $P =_{FDI} F(P)$, so it follows that $N =_{FDI} P$.

## Mutual recursion

Mutual recursion generalizes single recursion in the same way as in the models already introduced. The operational transition rule is adjusted in a similar way, modeling the recursive unwinding of any process variable $N_i$ as accompanied by an internal transition. As with the case for single recursion, exactly the same results concerning the traces model remain valid if this transition rule is used instead.

$$\boxed{\frac{}{N_i \xrightarrow{\tau} P_i} \quad [\,\underline{N} = \underline{P}\,]}$$

The failures, divergences, and infinite traces associated with all of the $N_i$ processes will be those that are predicted by the operational semantics. They will give the most nondeterministic processes that satisfy the set of defining equations—the ones with the most failures, divergences, and infinite traces. The theory of CSP guarantees that such processes must exist for any set of recursive CSP definitions.

The results concerning single recursion carry over to the more general case:

- If all of the recursive calls are event guarded, and none of the recursive definitions contains any infinite nondeterminism, then the semantics of the $N_i$ are the intersections of the semantics of the chain of approximations, starting from $DIV$. Each $N_i$ is defined by a function $F_i(\underline{N})$. If the $j$th approximation to $N_i$ is written as $N_i^j$, then each $N_i^0 = DIV$, and each $N_i^{j+1} = F_i(\underline{N}^j)$, where $\underline{N}^j$ is the vector of all of the $j$th approximations. Each approximation $N_i^j$ is associated with failures $\mathcal{F}\,[\![N_i^j]\!]$, divergences $\mathcal{D}\,[\![N_i^j]\!]$, and infinite traces $\mathcal{I}\,[\![N_i^j]\!]$. The limit $N_i$ will have failures, divergences, and infinite traces given by

$$\begin{aligned} \mathcal{F}\,[\![N_i]\!] &= \textstyle\bigcap_{j\in\mathbb{N}} \mathcal{F}\,[\![N_i^j]\!] \\ \mathcal{D}\,[\![N_i]\!] &= \textstyle\bigcap_{j\in\mathbb{N}} \mathcal{D}\,[\![N_i^j]\!] \\ \mathcal{I}\,[\![N_i]\!] &= \textstyle\bigcap_{j\in\mathbb{N}} \mathcal{I}\,[\![N_i^j]\!] \end{aligned}$$

- If all of the recursive calls are event guarded, then the finite behaviours—the failures and divergences—will be given even if infinite nondeterminism is present:

$$\begin{aligned} \mathcal{F}\,[\![N_i]\!] &= \textstyle\bigcap_{j\in\mathbb{N}} \mathcal{F}\,[\![N_i^j]\!] \\ \mathcal{D}\,[\![N_i]\!] &= \textstyle\bigcap_{j\in\mathbb{N}} \mathcal{D}\,[\![N_i^j]\!] \end{aligned}$$

  However, the infinite traces may not be accurate, though they will be contained in the intersection of those of the finite approximations:

$$\mathcal{I}\,[\![N_i]\!] \subseteq \textstyle\bigcap_{j\in\mathbb{N}} \mathcal{I}\,[\![N_i^j]\!]$$

Law recursion-unwinding will hold for any family of mutually recursive definitions. Whenever $N_i = P_i$ appears as a recursive definition, then $N_i =_{FDI} P_i$.

Law UFP also generalizes to mutual recursion. In a mutually recursive definition $\underline{N} = \underline{P}$, a process variable $N_i$ is recursive if it appears in any of the $P_j$. If each process definition $P_i$ associated with any recursive $N_i$ is event guarded in all of the process variables that appear in it, then the recursive definition is event guarded. If two families $\underline{P}$ and $\underline{P}'$ of finitely nondeterministic processes both satisfy the same guarded recursive equation, then they must be equivalent:

$$\boxed{\begin{aligned}&(\underline{P} \text{ finitely nondeterministic and } \underline{P}' \text{ finitely nondeterministic}\\ &\quad \wedge \underline{F} \text{ event guarded} \wedge (\underline{F}(\underline{P}) =_{FDI} \underline{P}) \wedge (\underline{F}(\underline{P}') =_{FDI} \underline{P}'))\\ &\qquad\qquad \Rightarrow \underline{P} =_{FDI} \underline{P}' \qquad\qquad \langle \mathsf{UFP} \rangle\end{aligned}}$$

## 8.4 SPECIFICATION AND VERIFICATION

### Property-oriented specification

The inclusion of refusal, divergence, and infinite trace information in the FDI model allows a wider range of specification to be presented than was possible in the traces model or the stable failures model. Since there are three sets of behaviours associated with any process, the most general form of specification will consist of three parts which each describe the required property of observations from the corresponding behaviour set. A specification $S$ can be written as a triple $(S_F(tr,X), S_D(tr), S_I(u))$.

$$\begin{aligned} P \ \mathbf{sat}\ (S_F(tr,X), S_D(tr), S_I(u)) \quad = \quad & \forall (tr,X) \in \mathcal{F}\,[\![P]\!] \bullet S_F(tr,X)\\ & \wedge \forall\, tr \in \mathcal{D}\,[\![P]\!] \bullet S_D(tr)\\ & \wedge \forall\, u \in \mathcal{I}\,[\![P]\!] \bullet S_I(u)\end{aligned}$$

The most basic form of liveness is divergence-freedom, which requires that at any stage of an execution the process should at least eventually reach a stable state. The appropriate specification $S_D(tr)$ on the divergence set of a process states simply that no divergence should occur. This is captured by requiring that $S_D(tr)$ should be false for any trace $tr$. Divergence-freedom is then the specification

$$\textsf{divergence-free} \quad = \quad (true(tr,X), false(tr), true(u))$$

It is satisfied by any process which has no divergences, and by no process that can diverge at any stage. The most nondeterministic process which satisfies it is *CHAOS*.

EXAMPLE 8.9 A buffer $B(N)$ of size $N$ is guaranteed not to diverge provided it is not overloaded. It provides no guarantees about its behaviour if it is ever supplied with more than $N$ pieces of data. In this case, the specification $S_D(tr)$ that it meets will be

$$S_D(tr) \quad = \quad \exists\, tr' \leqslant tr \bullet tr' \downarrow in > tr' \downarrow out + N$$

This states that any divergence *tr* must begin with a sequence $tr'$ witnessing the fact that too many inputs occurred. In order to guarantee divergence-freedom of a system which contains such a component, the rest of the system will have to ensure that the buffer is never asked to carry more items than its capacity $N$.

Divergence here is associated with a limitation on the capabilities of the component. A manufacturer would be unlikely to provide a component which is sure to diverge when its capacity is exceeded. It is more likely that the manufacturer simply makes no claims about the component in this case. For example, a bridge with a weight limit of 44 tonnes does not claim to support a heavier load: any weight which does collapse the bridge should be heavier than 44 tonnes. □

EXAMPLE 8.10 (DEADLOCK-FREEDOM) Deadlock-freedom is captured by the specification

$$\textsf{strong deadlock-free}(tr, X) \quad = \quad X \neq \Sigma^{\checkmark}$$

Whatever trace has already occurred, the process cannot refuse the entire set of events $\Sigma^{\checkmark}$. A deadlock-free process must therefore be divergence-free, since all events could be refused on divergence.

A weaker form of deadlock-freedom, which allows the possibility of termination is expressed as

$$\textsf{deadlock-free}(tr, X) \quad = \quad \checkmark \notin \sigma(tr) \Rightarrow X \neq \Sigma^{\checkmark}$$

Deadlock-freedom for a process $P$ can be established in the stable failures model provided $P$ can also be shown to be divergence-free in the FDI model, since in that case $\mathcal{SF}\,[\![P]\!] = \mathcal{F}\,[\![P]\!]$. It also follows that if a divergence-free process $P$ **sat** $\textsf{strong deadlock-free}(tr, X)$ in the stable failures model then this must also be true in the FDI model. □

The relationship between the traces model and the FDI model, on divergence-free processes, means that results concerning safety obtained using the traces model can be imported into the FDI model. If $S_T(tr)$ is a predicate on traces, then a corresponding predicate on failures can be defined to hold whenever $S_T$ holds on the trace component: $S_F(tr, X) \Leftrightarrow S_T(tr)$. If $P$ is known to be divergence-free, then the traces model can be used to verify any safety specification, and the result can be imported into the FDI model, since the semantics in the two models give the same traces. In particular, if $P$ **sat** $\textsf{divergence-free}$, then

$$P \textbf{ sat } S_T(tr) \quad \Rightarrow \quad P \textbf{ sat } (S_F(tr, X), \mathit{false}(tr), \mathit{true}(u))$$

This allows reasoning from the traces model and the FDI model to be combined within a single system's analysis.

EXAMPLE 8.11 (BUFFERS) As well as respecting the order on messages passing through the system, and offering to receive input and provide output as appropriate, a buffer should also be divergence-free. This means that the specification of a buffer in the FDI model combines the trace specification and the stable failures specification into a requirement on the FDI failures:

$$\begin{array}{rcl} \mathit{Buff}_F(tr, X) & = & tr \Downarrow out \leqslant tr \Downarrow in \\ & & \wedge\ tr \Downarrow out = tr \Downarrow in \Rightarrow in.T \cap X = \{\} \\ & & \wedge\ tr \Downarrow out < tr \Downarrow in \Rightarrow out.T \nsubseteq X \end{array}$$

The restriction on traces and failures means that any process satisfying this predicate cannot diverge, since any divergence will give rise to failures which violate the specification. □

Specifications on infinite traces will often be concerned with some kind of progress requirement or fairness constraint. For example, any infinite sequence of tosses of a fair coin should contain infinitely many heads and infinitely many tails:

$$u \downarrow heads = \infty \wedge u \downarrow tails = \infty$$

All finite sequences of heads and tails will be permitted in the failures of a process representing such a coin, but the infinite traces will be constrained. Stronger requirements can also be expressed, such as the expectation that the proportion of heads will approach $0.5$ as the length of the trace increases:

$$S_I(u) \quad = \quad \lim_{n \to \infty} (((u \upharpoonright n) \downarrow heads)/n) = 0.5$$

EXAMPLE 8.12 Resource allocation by a scheduler is often subject to fairness requirements with regard to its servicing of requests from various processes. If a request of process $i$ is represented by an event $req_i$, and an allocation to process $i$ is represented by $alloc_i$, then one possible requirement is that no request should be ignored for ever. This may be captured as the specification

$$S_I(u) \quad = \quad \forall\, i \bullet (u \downarrow req_i = \infty \Rightarrow u \downarrow alloc_i = \infty)$$

In order to ensure that processes are not prevented from making their request by another process starving them out with an infinite sequence of resource allocations, it might be appropriate to ensure that the $req_i$ events are entirely under the control of the processes themselves and so cannot be blocked by the scheduler or other processes. □

EXAMPLE 8.13 A communications medium which can sometimes lose messages might be used as the basis for a link between two peers which wish to communicate reliably. They will be able to do this provided the medium can guarantee some form of progress, namely that it can never lose an infinite sequence of messages, and must eventually provide output if a message is input sufficiently often.

The possibility of messages being lost, but not duplicated or reordered, is captured by the trace specification $tr \Downarrow out \preccurlyeq tr \Downarrow in$: the sequence of output messages must be a subsequence of the input messages, so everything that is output was previously input. However, not all inputs are guaranteed to be successfully transmitted.

Liveness might be captured as deadlock-freedom: the medium must always be ready to input or output.

The progress property is then described as

$$S_I(u) \quad = \quad u \downarrow in = \infty \Rightarrow u \downarrow out = \infty$$

Any infinite traces of a process must be consistent with the finite traces, which meet the safety property, so it is not necessary to specify that the outputs of the infinite sequence must be a subsequence of the inputs. This follows from the corresponding property on the finite traces. $\square$

## Admissible specifications

The infinite traces of a process must be consistent with its finite ones, as described by property $I1$. If a process is required to meet a progress or fairness condition, then its infinite traces must be examined and checked against the specification $S_I(u)$. However, specifications are often expressed only in terms of their failures, and no constraints are placed explicitly on the infinite traces except those required by consistency with the constrained finite behaviours. For example, deadlock-freedom is concerned with refusal information after finite traces, and buffer specifications are concerned with maintaining the order between inputs and outputs, and with liveness at finite stages of the execution. Such specifications are easier to check, since they are dependent only on the finite behaviours of processes, and these are often more straightforward to calculate and reason about. They are known as *admissible* specifications, and are equivalent to specifications of the form $(S_F(tr, X), S_D(tr), true(u))$.

## Verification

The semantic equations associated with the CSP operators can support a number of proof rules for reasoning about CSP process descriptions, but in practice the most common form of specification is concerned with divergence-freedom. Once this is established, the stable failures model can be used to analyze the requirements on the failures of the process. This section will be concerned with conditions for establishing divergence-freedom of process descriptions.

Processes can be shown to be divergence-free by calculating their semantics directly, in particular their set of divergences. However, there are some common techniques for making sure that divergence is not introduced at any stage of a process description, resulting in a process that is divergence-free by construction.

The only operators that can introduce divergence into a process description are $DIV$, the hiding operator, and recursion. All of the other operators preserve the property of 'divergence-freedom': if their components satisfy divergence-free then so too will their combination. These three operators will be considered in turn.

### *DIV*

This process is divergent, and any use of it will introduce divergence into the process description. It is therefore best avoided when constructing divergence-free processes.

### Hiding

If $P$ **sat** divergence-free, then a divergence of the process $P \setminus A$ must arise from a trace $u$ of $P$ which ends in an infinite sequence of events from $A$. In order for $P \setminus A$ to satisfy divergence-free it is necessary to ensure that this possibility cannot arise.

This will be guaranteed if any trace $tr \setminus A$ of $P \setminus A$ is associated with some bound on the length that the original trace $tr$ can be. This will ensure that $tr \setminus A$ cannot be generated from any infinite trace of $P$, and so there is no possibility of divergence. The bound associated with a trace $tr \setminus A$ will be given by a bounding function $\beta : TRACE \to \mathbb{N}$: $P$ should satisfy the specification that the length of any of its traces $tr$ must be no greater than the bound $\beta(tr \setminus A)$. This means that there is a bound on the length of all of the traces that could have given rise to $tr \setminus A$. If $P$ is divergence-free, then this may be established in the traces model.

The rule is as follows:

$$\frac{\begin{array}{l} P \textbf{ sat } \text{divergence-free} \\ P \textbf{ sat } \#tr \leqslant \beta(tr \setminus A) \end{array}}{P \setminus A \textbf{ sat } \text{divergence-free}}$$

EXAMPLE 8.14 Consider the process $P$ given by $P = a \to b \to c \to P$ discussed in Example 7.3. The proof rule will be used to establish that $P \setminus \{b\}$ **sat** divergence-free.

The set $\{b\}$ is to be hidden, so it is necessary to find a bounding function. In fact, $\beta(tr') = 2 * \#tr'$ is such a function, in that $P$ **sat** $\#tr \leqslant 2 * \#(tr \setminus \{b\})$—any trace of $P \setminus \{b\}$ of length $n$ must have come from a trace of $P$ of length no greater than $2n$.

The existence of the bounding function yields that $P \setminus \{b\}$ **sat** divergence-free.

□

## 8.5 RECURSION INDUCTION

Recursive definitions have the potential to introduce divergent behaviours, so the possibility of divergence must be addressed within a verification of a recursive process.

If a recursive definition $N = F(N)$ is event guarded, and $F$ preserves divergence-freedom, then $N$ will be divergence-free.

$$\frac{\begin{array}{l}\forall\, Y \bullet (\ Y \ \mathbf{sat}\ \textsf{divergence-free} \\ \qquad\qquad \Rightarrow F(Y)\ \mathbf{sat}\ \textsf{divergence-free})\end{array}}{N\ \mathbf{sat}\ \textsf{divergence-free}} \quad \begin{bmatrix} N = F(N) \\ F \text{ guarded} \end{bmatrix}$$

The event guard means that the unfolding of the recursive definition will not itself introduce a divergent loop. The antecedent that $F$ preserves divergence-free means that $F$ does not introduce any divergences itself. This is required to exclude functions such as

$$F(Y) \quad = \quad a \rightarrow (Y \,|||\, b \rightarrow DIV)$$

which is guarded but does not preserve divergence-free, since $F(STOP)$ has a divergent trace.

EXAMPLE 8.15 The light switch process of Example 7.6 has a guarded recursive definition:

$$LIGHT \quad = \quad on \rightarrow off \rightarrow LIGHT$$

It can be seen to be divergence-free by observing that the function $F(Y) = on \rightarrow off \rightarrow Y$ is guarded, and that it does not introduce divergence since it contains only operators that do not introduce divergence. □

A more general recursion induction rule allows an explicit treatment of divergence. In this case there are no constraints on the form of the recursive function $F$, so in particular it need not be guarded. This rule instead requires that each recursive unfolding of $F$ increases the length of any possible divergence. The fixed point of $F$ thus cannot have any divergence of finite length.

Let $Spec_n = (true(tr, X), \#tr \geqslant n, true(u))$: this specification requires that any divergence is of length at least $n$.

The $Spec_n$ specifications are progressively stronger as $n$ increases, with a limit of divergence-free. In order to check if a process meets a specification $Spec_n$, only those behaviours with traces of length less than $n$ need be considered. So any process will meet any $Spec_0$.

$$\frac{\forall\, n, Y \bullet (Y\ \mathbf{sat}\ Spec_n \Rightarrow F(Y)\ \mathbf{sat}\ Spec_{n+1})}{N\ \mathbf{sat}\ \textsf{divergence-free}} \quad [\, N = F(N)\, ]$$

If recursive calls allow the approximations $Spec_n$ to get closer to divergence-free, then in the limit the specification divergence-free will itself be met by the recursive process.

In order to establish the antecedent, more general proof rules would be required than those given in this section, since there is also a need to consider divergences explicitly. This may be done by a direct appeal to the semantics of the CSP function $F$, or alternatively by developing from the semantics a more detailed collection of rules that handle more general specifications on divergence.

EXAMPLE 8.16 The definition of a buffer whose capacity expands as it accepts messages is given as follows:

$$EXPBUFF \quad = \quad in?x : T \rightarrow ((out!x \rightarrow EXPBUFF) \gg COPY)$$

Each time an input is provided, a fresh buffer *COPY* is spawned and added to the chain, and the input value is passed to it. The definition of the body of the recursion $F(Y) = in?x : T \rightarrow ((out!x \rightarrow Y) \gg COPY)$ is not guarded, since there is an implicit hiding operator within the chaining operator. However, divergence-freedom can be proved by the recursion induction rule for divergence-freedom.

Let $tr$ be a divergence of $Y$ of length at least $n$ giving rise to a divergence $\langle in.v \rangle \frown tr'$ of $F(Y)$. Then $\langle out.x \rangle \frown tr$ is a divergence of $out!x \rightarrow Y$, and there is some trace $tr''$ of *COPY* whose inputs match the outputs of $\langle out.x \rangle \frown tr$: $tr'' \Downarrow in = (\langle out.x \rangle \frown tr) \Downarrow out$. Recall that

$$COPY \quad \mathbf{sat} \quad tr \Downarrow out \leqslant_1 tr \Downarrow in$$

so $tr'' \downarrow out.T \geqslant tr'' \downarrow in.T - 1$.

The events in $tr'$ will be the inputs of $tr$ and the outputs of $tr''$. So

$$\begin{aligned}
\#tr' &= tr \downarrow in.T + tr'' \downarrow out.T \\
&\geqslant tr \downarrow in.T + (tr'' \downarrow in.T - 1) \\
&= tr \downarrow in.T + (((\langle out.x \rangle \frown tr) \downarrow out.T - 1) \\
&= tr \downarrow in.T + tr \downarrow out.T \\
&= \#tr \\
&\geqslant n
\end{aligned}$$

and so $\#\langle in.v \rangle \frown tr' \geqslant n+1$. Thus the antecedent to the inference rule is true. If all divergences of $Y$ are of length at least $n$, then those of $F(Y)$ are of length at least $n+1$. Applying a recursive call increases the length of a divergent trace, and so the rule allows the deduction that the process cannot diverge. □

## Process-oriented specification

As well as its use in identifying fixed points, the refinement relation $P_1 \sqsubseteq_{FDI} P_2$ supports a process-oriented approach to specification, similar to that taken in the traces model and stable failures model. A specification describes behaviours that are acceptable in a particular situation, and these behaviours can be described either by use of predicates, or else by means of a CSP process expression itself. A process description *SPEC* will have particular failures, divergences, and infinite traces associated with it, and these are taken to be all the acceptable behaviours. An implementation process *IMP* meets this specification if all of its possible behaviours are allowed by *SPEC*, or in other words, if $SPEC \sqsubseteq_{FDI} IMP$.

The specification is the most nondeterministic process which meets the requirement. This means that *SPEC* should allow all possibilities that are not expressly forbidden.

The specification for divergence-freedom is *CHAOS*: any non-divergent behaviour is permitted, but *CHAOS* does not admit any divergences.

The FDR tool allows processes to be checked against process-oriented specifications with regard to their failures and divergences. The behaviour sets associated with finite state processes are closed, since no such processes can have any infinite nondeterminism, so the infinite traces do not need to be checked separately: if the failures and divergences are all acceptable, then the infinite traces must also be.

EXAMPLE 8.17 (ALTERNATING BIT PROTOCOL) The alternating bit protocol of Example 7.7 was verified with regard to its traces and stable failures. It remains to establish divergence-freedom.

Since the medium is not itself described as a CSP process, further properties need to be described in the FDI model in order to establish divergence-freedom. Firstly, that the medium is itself be divergence-free. Secondly, a fairness assumption is required to ensure that the medium does not simply lose messages for ever. These assumptions cannot be expressed in the stable failures model.

$$\begin{array}{rcl} Med_I(in, out)(u) & = & \#(u \Downarrow in) = \infty \Rightarrow \#(u \Downarrow out) = \infty \\ Med_D(in, out)(tr) & = & false \end{array}$$

The predicate on infinite traces requires that enough messages pass through the medium: if infinitely many have been input, then infinitely many must be output. The requirement that there are no divergences is a consequence of the other requirements: if some divergence is possible, then the other requirements would be violated by the arbitrary behaviours following divergence.

The system can then be shown to be divergence-free. The components themselves are individually divergence-free, and the internalizing of the communications over the channels does not introduce divergence. If there were some infinite sequence $u$ of internal actions on the $c$ and $d$ channels without any inputs or outputs, then $u$ would have to contain an infinite sequence of transmissions along channel $c_1$, since the other channels can only see as many messages as $c_1$. This would have to end with an infinite sequence of the same message $x.b$, since they all arise from the state $S(b, x)$. Hence by $Med_I$ and $Med_F$ there would be an infinite number of occurrences of $x.b$ on channel $c_2$, and infinitely many acknowledgements $b$ sent along $d_1$, and so infinitely many acknowledgements $b$ received along $d_2$. But this is impossible, since receipt of $b$ in state $S(b, x)$ means that $S$ becomes ready for the next input, and does not engage in any further internal activity until this arrives.

This permits the conclusion that *ABP* is divergence-free, and hence that it is a buffer.

□

## 8.6 CASE STUDY: DISTRIBUTED SUM

The final property required of the distributed summing network *DISTSUM* is divergence-freedom.

The effect of abstracting the internal channels must be considered.

$$DISTSUM \quad = \quad NETWORK \setminus \{c_{ij} \mid (i,j) \in E\}$$

If *NETWORK* can be shown to be divergence-free, and a bound on the length of its traces can be found, then it follows that *DISTSUM* is divergence-free.

All the recursive definitions for the family of processes *TOT* are guarded, and none of the definitions uses either *DIV* or the hiding operator, so all the *TOT* processes are divergence-free. This means that all of the $NODE(i)$ processes are divergence-free, since they are composed of divergence-free processes, and hence that *NETWORK* is divergence-free, being a parallel combination of divergence-free processes.

There is also a bound on the number of communications each node can be involved in. It will communicate on each channel at most once, and can terminate at most once.

$$N2_i \qquad NODE(i) \ \mathbf{sat} \ \#tr \leqslant \# \,\mathsf{channels}(A_i) + 1$$

This may be established in the traces model (see Exercise 5.16), and imported into the FDI model using the fact that each $NODE(i)$ is divergence-free.

This places a bound on the length of a trace *tr* of *NETWORK* number of internal events that may occur. The property $N2$ on each node *i* identifies a bound on the length of the trace $tr \restriction A_i^{\checkmark}$, and $tr = tr \restriction (\bigcup_i A_i)^{\checkmark}$, so $\#tr \leqslant \Sigma_i(tr \downarrow A_i^{\checkmark}) \leqslant \Sigma_{i \in N}(2* \mid adj(i) \mid +1)$.

Hence an appropriate bounding function $\beta(tr)$ is simply the constant function $\beta(tr) = \Sigma_{i \in N}(2* \mid adj(i) \mid +1)$: no more than this number of internal events can ever occur. It follows that *DISTSUM* is divergence-free.

## 8.7 MUST TESTING AND FDI EQUIVALENCE

The FDI model is equivalent to another form of testing equivalence which can be defined directly on the operational semantics. In contrast to may testing, which was concerned with the *possibility* of a process passing a test, the form of testing appropriate for the FDI model is one which considers when processes are *guaranteed* to pass tests. This is known as *must testing*.

A process $P$ is tested in the same way as described in Section 4.3: $P$ is placed in parallel with a test $T$, and all of the events in $\Sigma$ are hidden. Since this form of testing is concerned with the guarantees about a process' execution, only the maximal executions of $(P \underset{\Sigma}{\|} T) \setminus \Sigma$ are considered, to allow the process the opportunity to exhibit the desired behaviour.

The maximal executions are those sequences of states and transitions that cannot be extended, either because they reach a final state from which no further events are possible, or else because they consist of an infinite sequence of transitions. For example, the process

$$N \quad = \quad P = a \to N \,\Box\, b \to c \to STOP$$

has $\langle N, \tau, P, a, N, \tau, P, b, (c \to STOP), c, STOP\rangle$ and $\langle N, \tau, P, a, N, \tau, P, a, N, \ldots\rangle$ as maximal executions. On the other hand, $\langle N, a, N, b, (c \to STOP)\rangle$ is not a maximal execution.

The process $P$ passes a test $T$ if *all* of the maximal executions of $(P \underset{\Sigma}{\|} T) \setminus \Sigma$ pass through some state in which a success event $\omega$ was possible. This is written $P$ **must** $T$.

Two processes are considered to be equivalent if they must pass exactly the same tests:

$$P_1 \equiv_{must} P_2 \quad = \quad \forall\, T \bullet P_1 \textbf{ must } T \Leftrightarrow P_2 \textbf{ must } T$$

The requirement that only maximal executions are considered allows the identification of refusal behaviour, since a finite maximal execution must end in a deadlocked state. In such a state, any events not refused by the testing process must be refused by the process under test.

Must testing also allows the distinction of internal from external choice. For example, the two processes

$$\begin{aligned} P_1 &\quad = \quad a \to STOP \,\Box\, b \to STOP \\ P_2 &\quad = \quad a \to STOP \sqcap b \to STOP \end{aligned}$$

are distinguished by the test $T = b \to SUCCESS$. The first process $P_1$ must pass this test, since it is unable to refuse to synchronize on the initial $b$. The empty execution is not maximal for $(P_1 \underset{\Sigma}{\|} T) \setminus \Sigma$; its only maximal execution indeed passes through a success state. The other process $P_2$ might not pass the test, since one of its possibilities is to resolve the choice in favour of $a \to STOP$, resulting in a deadlock and the inability of the test to reach its success state.

The must approach to testing also results in the treatment of divergence as catastrophic, and the equivalence of all processes that might possibly diverge. If $N = (a \to N) \setminus \{a\}$, then the divergent process $N$ is equivalent under must testing to a process $P_3$ which has the possibility of diverging among other possibilities:

$$P_3 \quad = \quad P_3 \,\Box\, b \to STOP$$

If a test $T$ is not initially in a success state, then the infinite execution

$$\langle (P_3 \underset{\Sigma}{\|} T) \setminus \Sigma \ , \ \tau \ , \ (P_3 \underset{\Sigma}{\|} T) \setminus \Sigma \ , \ \tau \ , \ (P_3 \underset{\Sigma}{\|} T) \setminus \Sigma \ , \ \ldots \rangle$$

is a maximal execution which does not take $T$ through a success state. The possibility of divergence in $P_3$ ensures that $P_3$ does not always pass $T$, and so $P_3 \equiv_{must} N$. No guarantees can be provided about the behaviour of a process that might diverge.

This notion of testing equivalence turns out to correspond exactly to equivalence in the FDI model:

$$P_1 \equiv_{must} P_2 \quad \Leftrightarrow \quad (\mathcal{F}\,[\![P_1]\!], \mathcal{D}\,[\![P_1]\!], \mathcal{I}\,[\![P_1]\!]) = (\mathcal{F}\,[\![P_2]\!], \mathcal{D}\,[\![P_2]\!], \mathcal{I}\,[\![P_2]\!])$$

If two processes have different semantics in the FDI model, so there is some behaviour of one that is not a behaviour of the other, then a test can be constructed which distinguishes them; and conversely, if there is a test that distinguishes two processes, then there is some behaviour in the FDI semantics of one that is not in the semantics of the other (see Questions 8.23 and 8.24). The FDI model is fully abstract with respect to must testing.

Must testing also gives rise to a natural notion of refinement:

$$P_1 \sqsubseteq_{must} P_2 \quad = \quad \forall\, T \bullet (P_1 \ \mathbf{must}\ T) \Rightarrow (P_2 \ \mathbf{must}\ T)$$

If the specification process $P_1$ provides some guarantees concerning its behaviour within a particular context, then any implementation of it $P_2$ must meet these guarantees.

Must testing refinement is equivalent to refinement in the FDI model:

$$P_1 \sqsubseteq_{must} P_2 \quad \Leftrightarrow \quad P_1 \sqsubseteq_{FDI} P_2$$

## 8.8 NOTES

The traces model for CSP was first introduced by Hoare in [46] . The failures model [13] refined the traces model with the introduction of failure observations, but it modelled divergence as allowing arbitrary failures. This approach is not altogether satisfactory for a number of technical reasons: some of the expected algebraic laws do not hold, and the semantics of recursive definitions are not always in accordance with the operational semantics. This was resolved by Brookes and Roscoe with the introduction of divergences, resulting in the failures-divergences model [14, 12, 100].

Brookes also provided the first *algebraic semantics* for CSP [12] , in which a set of algebraic laws on process terms are actually taken to define equivalence between processes.

A more recent presentation of algebraic semantics for CSP can be found in [103]. This is also the primary semantic approach taken by the process algebra ACP [5].

Neither the failures model nor the failures-divergences model are able to model infinite nondeterminism, because the refusal sets essentially have to be finite in order to ensure that the model is a complete partial order, required for defining recursive processes. This limitation forces certain restrictions on the language of CSP: alphabet renaming has to be finite-to-one; hiding must be restricted to finite sets; and only finite nondeterministic choices are permitted. These restrictions were all enforced in [47] by the requirement that process alphabets should be finite. The introduction of the FDI model [101] introduced the ability to model arbitrary nondeterminism with the introduction of infinite traces as additional observations. See [103] for an alternative presentation. The technical property $I2$ of that model is defined rigorously and discussed fully there and in [9]. A different fixed point theory, proposed by Roscoe [101] and refined by Barrett [6, 79], is needed for this model as the standard approaches are not applicable (see [101]).

The stable failures model for CSP [103], is a relatively recent development, arising from a collaboration between Jategoankar, Meyer and Roscoe. A similar model was presented by Valmari [118]. Rather than attempt to model divergence within a failures model, the insight behind the stable failures model is that divergence can often usefully be ignored.

The traces model and the stable failures model both form a complete lattice (and hence a complete partial order) under the subset order. In both cases all of the CSP operators are monotonic and continuous, with *STOP* as the bottom element of the traces model, and *DIV* as the bottom element of the stable failures model. This ensures that all recursive definitions have a least fixed point, which means that all recursive processes are well-defined. Furthermore, they each form a complete metric space with the distance function

$$d(P, P') \quad = \quad \inf\{2^{-n} \mid P \upharpoonright n = P' \upharpoonright n\}$$

where in the traces model $P \upharpoonright n = \{tr \in traces(P) \mid \#tr < n\}$ and in the stable failures model $P \upharpoonright n = \{(tr, X) \in \mathcal{SF}\,[\![P]\!] \mid \#tr < n\}$. In each case, the longer it takes to tell $P$ from $P'$, the closer together they are. In this metric space, all guarded CSP functions correspond to contraction mappings, and hence have a *unique* fixed point.

The FDI model also has the partial order structure and the same results hold for it. However, this structure is weaker than those of the other models, and so more effort is required to ensure that all the processes have the correct semantics. [21] provides an introduction to the lattices and partial order structures involved. Introductory material on metric spaces can be found in [115]. Appendix A of [103] also discusses both kinds of structures with particular emphasis on their use in CSP.

Note that the terminology of *CHAOS* follows the more recent treatment given in [103] and differs from [47]. In the latter book, *CHAOS* was the immediately divergent process (what we call *DIV*) and there was no special name for the most nondeterministic divergence-free process.

The distributed sum algorithm is due to [18]; it is a variant of Segall's Propogation of Information with Feedback protocol [112]. The first formal verification of the algorithm was given by Vaandrager [117]. Groote has also provided a formal verification [40].

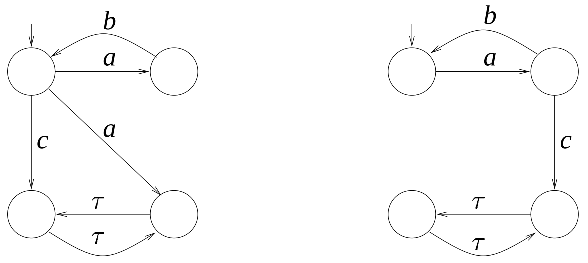


***Fig. 8.3*** Two recursive processes

## Exercises

EXERCISE 8.1 What are the failures, divergences, and infinite traces associated with the finite state machines in Figure 8.3.?

EXERCISE 8.2 What are the FDI semantics of:

1. $N_1 = (a \rightarrow N_1) \;\square\; b \rightarrow STOP$
2. $N_2 = b \rightarrow a \rightarrow DIV$
3. $N_1 \parallel N_2$
4. $N_1 \underset{\{b\}}{\parallel} N_2$
5. $N_1 \;|||\; N_2$
6. $N_1 \setminus \{a\}$
7. $N_3 = (a \rightarrow (N_3;\; b \rightarrow SKIP) \;\square\; c \rightarrow SKIP)$
8. $N_4 = (a \rightarrow N_4) \;\triangle\; b \rightarrow STOP$
9. $N_4 \setminus \{b\}$
10. $N_4 \setminus \{a\}$

EXERCISE 8.3 Does the law $\langle\square\text{-}\sqcap\text{-dist}\rangle$ on Page 180 hold in the FDI model ?

EXERCISE 8.4 Are the operational rules you gave for *DIV* and *CHAOS* consistent with their FDI semantics? If not, give rules which are consistent with both the stable failures and the FDI semantics.

EXERCISE 8.5 1. Give a deterministic process *P* for which $P \setminus A$ is nondeterministic

2. Can a nondeterministic process $P$ have that $P \setminus A$ is deterministic?
3. Give a process P for which $P \parallel P \neq_{FDI} P$
4. Give a nondeterministic process $P$ for which $P \parallel P =_{FDI} P$
5. Give two nondeterministic processes $P_1$ and $P_2$ for which $P_1 \mid\mid\mid P_2$ is deterministic.
6. If $P$ is nondeterministic, can $P \mid\mid\mid P$ be deterministic?
7. Find a deterministic $P$ for which $P \mid\mid\mid P$ is nondeterministic.

EXERCISE 8.6 Give operational rules for *DIV* and *CHAOS* which are consistent with their FDI semantics given above.

EXERCISE 8.7 Capture the following specifications in either the property-oriented or the process-oriented style of specification:

1. A process cannot diverge if it has accepted at least as many coins as it has dispensed chocolates;
2. A process can diverge only if it accepts three consecutive inputs without providing output;
3. A process will not diverge if all of its inputs are less that $2^{32} - 1$;
4. On average, no more than $\frac{1}{3}$ of messages should be lost;
5. Any infinite trace must contain an '$a$' at some point;
6. Every request is eventually serviced;
7. Every execution eventually terminates

EXERCISE 8.8 Find an *FFN* process $P$ such that $P \setminus A$ is not *FFN*.

EXERCISE 8.9 Find an *FFN* process $P$ such that $f(P)$ is not *FFN*.

EXERCISE 8.10 Which of the following variants of the process *EXPBUFF* of Example 8.16 are divergence-free?

$$\begin{array}{lcl}
EXPBUFF2 & = & in?x : T \rightarrow (COPY \gg out!x \rightarrow EXPBUFF2) \\
EXPBUFF3 & = & in?x : T \rightarrow (COPY \gg out!x \rightarrow EXPBUFF3 \gg COPY) \\
EXPBUFF4 & = & COPY \gg (in?x : T \rightarrow out!x \rightarrow EXPBUFF4) \\
EXPBUFF5 & = & in?x : T \rightarrow (out!x \rightarrow EXPBUFF5 \gg EXPBUFF5)
\end{array}$$

Which of them can never refuse input?

EXERCISE 8.11 Which of the following processes are deterministic? Which are *FFN*? Which are closed?

1. $BAG = in?x : T \rightarrow (BAG \mid\mid\mid out!x \rightarrow STOP)$
2. *DISTSUM* of Page 161
3. *ABP* of Example 7.7
4. $INN = in?n \rightarrow A(n)$ where
   $$\begin{array}{lcl} A(0) & = & b \rightarrow STOP \\ A(n+1) & = & a \rightarrow A(n) \end{array}$$
5. $INN \setminus in.\mathbb{N}$
6. $f(INN)$, where $f(in.n) = b$ for any $n$
7. $INN \setminus in.\mathbb{N} \cup \{a\}$
8. $ND = a \rightarrow ND \sqcap b \rightarrow ND \sqcap a \rightarrow b \rightarrow ND$
9. *DIV*
10. *CHAOS*

EXERCISE 8.12 If $\alpha$ is an ordinal number, identified with the set of ordinals less than $\alpha$, then for each $\beta \in \alpha$ define

$$P_\beta \quad = \quad \gamma : \beta \rightarrow RUN_\alpha$$

This process allows any ordinal less than $\beta$ as its initial event, and then allows all sequences of ordinals less than $\alpha$. The recursive definition

$$N \quad = \quad \beta : \alpha \rightarrow ((P_\beta \parallel N) \setminus \alpha)$$

is not guarded, because of the use of the hiding operator within the definition.

1. Can $N$ diverge?
2. How long will the chain of approximations $F^\beta(DIV)$ be before it reaches the fixed point? How many of these approximations can diverge?

Deduce that for any ordinal $\alpha$, if $\alpha \subseteq \Sigma$ then there are recursions that require at least $\alpha$ approximations before the sets of failures and divergences is fixed.

EXERCISE 8.13 The following processes are defined:

$$\begin{aligned}
P_1 &= a \to STOP \sqcap b \to c \to STOP \\
P_2 &= a \to STOP \,\square\, b \to c \to STOP \\
P_3 &= a \to STOP \,\square\, P_3 \\
P_4 &= a \to STOP \,\square\, b \to RUN
\end{aligned}$$

Which of them must pass which of the following tests:

$$\begin{aligned}
T_1 &= a \to SUCCESS \\
T_2 &= b \to c \to SUCCESS \\
T_3 &= a \to SUCCESS \mathbin{|||} b \to SUCCESS \\
T_4 &= (a \to SUCCESS) \setminus \{a\} \\
T_5 &= SUCCESS
\end{aligned}$$

EXERCISE 8.14 Find a test which distinguishes $a \to STOP$ from $STOP \sqcap a \to STOP$.

EXERCISE 8.15 Find a test which distinguishes $a \to b \to STOP$ from $a \to STOP \mathbin{|||} b \to STOP$.

EXERCISE 8.16 Using your operational rule for $DIV$ from Question 8.4, find a test that distinguishes $DIV$ from $STOP$.

EXERCISE 8.17 Using your operational rules for $CHAOS$, find a test that distinguishes $CHAOS$ from $RUN$.

EXERCISE 8.18 Find a test that distinguishes $P_1 = \bigsqcap_{n \in \mathbb{N}} A(n)$ from $P_2 = P_1 \sqcap (N = a \to N)$, where $A(0) = STOP$ and $A(n+1) = a \to A(n)$.

EXERCISE 8.19 Find a test $T$ such that $P$ **must** $T$ if and only if $\langle a, b, c\rangle \notin \mathcal{D}\,[\![P]\!]$.

EXERCISE 8.20 Find a test $T$ such that $P$ **must** $T$ if and only if $(\langle a, b, c\rangle, \{d, e\}) \notin \mathcal{F}\,[\![P]\!]$.

EXERCISE 8.21 Find a pair of processes that are equivalent under may testing but distinguished by must testing.

EXERCISE 8.22 Find a pair of processes that are equivalent under must testing but distinguished by may testing.

EXERCISE 8.23 1. Given $tr \in \mathcal{D}\,[\![P_1]\!]$ and $tr \notin \mathcal{D}\,[\![P_2]\!]$, find a test $T$ such that $\neg(P_1$ **must** $T)$ and $(P_2$ **must** $T)$.

2. Given $\mathcal{D}\,[\![P_1]\!] = \mathcal{D}\,[\![P_2]\!]$ and $(tr, X) \in \mathcal{F}\,[\![P_1]\!]$ and $(tr, X) \notin \mathcal{F}\,[\![P_2]\!]$, find a test $T$ such that $\neg(P_1$ **must** $T)$ and $(P_2$ **must** $T)$.

3. Given $\mathcal{D}\,[\![P_1]\!] = \mathcal{D}\,[\![P_2]\!]$ and $u \in \mathcal{I}\,[\![P_1]\!]$ and $u \notin \mathcal{I}\,[\![P_2]\!]$, find a test $T$ such that $\neg(P_1$ **must** $T)$ and $(P_2$ **must** $T)$.

4. Deduce that if $P_1 \equiv_{must} P_2$ then $P_1 =_{FDI} P_2$.

EXERCISE 8.24 Assume that $\neg(P_1$ **must** $T)$ and $P_2$ **must** $T$. Show that there is some divergence, failure, or infinite trace of $P_1$ which is not also in the semantics of $P_2$.

# *Appendix B: Model-checking with FDR*

This appendix provides a brief overview of the operation and use of the model-checking tool FDR[1], which provides automated analysis and verification of CSP process descriptions. Section B.1 describes the use of the FDR tool; Section B.2 is concerned with the theory underlying the implementation of the tool; and Section B.3 introduces the form of machine-readable CSP required to provide input to the tool.

Some understanding of the workings of FDR is required in order to make best use of the tool. However, for large and complex systems it supports a number of sophisticated techniques which are beyond the scope of this appendix, and the interested reader can find further information in [102, 103, 104, 33].

FDR stands for 'Failures Divergences Refinement checker'. It is a software tool for carrying out automatic analysis of (untimed) CSP processes. Its main operation is in checking whether or not one CSP process refines another. This provides a surprisingly powerful analysis mechanism, since many important questions about processes can be expressed in terms of refinement by employing the process-oriented approach to specification, as discussed in Sections 5.5, 7.4, and 8.5. FDR provides refinement checking for each of the untimed models presented in this book. It also permits analysis for particular common properties, such as deadlock, divergence (livelock), and determinism.

[1]developed and marketed by Formal Systems (Europe) Ltd

FDR has a companion tool, ProBE [34], which stands for ‘Process Behaviour Explorer’. This tool interprets and animates CSP process descriptions, allowing the user to interact with a process and thus explore its behaviour patterns. It allows the user to synchronize on events, to observe the available options at each stage, to backtrack, and to watch the trace being constructed as the process is executed. ProBE does not provide formal analysis; it does provide a better informal understanding of CSP process descriptions. It is simple to use: it can input the same CSP files as FDR, and the user interacts with a process through a window interface by selecting events to perform from the menus of events that are offered. An exploration of Example B.4 is illustrated in Figure B.1. Unexplored events are marked with a ‘$+$’. Any event that has been performed is marked with a ‘$-$’, and followed by a description of the process reached by performing it. For example, subsequent to the performance of $enter.kate$ is the process $PERSON(eleanor) \parallel PERSON(isabella) \parallel PRESENT(kate)$. The menu of events that this combination offers is listed immediately below it:

$enter.eleanor$
$leave.kate$
$enter.isabella$

ProBE then offers the opportunity to explore any of these events.

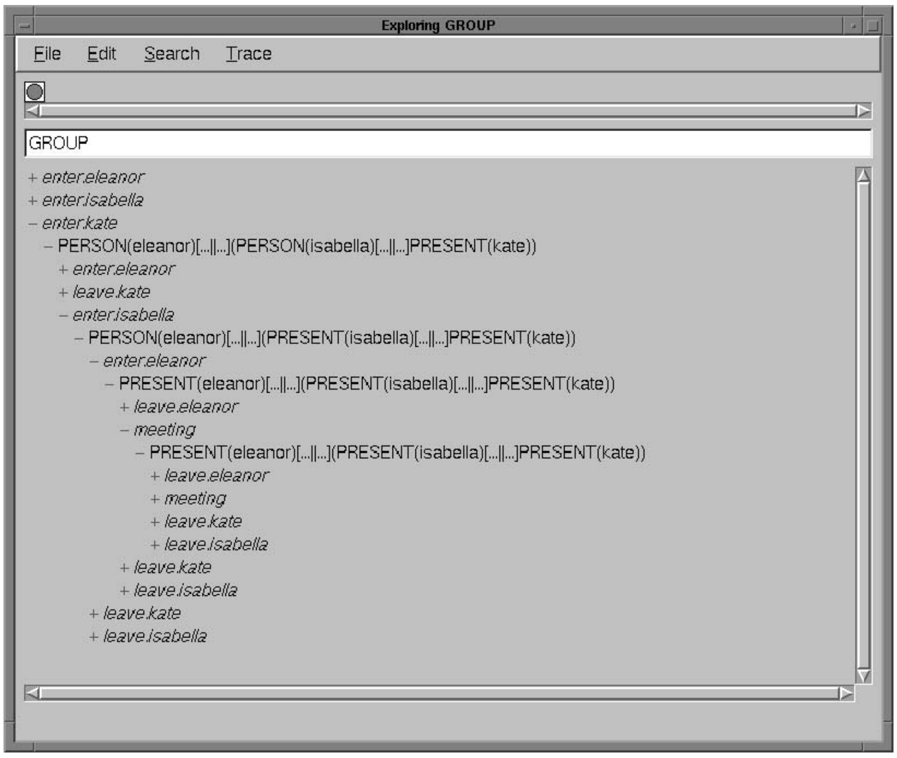
## B.1 INTERACTING WITH FDR

FDR allows automatic refinement checking between (finite state) processes. In order to apply FDR, it is necessary to supply both the specification process and the implementation process. This is accomplished by loading a CSP *script* into FDR. This is simply a text file containing a collection of CSP process definitions. On loading such a script, FDR identifies all of the potential processes within it.

FDR offers the opportunity to do a number of checks on any of the processes that have been loaded. There are specific options to check for deadlock, livelock, and determinism. In addition, refinement checks between processes can be carried out. The interface is illustrated in Figure B.2.

In requesting a refinement check, it is necessary to provide FDR with three pieces of information:

- The specification process. This can be any of the processes that have been loaded in as part of the CSP script, or a process definition typed directly into the ‘Specification’ box. In Figure B.2, the process *SPEC* has been selected.
- The implementation process. This can also be any of the processes that have been loaded, or a process typed into the ‘Implementation’ box. In Figure B.2, *SYSTEM* has been selected.
- The model in which the refinement relation is to be checked. There are three choices: traces, failures, and failures/divergences. These correspond to the three untimed models



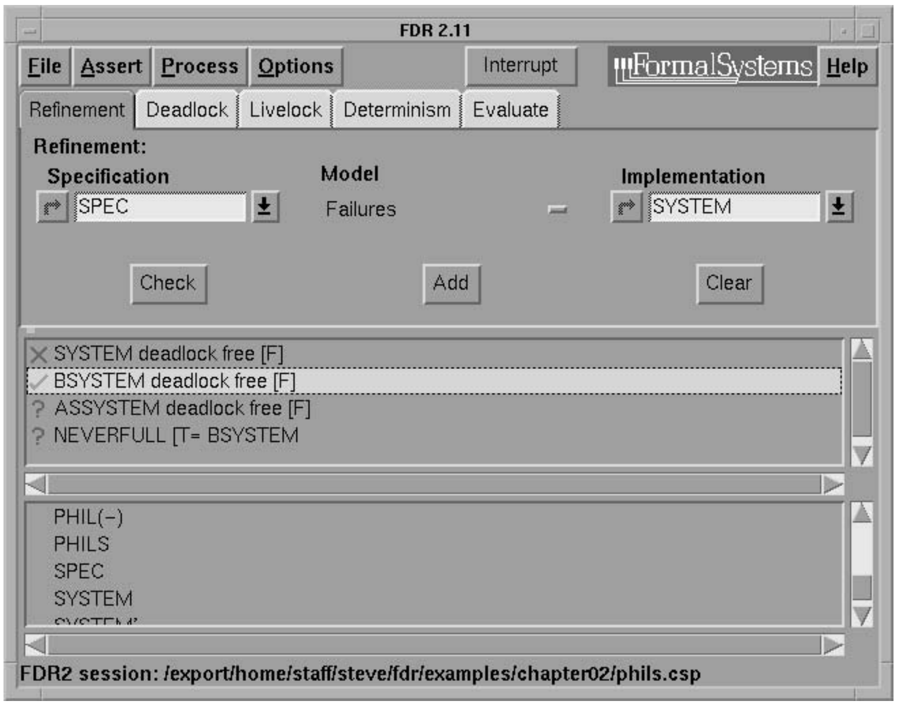
```
GROUP
+ enter.eleanor
+ enter.isabella
– enter.kate
  – PERSON(eleanor)[...||...](PERSON(isabella)[...||...]PRESENT(kate))
    + enter.eleanor
    + leave.kate
    – enter.isabella
      – PERSON(eleanor)[...||...](PRESENT(isabella)[...||...]PRESENT(kate))
        – enter.eleanor
          – PRESENT(eleanor)[...||...](PRESENT(isabella)[...||...]PRESENT(kate))
            + leave.eleanor
            – meeting
              – PRESENT(eleanor)[...||...](PRESENT(isabella)[...||...]PRESENT(kate))
                + leave.eleanor
                + meeting
                + leave.kate
                + leave.isabella
            + leave.kate
            + leave.isabella
        + leave.kate
        + leave.isabella
```

***Fig. B.1*** The ProBE tool

for CSP presented in Part 2 of this book. (The failures/divergences model is the same as the FDI model for finite-state processes). In Figure B.2, the model selected is the failures model.

The CSP script loaded into FDR should therefore in general contain both the specification process and the implementation to be checked against it. There will usually be a number of checks of interest (perhaps a number of properties to check of an implementation, or a number of implementations to check), and so the script will need to contain all of the definitions required for these. The checks to be carried out (refinements or specific options) are either entered into the relevant window in FDR, or they can be included directly in the script as assertions. This enables them to be loaded into FDR alongside the processes, and FDR will list them when the script is loaded. In Figure B.2, a refinement check has been entered directly into the refinement window. Furthermore, four checks have loaded directly from the script: deadlock-freedom assertions on three different processes, and one traces refinement assertion.

If a check is successful, then this is reported by the tool with a $\checkmark$ alongside the verified assertion. If the check fails, reported by a $\times$, then this will be because the tool has identified a particular erroneous behaviour of the implementation process that violates the assertion.



Refinement:
Specification SPEC
Model Failures
Implementation SYSTEM

```
✗ SYSTEM deadlock free [F]
✓ BSYSTEM deadlock free [F]
? ASSYSTEM deadlock free [F]
? NEVERFULL [T= BSYSTEM
```

```
PHIL(-)
PHILS
SPEC
SYSTEM
```

FDR2 session: /export/home/staff/steve/fdr/examples/chapter02/phils.csp

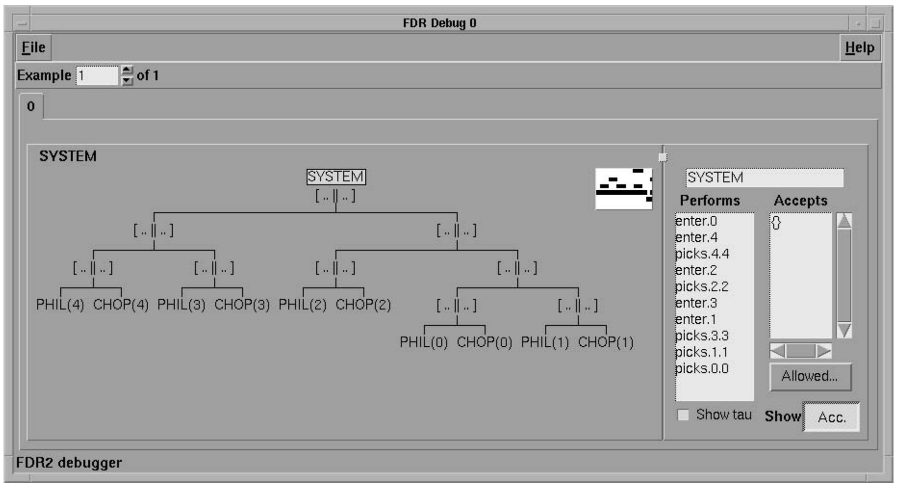
***Fig. B.2*** The FDR 2.11 main screen

In this case, the tool will identify the behaviour and provide it as feedback to the user. For example, with *SYSTEM* as a description of the dining philosophers combination checked for deadlock-freedom, the feedback window provided when it identifies a possible deadlock is given in Figure B.3. The trace leading to it is given in the ‘Performs’ box towards the right hand side.

Deadlock is shown by the fact that the resulting state has an empty acceptance set as shown in the ‘Accepts’ box, which means that all possible events can be refused.

The process tree labelled ‘SYSTEM’ shows the concurrent components of the process *SYSTEM*. It is possible to pick out the contribution of particular subcomponents to the erroneous behaviour. For example, the contribution $\langle enter.2, picks.2.2\rangle$ of $PHIL(2)$ can be extracted by clicking on $PHIL(2)$.

This feedback is useful for debugging purposes. In establishing how a concurrent system is able to reach an incorrect state, design mistakes can be understood and corrected.



```
Performs     Accepts
enter.0      {}
enter.4
picks.4.4
enter.2
picks.2.2
enter.3
enter.1
picks.3.3
picks.1.1
picks.0.0
```

***Fig. B.3*** Feedback from FDR

## B.2 HOW FDR CHECKS REFINEMENT

FDR checks refinement claims of the form *SPEC* $\sqsubseteq$ *IMP*, where *SPEC* is a process-oriented specification. This is achieved by exploring the state space given by the operational semantics of the process *IMP*, and checking for each state that all of the possibilities of event performance (as well as refusals and divergences where appropriate) for the implementation *IMP* are allowed by the specification *SPEC*. This is achieved by matching *IMP*'s transitions in *SPEC*, and examining what *SPEC* will allow in each state that *IMP* can reach. Since *SPEC* is considered as a specification, *SPEC*'s behaviours are taken to be all of the behaviours that are permitted for any refinement *IMP*.

In order to use *SPEC* effectively in tracking the transitions of *IMP*, it is necessary to identify, for any given trace that *IMP* could potentially perform, all of the possible events that *SPEC* can allow next, and also all of the refusals that *SPEC* will permit. FDR achieves this by determinizing the transition graph of *SPEC*, , and adding refusal and divergence information to obtain a 'normalized' graph. Determinization involves removing all $\tau$ events, and coalescing sets of states that can all be reached via the same sequence of visible events. Thus in the determinized state space, each state will correspond to a set of states in the original state space. Any event will appear on at most one transition out of each node, and so for any particular trace of *SPEC* there is a unique node that can be reached in the determinized graph. For example, the determinization of

$$\begin{array}{lcl} VMSPEC & = & coin \rightarrow (\ (tea \rightarrow VMSPEC) \\ & & \qquad\qquad \sqcap\ (coffee \rightarrow VMSPEC)) \\ & & \sqcap\ water \rightarrow VMSPEC \end{array}$$

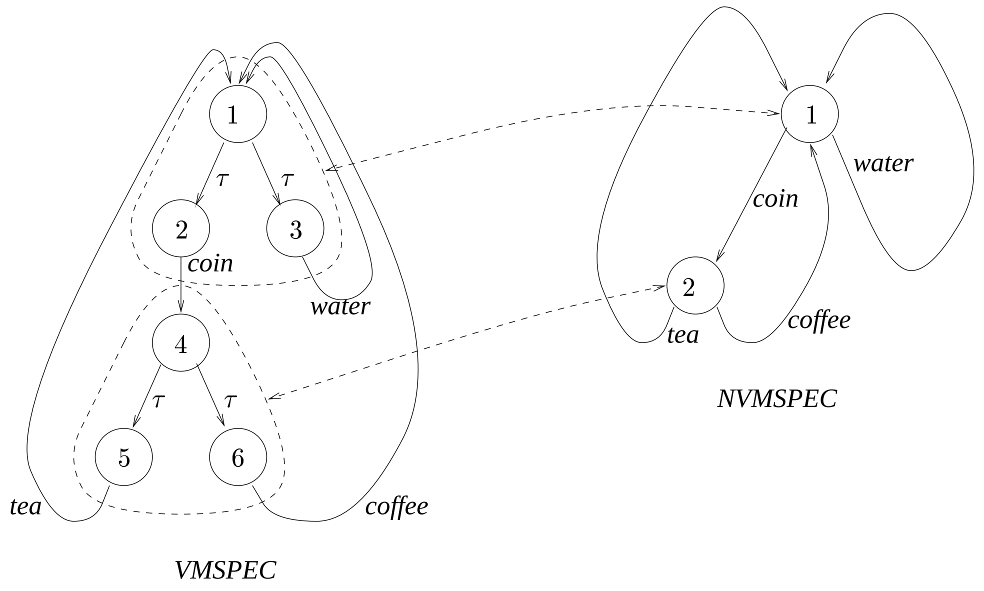


***Fig. B.4*** Determinizing a state graph

to the graph *NVMSPEC* is illustrated in Figure B.4.

Divergent states are also detected during this procedure, and labelled explicitly as such.

Refusal information is not contained in the transitions of the determinized graph alone, since transitions from different states of the original graph are all possible from their coalesced state. For example, states 5 and 6 of the *VMSPEC* graph of Figure B.4 are both part of state 2 of *NVMSPEC*, from which both *tea* and *coffee* are possible. The information that *tea* and *coffee* are also refusable in those states cannot be derived simply from the transitions of *NVMSPEC*.

The refusal possibilities thus need to be recorded explicitly with the nodes of the determinized graph. The complete normalization of *VMSPEC* is given in Figure B.5.

Although the process of normalizing a CSP process is exponential in the number of states in the worst case (since each state in the normalized graph will correspond to a set of states in the original graph), it has been found that in practice the worst case arises very rarely. This is partly because specification processes *SPEC* are often very simple, so that usually the size of the normalized graph is the same size or even smaller than the original graph.

FDR checks a process *IMP* by identifying for each reachable state the corresponding state of (the normalized) *SPEC* that indicates what is permitted. In exploring the state space of *IMP*, it checks for each reached state $IMP_0$ that all of the transitions $IMP_0 \xrightarrow{\mu} IMP'_0$ possible from that state are also possible from the corresponding state $SPEC_0$ of *SPEC*. If $\mu$ is an external event then there must be some (unique) $SPEC'_0$ for which $SPEC_0 \xrightarrow{\mu} SPEC'_0$, and the state $IMP'_0$ will need to be checked against $SPEC'_0$. The case where $\mu = \tau$ is a special

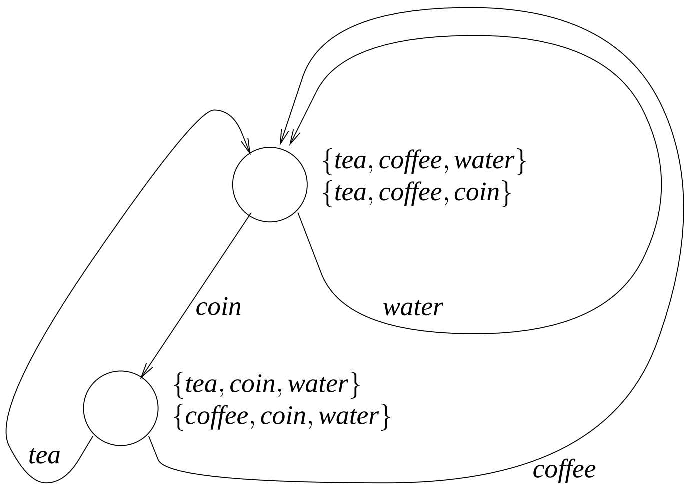


***Fig. B.5*** The state graph *NVMSPEC* labelled with maximal refusals

case—the matching state in *SPEC* is still $SPEC_0$, since the visible trace is not changed by this transition. Thus in this case $IMP_0'$ must be checked against $SPEC_0$.

With regard to refusal information, FDR also performs additional checks on the stable states of *IMP*: that the refusable sets of events are possible refusal sets for *SPEC*.

EXAMPLE B.1 The transition graph of the implementation

$$VMIMP \quad = \quad (coin \to tea \to water \to VMIMP)$$

is given in Figure B.6. It will be checked against the transition graph *NVMSPEC*.

Firstly, state 1 of *VMIMP* is checked against state 1 of *NVMSPEC*. In this state *coin* can be performed, and this is possible for the corresponding state of *NVMSPEC*. Furthermore, only subsets of $\{tea, coffee, water\}$ can be refused by *VMIMP*, and these refusals are permitted by *NVMSPEC*.

In covering the state space of *VMIMP*, the transition *coin* is followed, which requires state 2 of *VMIMP* to be checked against state 2 of *NVMSPEC*. This check is also successful: the refusal sets of *VMIMP* are allowed, and the single transition *tea* is permitted, taking *VMIMP* to state 3, matched by the transition of *NVMSPEC* back to state 1. In this state, the refusals are all permitted by the possible refusals in *NVMSPEC*, and the single transition *water* can be matched, returning both *VMIMP* and *NVMSPEC* to their initial states. Since this pair of states has already been checked, the exploration of the state graph for *VMIMP* is complete and the refinement relation $VMSPEC \sqsubseteq_{failures} VMIMP$ is confirmed. At this point FDR will confirm that the refinement relation holds. □

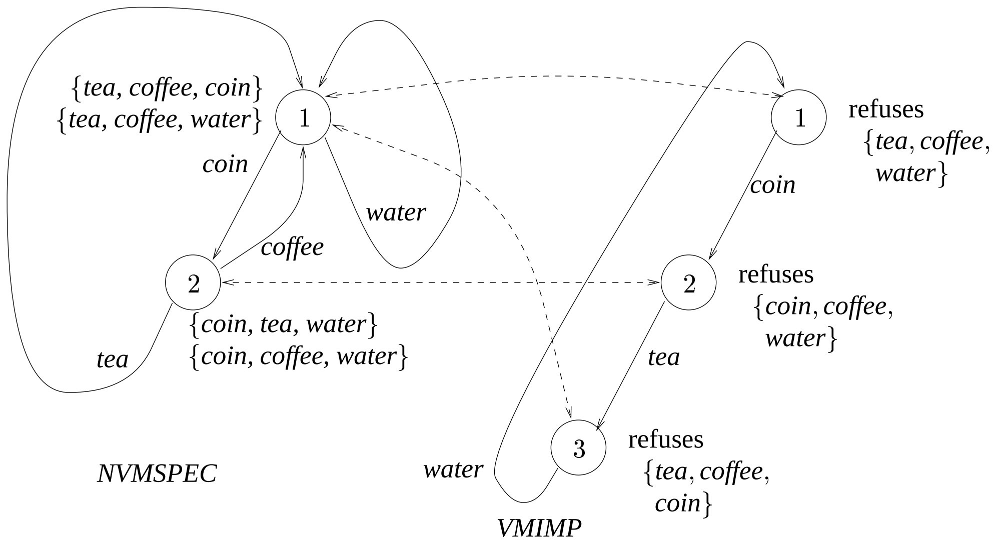


**Fig. B.6** Checking *VMIMP* against *NVMSPEC*

EXAMPLE B.2 An alternative implementation

$$VMIMP2 \quad = \quad (coin \rightarrow tea \rightarrow SKIP \; ||| \; water \rightarrow SKIP); \; VMIMP2$$

fails to refine *NVMSPEC*. The pairs of states that are checked is illustrated in Figure B.7. When checking *VMIMP*2 state 2 against *NVMSPEC* state 2, we find a *water* transition that is not permitted by the specification. This means that the implementation can perform a trace that is not allowed by the specification: the trace $\langle coin, water \rangle$. In this way the model-checking provides a witness trace which demonstrates that *VMIMP*2 is not a trace refinement of *NVMSPEC*. When FDR finds that a refinement fails to hold, it provides the counterexample which it has discovered. The feedback window that it provides for this example appears in Figure B.8. It simply provides the trace $\langle coin, water \rangle$ which fails the specification.

□

The checks that are carried out when the state space is explored depend on the semantic model under consideration.

**Traces model:** checking for refinement in the traces model means checking that all the possible traces of the implementation are allowed by the specification. The refusal and divergence information in the normalized system is ignored. This is appropriate for checking safety properties.

**Failures model:** In this case, stable states that are reached while exploring the implementation are checked with respect to the refusal information recorded in the normalized specification.

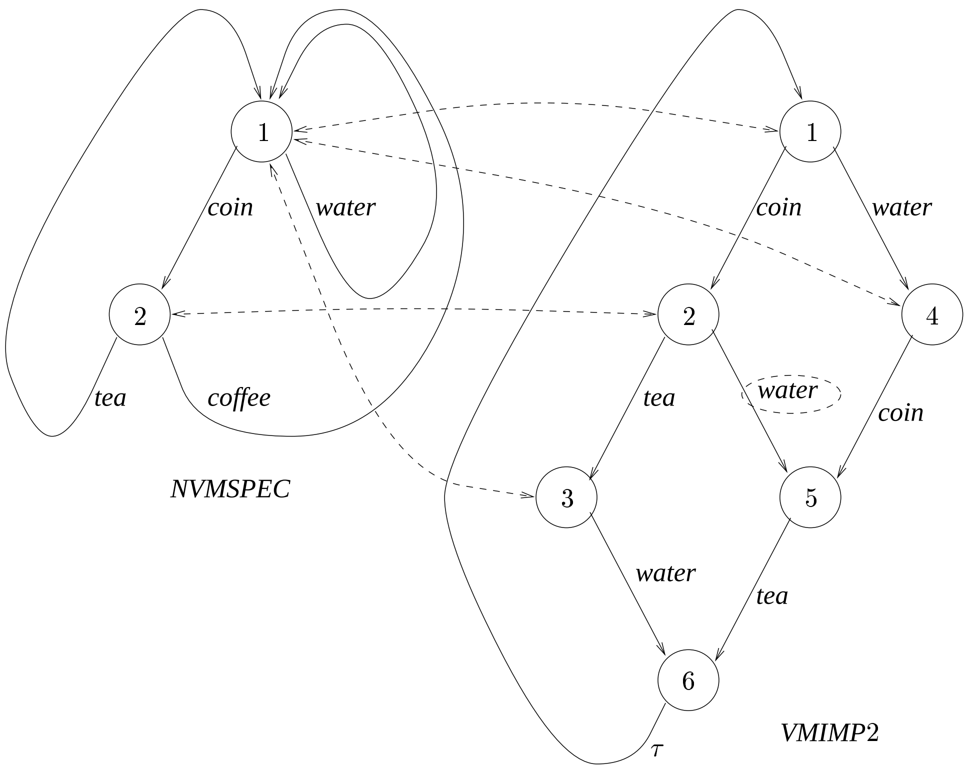


***Fig. B.7*** Checking *VMIMP*2 against *NVMSPEC*

The failures model completely ignores divergence, and so the possibility of divergence in the implementation is simply ignored. This means that if some states are divergent, then they will not be checked with regard to refusal information.

**Failures/Divergences model:** In this case, states in the implementation are also checked for divergence, which will correspond to $\tau$ loops. This can add some considerable overhead to the state space exploration. In practice, implementations will not contain divergences, so it will generally be more efficient to check an implementation specifically for divergence-freedom first (using the specific ‘livelock check’ option). Once this has been established then a check in the failures model will be equivalent to a check in the failures/divergences model and the faster failures refinement check can be carried out.

## Compression

When the state space of the implementation becomes large, it is often advantageous to reduce it before carrying out the state space exploration. FDR provides a number of ways of compressing

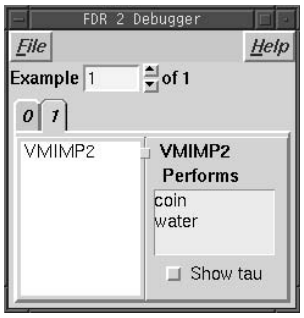


***Fig. B.8*** Feedback on VMIMP2 from FDR

the state space of a process to another (smaller) graph which has the same CSP semantics, and which will take less time to explore. We mention the main mechanisms briefly here.

One example of graph transformation is the normalization that FDR applies to the specification process of a refinement, though in this case more states may result. Many of the compression mechanisms, such as *diamond elimination* and *$\tau$-loop elimination* are concerned with the elimination of $\tau$ events in a semantics-preserving way. Other mechanisms factor a graph by some semantic equivalence. This means that all nodes which have the same semantics are represented by a single node. This reduction can be done efficiently for strong bisimulation (which is stronger than all the CSP semantic equivalences). Its computational expense for other semantic equivalences is rather greater.

There are other mechanisms which achieve some compression but do not in general preserve the semantics, and should therefore be used with extreme care. The *chase* operator will select internal $\tau$ transitions over external ones, possibly removing some of the nondeterminism present and resulting in a refinement of the system it is applied to. It will therefore preserve the semantics of a deterministic system, but may not preserve semantics in general. The *priority* operator (still under development) also alters the operational semantics by removing some of the possible transitions when those of a higher priority are present. This does not even result in a refinement (except in the traces model), since some new refusals may be introduced by the removal of external transitions.

The compression mechanisms are discussed in more detail in [103, 104].

## B.3 MACHINE READABLE CSP

A CSP script defines a number of processes. Channels used by the processes in a CSP script must be typed, where the types are either predefined types or else defined explicitly within the script. For example, the set of dining philosophers and their chopsticks can be defined by

```
nametype Phils = {0..4}
```

| Operator | Syntax | ASCII form |
|---|---|---|
| Stop | $STOP$ | `STOP` |
| Run | $RUN_A$ | `RUN(A)` |
| Chaos | $CHAOS_A$ | `CHAOS(A)` |
| Prefixing | $a \to P$ | `a -> P` |
| Prefix choice | $x : A \to P(x)$ | `[] a :  A @ a -> P(a)` |
| Output | $c!v \to P$ | `c!v -> P` |
| Input | $c?m : T \to P(m)$ | `c?m:T -> P(m)`<br>`channel c :  S`<br>where `T` $\subseteq$ `S` |
| Recursion | $N = P$ | `N = P` |
| Mutual recursion | $N(e) = P(e)$ | `N(e) = P(e)` |
| Choice of process definitions | $\begin{cases} P_1 & \text{if } b \\ P_2 & \text{otherwise} \end{cases}$ | `if b then` $P_1$ `else` $P_2$ |
| External choice | $P_1 \Box P_2$ | $P_1$ `[]` $P_2$ |
| General external choice | $\Box_{i \in I} P_i$ | `[] i :  I @` $P_i$ |
| Internal choice | $P_1 \sqcap P_2$ | $P_1$ `\|~\|` $P_2$ |
| General internal choice | $\bigsqcap_{i \in J} P_i$ | `\|~\| i :  J @` $P_i$ |

***Fig. B.9*** The ASCII representation of CSP: prefix and choice

```
nametype Chops = {0..4}
```

All events and channels that are used by any CSP process in the script must be declared explicitly. Simple events are treated as channels which do not carry messages. For example, the following declarations may appear in a CSP script:

```
channel coin, tea, copy

channel in, out : {0,1}

channel picks : Phils.Chops
```

The first declares three events, so `tea` will be permitted as a process event; the second declares two channels `in` and `out` which both carry bits, so an example communication along

| | | |
|---|---|---|
| Alphabetized parallel | $P_1 \; {}_A\Vert_B \; P_2$ | $P_1$ `[ A \|\| B ]` $P_2$ |
| Alphabetized parallel | $P_1 \Vert P_2$ | |
| General alphabetized parallel | $\Vert_{A_i}^{i\in I} P_i$ | `\|\| i : I @ [`$A_i$`]` $P_i$ |
| General alphabetized parallel | $\Vert^{i\in I} P_i$ | |
| Interleaving | $P_1 \vert\vert\vert P_2$ | $P_1$ `\|\|\|` $P_2$ |
| General interleaving | $\vert\vert\vert_{i\in I} P_i$ | `\|\|\| i : I @` $P_i$ |
| Interface parallel | $P_1 \underset{A}{\Vert} P_2$ | $P_1$ `[\| A \|]` $P_2$ |
| General interface parallel | $\underset{A\, i\in I}{\Vert} P_i$ | `[\| A \|] i : I @` $P_i$ |
| Hiding | $P \setminus A$ | `P \ A` |
| Forward renaming | $f(P)$ | `P[[`$f$`]]` |
| Labelling | $l : P$ | `P[[`$f_l$`]]` |
| Backward renaming | $f^{-1}(P)$ | `P[[`$f^{-1}$`]]` |
| Chaining | $P_1 \gg P_2$ | $P_1$ `[out <-> in]` $P_2$ |
| General chaining | $\gg_{i=1}^{n} P_i$ | `[out <-> in] i : <1..n> @` $P_i$ |

***Fig. B.10*** The ASCII representation of CSP: concurrency and abstraction

these channels would be `out.1`; and the third declares a channel `picks` which carries two values—a value from `Phils` and a value from `Chops`—so an example event on this channel would be `picks.3.4`.

Once the channels are declared, the CSP processes themselves are given. An ASCII form of CSP is used to enable machine readability. Figures B.9, B.10, and B.11 give the main constructs of the ASCII form of CSP.

Mathematical style expressions can be used in the manipulation of process parameters and data variables, and the definition of indexing sets and data types. FDR also handles notation for arithmetic, sets, and sequences, and permits functional style definitions.

EXAMPLE B.3 An bounded $N$ place version of the buffer given in Example 1.22, may be defined using a mutual recursion indexed by sequences of messages. In this example, it will have type $T = \{high, low, middle\}$.

| | | |
|---|---|---|
| Successful termination | *SKIP* | `SKIP` |
| Sequential Composition | $P_1;\ P_2$ | $P_1$ `;` $P_2$ |
| Interrupt | $P_1 \bigtriangleup P_2$ | $P_1$ `/\` $P_2$ |
| Event interrupt | $P_1 \bigtriangleup e \rightarrow P_2$ | `int(`$P_1$`,e,`$P_2$`)` |

***Fig. B.11*** The ASCII representation of CSP: flow of control

$$
\begin{array}{rcll}
B(\langle\rangle) & = & in?x : T \rightarrow B(\langle x\rangle) & \\
B(\langle x\rangle \frown tr) & = & (in?y : T \rightarrow B(\langle x\rangle \frown tr \frown \langle y\rangle) & \\
 & & \Box\ out!x \rightarrow B(s) & \text{if } \#(\langle x\rangle \frown tr)) < N \\
 & & out!x \rightarrow B(s) & \text{otherwise}
\end{array}
$$

This definition involves pattern matching on the sequence parameter to the process $B$, as well as the manipulation of sequence expressions. It may be rendered into a CSP script as shown below.

```
N = 5

datatype Message = high | low | middle

channel in, out : Message

B(<>) = in?x -> B(<x>)

B(<x>^tr) = if # (<x>^tr) < N
            then (in?y -> B(<x>^tr^<y>) [] out!x -> B(tr))
            else (out!x -> B(tr))
```

In order to check a process, values have to be provided for `N` and `T`. The value for `N` is simply declared, whereas `T` is instantiated by the enumerated type `Message` □

EXAMPLE B.4 Example 2.10 is concerned with conditions under which a meeting of a group of people is considered to be quorate. It consists of a number of people who can each enter or leave the meeting independently, but who are required to synchronize on the common event *meeting*. It may be given in machine-readable form as follows:

```
datatype NAMES = kate | eleanor | isabella
```

```
channel enter, leave : NAMES

channel meeting

A(n) = {enter.n, leave.n, meeting}

PERSON(n) = enter.n -> PRESENT(n)

PRESENT(n) = (leave.n -> PERSON(n)) [] (meeting -> PRESENT(n))

GROUP = || name : NAMES @ [A(name)] PERSON(name)
```

Observe that the process `GROUP` could also have been defined by means of an interface parallel construction:

```
GROUP = [| {meeting} |] name : NAMES @ PERSON(name)
```

In order to check that at any stage either a meeting is possible or else someone can enter, the following specification will be used.

```
SPEC = MEETENT ||| CHAOS({|leave|})

MEETENT = (meeting -> MEETENT |~| enter?x -> MEETENT)
```

The specification `SPEC` must always allow one of `meeting` and `enter` to be offered. It places no restrictions on occurrences of `leave`, reflecting the fact that it is not concerned with such events.

The following assertion captures the failures refinement check to be carried out on `GROUP`:

```
assert SPEC [F= GROUP
```

A CSP script to verify `GROUP` would need to contain all of these components. □

## Exercises

EXERCISE B.1 Give a machine-readable version of the cloakroom system of Example 2.3, whose components are as follows:

$$\begin{aligned} ATT &= coat.off \to store \to ATT \\ &\quad \Box\; retrieve \to coat.on \to ATT \\ CUST &= enter \to coat.off \to eat \to coat.on \to CUST \end{aligned}$$

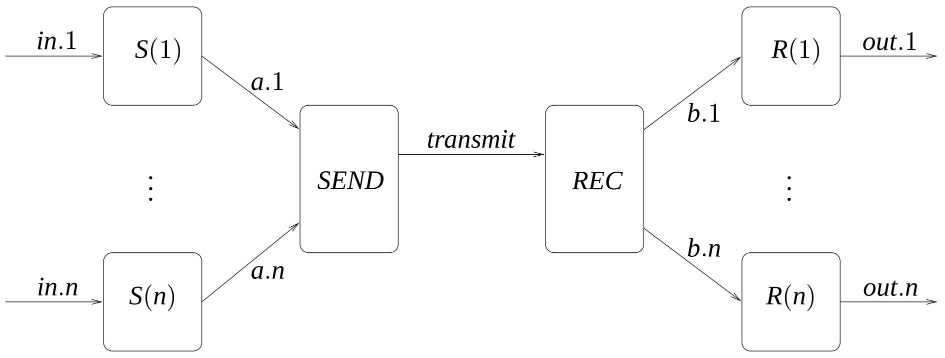


***Fig. B.12*** A multiplexing scheme

Use FDR to check whether $ATT \underset{coat}{\|} CUST$ is deadlock-free.

EXERCISE B.2 Give a machine-readable version of the payment system of the bookshop of Example 2.18. Alter the description of *CHIT* and *RECEIPT* (to $CHIT_4$ and $RECEIPT_4$) so they allow only four chits and receipts to circulate. Use your description to check whether

$$CASHIER \| BOOK \| CHIT_4 \| RECEIPT_4$$

is deadlock-free. Do the results of your analysis apply to the case where there are an unlimited number of chits and receipts?

EXERCISE B.3 Use FDR to check that the railway crossing system $TRAIN \| CROSSING$ of Example 13.4 cannot deadlock. Check further that the gate can only ever move up when the train is not on or entering the crossing. Obtain a trace from FDR that shows that the train can enter the crossing when the gate is down.

EXERCISE B.4 A multiplexing scheme for a family of buffers over a single transmission channel *transmit* is illustrated in Figure B.12. The network is intended to behave as an interleaved collection of buffer processes $BUFFERS(N) = |||_{i \in I} BUFF(i, N)$, where $BUFF(i, N)$ is the general (nondeterministic) specification of a buffer with maximum capacity $N$, on channels $in.i$ and $out.i$.

The components are described as follows:

$$\begin{array}{rcl} S(i) & = & in.i?m \rightarrow a.i!m \rightarrow S(i) \\ R(i) & = & b.i?m \rightarrow out.i!m \rightarrow R(i) \\ SEND & = & a?j?m \rightarrow transmit!j!m \rightarrow SEND \\ RECEIVE & = & transmit?j?m \rightarrow b.j!m \rightarrow RECEIVE \end{array}$$

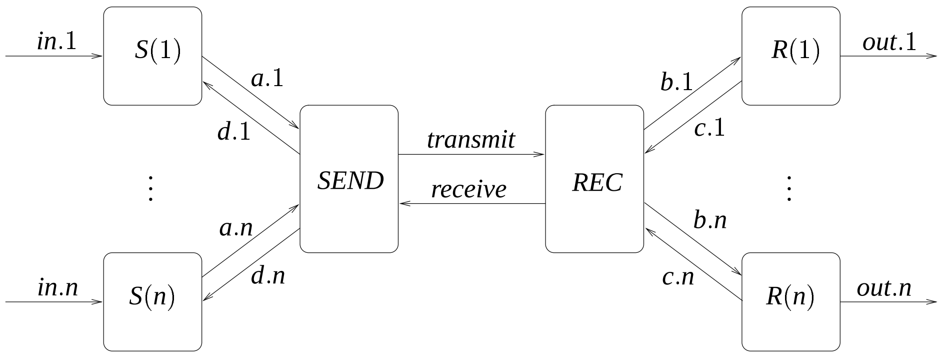


***Fig. B.13*** An alternative multiplexing scheme

Use FDR to check that the combination system (with the internal channels hidden) is a trace refinement of $\mathit{BUFFERS}(N)$ for a suitably large $N$. Is it a failures refinement? Is it deadlock-free? Is it a trace refinement of $\mathit{BUFFERS}(1)$? What is the fundamental problem with this multiplexing scheme?

EXERCISE B.5 In order to overcome some of the shortcomings of with the multiplexing scheme given in Exercise B.4, message acknowledgement is added to the protocol as illustrated in Figure B.13.

1. The components are now described as follows:

$$\begin{array}{rcl} S(i) & = & in.i?m \to a.i!m \to d.i \to S(i) \\ R(i) & = & b.i?m \to out.i!m \to c.i \to R(i) \\ \mathit{SEND} & = & a?j?m \to \mathit{transmit}!j!m \to \mathit{SEND} \\ & & \Box\ \mathit{receive}?j \to d.j \to \mathit{SEND} \\ \mathit{RECEIVE} & = & \mathit{transmit}?j?m \to b.j!m \to \mathit{RECEIVE} \\ & & \Box\ c?j \to \mathit{receive}!j \to \mathit{RECEIVE} \end{array}$$

   Use FDR to check that the combination system (with the internal channels hidden) is a trace refinement of $\mathit{BUFFERS}(1)$. Is it a failures refinement?

2. Place a one-place buffer on the *transmit* channel so the communication between *SEND* and *RECEIVE* becomes asynchronous. Is the resulting system divergence-free? Is it a failures refinement of *BUFFERS*?

# *References*

1. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1), 1993.

2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

3. J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

4. J. C. M. Baeten and J. A. Bergstra. Discrete time process algebra. Technical Report P9208b, University of Amsterdam, 1992.

5. J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

6. G. Barrett. The fixed-point theory of unbounded nondeterminism. *Formal Aspects of Computing*, 1991.

7. J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

8. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

9. S. R. Blamey. The soundness and completeness of axioms for CSP processes. In *Topology and category theory in computer science*. Oxford University Press, 1991.

10. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN systems*, 14(1), 1987.

11. T. Bolognesi and F. Lucidi. *A Timed Full LOTOS with Time/Action Tree Semantics*, chapter 8. Theories and Experiences for Real-time System Development, T. Rus and C. Rattray (eds) - AMAST series in Computing. World Scientific, 1995.

12. S. D. Brookes. *A Model for Communicating Sequential Processes*. D. Phil thesis, Oxford University, 1983.

13. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

14. S. D. Brookes and A. W. Roscoe. An improved failures model for CSP. In *Pittsburgh Seminar on Concurrency*. Springer, LNCS 197, 1985.

15. S. D. Brookes, A. W. Roscoe, and D. J. Walker. An operational semantics for CSP. Technical report, Oxford, 1988.

16. J. Bryans, J. W. Davies, and S. A. Schneider. Towards a denotational semantics for ET-LOTOS. In *CONCUR '95*. Springer, LNCS 962, 1995.

17. A. J. Camellieri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9), 1990.

18. C-T. Chou. Practical use of the notions of events and causality in reasoning about distributed algorithms. Technical Report 940035, UCLA, 1994.

19. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: a semantics based verification tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

20. R. Cleaveland and A. E. Zwarico. A theory of testing for real-time. Technical report, North Carolina S.U. and Johns Hopkins, 1992.

21. B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

22. J. W. Davies. *Specification and Proof in Real-time CSP*. Cambridge University Press, 1993.

23. J. W. Davies, D. M. Jackson, and S. A. Schneider. Broadcast communication for real-time processes. In *Proceedings of the symposium on real-time and fault-tolerant systems*. Springer, LNCS 571, 1992.

24. J. W. Davies and S. A. Schneider. Factorising proofs in timed CSP. In *Proceedings of the Fifth International conference on the Mathematical Foundations of Programming Semantics*. Springer, LNCS 442, 1990.

25. J. W. Davies and S. A. Schneider. Recursion induction for real-time processes. *Formal Aspects of Computing*, 5(6), 1993.

26. J. W. Davies and S. A. Schneider. Real-time CSP. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*, volume 2 of *AMAST Series in Computing*. World Scientific, 1995.

27. J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*. Springer, LNCS 600, 1991.

28. R. de Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1):83–134, 1987.

29. B. Dutertre and S. A. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *Theorem Proving in Higher Order Logics*, 1997.

30. C. Fencott. *Formal Methods for Concurrency*. International Thomson Computer Press, 1996.

31. J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *8th Conference on Computer-Aided Verification*. Springer, LNCS 1102, 1996.

32. Formal Systems (Europe) Ltd. Real-time concurrency. Formal Systems Information Pack, 1991.

33. Formal Systems (Europe) Ltd. Failures-divergences refinement: FDR2 manual, 1997.

34. Formal Systems (Europe) Ltd. Process behaviour explorer manual, 1997.

35. Formal Systems (Europe) Ltd. *Process Behaviour Explorer User Manual*, 1998.

36. H. Garavel. An overview of the Eucalyptus toolbox. In *Applied Formal methods in System Design*, 1996.

37. R. Gerth and A. Boucher. A timed model for extended communicating processes. In *ICALP '87*, pages 157–183. Springer, LNCS 267, 1987.

38. P. Gibbins, A. Kay, and S. A. Schneider. Asynchronous perceptrons in real-time CSP. ESPRIT CONCUR2 project deliverable, 1993.

39. D. Gray. *Introduction to the Formal Design of Real-Time Systems*. Springer, 1999.

40. J. F. Groote and J. Springintveld. Algebraic verification of a distributed summation algorithm. Technical Report CS-R9640, University of Amsterdam, 1996.

41. H. Hansson. *A Calculus for Communicating Systems with Time and Probabilities*. PhD thesis, University of Uppsala, 1991.

42. C. Heitmeyer and D. Mandrioli (editors). *Formal methods for Real-Time Computing*. Wiley, 1996.

43. M. Hennessy. *Algebraic Theory of Processes*. MIT press, 1988.

44. M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.

45. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

46. C. A. R. Hoare. A model for communicating sequential processes. In McKeag and MacNaughton, editors, *On the construction of programs*. Cambridge University Press, 1980.

47. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

48. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

49. J. Hooman. Compositional verification of real-time systems using extended Hoare triples. In de Bakker et al. [27].

50. F. Howles. Distributed arbitration in the futurebus protocol. Master's thesis, Oxford University, 1993.

51. Inmos Ltd. OCCAM2 *reference manual*. Prentice-Hall, 1988.

52. ISO/IEC-JTC1/SC21/WG1/FDT/C, IPS-OSI-LOTOS. A formal description technique based on the temporal ordering of observational behaviour, ISO standard 8807, 1989.

53. ISO/IEC-JTC1/SC21/WG7. Working draft on enhancements to LOTOS, 1997.

54. D. M. Jackson. The specification of aircraft engine control software in timed CSP. Master's thesis, Oxford University, 1989.

55. D. M. Jackson. Specifying timed communicating sequential processes using temporal logic. Technical Report TR–5–90, Programming Research Group, Oxford University, 1990.

56. D. M. Jackson. *Logical Verification of Reactive Software Systems*. D. Phil thesis, Oxford University, 1992.

57. F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, 1986.

58. A. Jeffrey. Discrete timed CSP. Technical Report PMG-78, Chalmers University of Technology, 1990.

59. A. Jeffrey. *Observation Spaces and Timed Processes*. D.Phil thesis, Oxford University, 1991.

60. A. Jeffrey, S. A. Schneider, and F. Vaandrager. A comparison of two axioms for timed transition systems. Technical Report CS-R9366, University of Amsterdam, 1993.

61. G. Jones. *A Timed Model of Communicating Processes*. D. Phil thesis, Oxford University, 1982.

62. G. Jones and M. H. Goldsmith. *Programming in* OCCAM2. Prentice-Hall, 1988.

63. M. Joseph (editor). *Real-Time Systems*. Prentice-Hall, 1995.

64. A. Kay and J. N. Reed. A specification of a telephone exchange in timed CSP. Technical Report TR–19–90, Programming Research Group, Oxford University, 1990.

65. R. L. C. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. PhD thesis, Eindhoven University of Technology, 1989.

66. R. Langerak. A testing theory for LOTOS using deadlock detection. In *Protocol Specification, Testing, and Verification*. Elsevier, 1990.

67. K. G. Larsen, P. Pettersson, and Wang Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer,* 1997.

68. K. G. Larsen and Wang Yi. Time abstracted bisimulation: implicit specifications and decidability. In *Mathematical Foundations of Programming Semantics*. Springer, LNCS 802, 1993.

69. L. Léonard. *An Extended LOTOS for the Design of Time-Sensitive Systems*. PhD thesis, University of Liège, 1997.

70. L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer networks and ISDN systems*, 29:271–292, 1997.

71. N. G. Leveson and J. L. Stolzy. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, 1987.

72. Liang Chen. *Timed Processes: Models, Axioms and Decidability*. PhD thesis, University of Edinburgh, 1992.

73. G. Lowe. *Probabilities and Priorities in Timed CSP*. D. Phil thesis, Oxford University, 1993.

74. N. Lynch and F. Vaandrager. Forward and backward simulations - Part II : Timing. *Information and Computation,* 128(1), 1996.

75. P. Merlin and D. J. Farber. Recovery of communications protocols - implications of a theoretical study. *IEEE Transactions on Communication*, COM-24:1036–1043, 1976.

76. R. Milner. *A calculus of communicating systems*. Springer, LNCS 92, 1980.

77. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

78. R. Milner, J. Parrow, and D. J. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992.

79. M. W. Mislove, A. W. Roscoe, and S. A. Schneider. Fixed points without completeness. *Theoretical Computer Science*, 1995.

80. F. Moller and C. Tofts. A temporal calculus of communicating systems. In *CONCUR '90*. Springer, LNCS 458, 1990.

81. A. Mukkaram. *A Refusal Testing Model for CSP*. D. Phil thesis, Oxford University, 1993.

82. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In de Bakker et al. [27].

83. X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.

84. X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In de Bakker et al. [27].

85. Y. Ortega-Mallén and D. de Frutos. Timed observations: a semantic model for real-time concurrency. In *IFIP-TC2 — Working Conference on Programming Concepts and Methods*. North-Holland, 1990.

86. J. S. Ostroff. *Temporal Logic for Real-Time Systems*. Wiley, 1989.

87. J. Ouaknine. *Connections Between CSP and Timed CSP*. Transfer dissertation, Oxford University, 1997.

88. J. Ouaknine. A framework for model-checking timed CSP. Technical report, Oxford University, 1999.

89. D. Park. On the semantics of fair parallelism. In *Abstract software specifications*. Springer, LNCS 86, 1980.

90. I. Phillips. Refusal testing. *Theoretical Computer Science*, 50, 1987.

91. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

92. K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25, 1995.

93. J. Quemada, D. de Frutos, and A. Azcorra. TIC: a TImed calculus. *Formal Aspects of Computing*, 5:224–252, 1993.

94. J. Quemada and A. Férnandez. Introduction of quantitative relative time into LOTOS. In *Seventh International Symposium on Protocol Specification, Testing, and Verification*, pages 105–121, 1987.

95. G. M. Reed. *A Uniform Mathematical Theory for Real-Time Distributed Computing*. D. Phil thesis, Oxford University, 1988.

96. G. M. Reed. A hierarchy of models for real-time distributed computing. In *Proceedings of the Fifth International Conference on the Mathematical Foundations of Programming Semantics*. Springer, LNCS 442, 1990.

97. G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *13th ICALP*. Springer, LNCS 226, 1986.

98. G. M. Reed and A. W. Roscoe. Metric spaces as models for real-time concurrency. In *Proceedings of the Third International Conference on the Mathematical Foundations of Programming Semantics*. Springer, LNCS 298, 1987.

99. G. M. Reed and A. W. Roscoe. A study of nondeterminism in real-time concurrency. In *Proceedings of the Second UK–Japan CS Workshop*. Springer, LNCS 491, 1991.

100. A. W. Roscoe. *A Mathematical Theory of Communicating Processes*. D. Phil thesis, Oxford University, 1982.

101. A. W. Roscoe. Unbounded nondeterminism in CSP. *Journal of Logic and Computation*, 3(2):131–172, 1993. Also in 'Two papers on CSP', Oxford University Technical Report PRG-67, 1988.

102. A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in honour of C. A. R. Hoare*. Prentice-Hall, 1994.

103. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

104. A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check $10^{20}$ dining philosophers for deadlock. In *1st TACAS*. Springer, LNCS 1019, 1995.

105. B. Scattergood. The description of a laboratory robot in timed CSP. Master's thesis, Oxford University, 1990.

106. F. B. Schneider, B. Bloom, and K. Marzullo. Putting time into proof outlines. In de Bakker et al. [27].

107. S. A. Schneider. *Correctness and Communication in Real-Time Systems*. D. Phil thesis, Oxford University, 1989.

108. S. A. Schneider. Unbounded non-determinism in timed CSP. ESPRIT SPEC project deliverable, 1991.

109. S. A. Schneider. An operational semantics for timed CSP. In *Workshop on Concurrency*, 1992. PMG-63, Chalmers University.

110. S. A. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116, 1995.

111. S A Schneider. Timewise refinement for communicating processes. *Science of computer programming*, 28, 1997. Also in 'Mathematical Foundations of Programming Semantics', Springer, LNCS 802.

112. A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.

113. R. Stamper. The specification of AGV control software in Timed CSP. Master's thesis, Oxford University, 1990.

114. S. Superville. Specifying complex systems with timed CSP: a decomposition and specification of a telephone exchange system which has a central controller. Master's thesis, Oxford University, 1991.

115. W. A. Sutherland. *Introduction to Metric and Topological Spaces*. Oxford University Press, 1975.

116. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In *Formal Methods Europe '97*, 1997.

117. F. Vaandrager. Verification of a distributed summation algorithm. In *CONCUR '95*. Springer, LNCS 962, 1995.

118. A. Valmari. The weakest deadlock-preserving congruence. *Information Processing Letters*, 53:341–346, 1995.

119. B. Victor and F. Moller. The mobility workbench—a tool for the $\pi$-calculus. In *Computer Aided Verification*. Springer, LNCS 818, 1994.

120. A. R. Wallace. A TCSP case study of a flexible manufacturing system. Master's thesis, Oxford University, 1991.

121. W. L. Yeung, S. A. Schneider, and F. Tam. Design and verification of distributed recovery blocks with CSP. Technical Report CSD-TR-98-08, Royal Holloway, University of London, 1998.

122. Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Chalmers University of Technology, 1991.

123. Wang Yi. CCS + time = an interleaving model for real-time systems. In *ICALP '91*. Springer, LNCS 510, 1991.

124. Zhou Chauchen, A. P. Ravn, and C. A. R. Hoare. A calculus of durations. *Information Processing Letters*, 40(5), 1991.

125. J. J. Zic. *CSP + T: a Formalism for Describing Real-Time Systems*. PhD thesis, University of Sydney, 1991.

126. A. Zwarico, I. Lee, and R. Gerber. A complete axiomatization of real-time processes. Technical Report MS-CIS-88-88, University of Pennsylvania, 1987.

## Notation

*Sets and logic*

| | |
|---|---|
| $a \in S$ | $a$ is a member of the set $S$ |
| $\{\}$ | the empty set |
| $\{a_1, \ldots, a_n\}$ | the set of the elements listed |
| $\{a \mid P(a)\}$ | set comprehension: the set of elements which meet predicate $P$ |
| $S \cup T, \quad \bigcup_{i \in I} S_i$ | set union |
| $S \cap T, \quad \bigcap_{i \in I} S_i$ | set intersection |
| $S \setminus T$ | set subtraction |
| $\mathbb{P}\, S$ | power set of $S$ |
| $\mathbb{F}\, S$ | finite power set: the finite subsets of $S$ |
| $\mathbb{N}$ | natural numbers |
| $\mathbb{R}$ | real numbers |
| $\mathbb{R}^+$ | non-negative real numbers |
| $\lceil r \rceil$ | $r$ rounded up to the integer immediately above |
| $\lfloor r \rfloor$ | $r$ rounded down to the integer immediately below |
| $I$ | interval of the real numbers |
| $[b, e)$ | the half-open interval from $b$ to $e$ (including $b$ and excluding $e$) |
| $[b, e]$ | the closed interval from $b$ to $e$ |
| $(b, e)$ | the open interval from $b$ to $e$ |
| $(b, e]$ | the half-closed interval from $b$ to $e$ |
| $\neg P$ | negation: not $P$ |
| $P_1 \wedge P_2$ | conjunction: $P_1$ and $P_2$ |
| $P_1 \vee P_2$ | disjunction: $P_1$ or $P_2$ |
| $P_1 \Rightarrow P_2$ | implication: $P_1$ implies $P_2$ |
| $\forall x \bullet S$ | for all $x$, $S$ holds |
| $\forall x : T \bullet S$ | for all $x$ of type $T$, $S$ holds |
| $\exists x \bullet S$ | there is some $x$ for which $S$ holds |
| $\exists x : T \bullet S$ | there is some $x$ of type $T$ for which $S$ holds |

*Event notation*

| | |
|---|---|
| $\Sigma$ | (Sigma) the universal set of events |
| $\Sigma^{\checkmark}$ | $\Sigma \cup \{\checkmark\}$ |
| $\Sigma^{\checkmark,\tau}$ | $\Sigma \cup \{\checkmark, \tau\}$ |
| $\checkmark$ | (tick) the termination event; not in $\Sigma$ |
| $\tau$ | (tau) the internal event; not in $\Sigma^{\checkmark}$ |
| $a$ | external event from $\Sigma^{\checkmark}$ |
| $\mu$ | external or internal event from $\Sigma^{\checkmark,\tau}$ |
| $c.v$ | communication event with channel $c$ and value $v$ |
| channel$(c.v)$ | the channel $c$ of the compound event $c.v$ |
| value$(c.v)$ | the value $v$ in the compound event $c.v$ |
| $A$ | set of events $A \subseteq \Sigma^{\checkmark}$ |
| $A^{\checkmark}$ | $A \cup \{\checkmark\}$ |

*Sequences and Traces*

| | |
|---|---|
| $seq$ | a sequence of elements |
| $\langle\rangle$ | the empty sequence |
| $\langle a_1, \ldots, a_n\rangle$ | the sequence of elements listed |
| $\langle a \mid a \leftarrow seq, P(a)\rangle$ | sequence comprehension |
| $seq_1 \frown seq_2$ | concatenation: $seq_1$ followed by $seq_2$ |
| $head(seq)$ | first element of the sequence |
| $tail(seq)$ | sequence $seq$ without the first element |
| $foot(seq)$ | last element of $seq$ |
| $init(seq)$ | $seq$ without the last element |
| $\#seq$ | length of the sequence |
| $\sigma(seq)$ | the set of events appearing in the sequence |
| $a$ **in** $seq$ | $a$ appears in the sequence $seq$ |
| $seq_1 \leqslant seq_2$ | sequence $seq_1$ is a prefix of $seq_2$ |
| $seq_1 \preccurlyeq seq_2$ | subsequence (not necessarily contiguous) |
| $seq@i$ | the *i*th element of the sequence (counting from 0) |
| $seq \upharpoonright A$ | the subsequence of elements of $seq$ in $A$ |
| $seq \setminus A$ | the sequence of elements of $seq$ not in $A$ |
| $seq \downarrow A$ | the number of occurrences of elements of $A$ |
| $f(seq)$ | $f$ applying $f$ to each element of $seq$ in turn |
| $\mathsf{channels}(tr)$ | the set of channels used in $tr$ |
| $seq$ $\mathsf{interleaves}$ $seq_1, seq_2$ | $seq$ is an interleaving of the sequences $seq_1$ and $seq_2$ |
| $seq$ $\mathsf{synch}_A$ $seq_1, seq_2$ | $seq$ synchronizes $seq_1$ and $seq_2$ on events in $A^{\checkmark}$ |
| $flatten(sseq)$ | the elements of $sseq$ concatenated together |
| $term(seq)$ | $seq$ contains a $\checkmark$ |
| $TRACE$ | the set of all finite traces |
| $ITRACE$ | the set of infinite traces |
| $tr$ | a finite trace |
| $u$ | an infinite trace |

*Timed Traces*

| | |
|---|---|
| $TT$ | the set of timed traces |
| $s$ | a timed trace |
| $s \upharpoonright A$ | $s$ restricted to $A$: $\langle (t,a) \mid (t,a) \leftarrow s, a \in A \rangle$ |
| $s \downarrow A$ | number of $A$'s in $s$: $\#(s \upharpoonright A)$ |
| $s \setminus A$ | $s$ without the elements of $A$: $s \upharpoonright (\Sigma \setminus A)$ |
| $strip(s)$ | $s$ with the times removed: $\langle a \mid (t,a) \leftarrow s \rangle$ |
| $s + t$ | $s$ delayed by $t$: $\langle (t' + t, a) \mid (t', a) \leftarrow s \rangle$ |
| $s - t$ | $s$ brought earlier by $t$: $\langle (t' - t, a) \mid (t', a) \leftarrow s, t' \geqslant t \rangle$ |
| $begin(s)$ | the time of the first event in $s$ (and $\infty$ for the empty trace) |
| $first(s)$ | the first event to appear in $s$ |
| $end(s)$ | the time of the last event in the finite trace $s$ (and $0$ for the empty trace) |
| $last(s)$ | the last event to appear in the finite trace $s$ |
| $s \uparrow I$ | $s$ during $I$ : $\langle (t,a) \mid (t,a) \leftarrow s, t \in I \rangle$ |
| $s \mathrel{\|\!\upharpoonright} t$ | $s$ strictly before $t$: $s \uparrow [0,t)$ |
| $s \upharpoonright t$ | $s$ before $t$: $s \uparrow [0,t]$ |
| $s \upharpoonleft t$ | $s$ after $t$: $s \uparrow [t,\infty)$ |
| $s \mathrel{\upharpoonleft\!\|} t$ | $s$ strictly after $t$: $s \uparrow (t,\infty)$ |
| $s \upharpoonright A$ | $s$ projected onto $A$: $\langle (t',a) \mid (t',a) \leftarrow s, a \in A \rangle$ |

*Timed refusals*

| | |
|---|---|
| $RSET$ | the set of possible refusal sets: sets of timed events |
| $\aleph$ | (aleph) a timed refusal |
| $\aleph \uparrow I$ | $\aleph$ during $I$ : $\{(t,a) \in \aleph \mid t \in I\}$ |
| $\aleph \uparrow t$ | $\aleph$ at $t$ : $\{(t',a) \in \aleph \mid t' = t\}$ |
| $\aleph \mathrel{\|\!\upharpoonright} t$ | $\aleph$ before $t$ : $\aleph \uparrow [0,t)$ |
| $\aleph \upharpoonleft t$ | $\aleph$ after $t$ : $\aleph \uparrow [t,\infty)$ |
| $\aleph + t$ | $\aleph$ translated through $t$: $\{(t' + t, a) \mid (t',a) \in \aleph)\}$ |
| $\aleph - t$ | $\aleph$ translated backwards through $t$: $\{(t' - t, a) \mid (t',a) \in \aleph, t' \geqslant t\}$ |
| $\aleph \upharpoonright A$ | $\aleph$ restricted to $A$: $\{(t,a) \in \aleph \mid a \in A\}$ |
| $\aleph \setminus A$ | $\aleph$ with $A$ events removed: $\aleph \upharpoonright (\Sigma^{\checkmark} \setminus A)$ |
| $\sigma(\aleph)$ | the set of events appearing in $\aleph$ |
| $begin(\aleph)$ | the time of the first event in $\aleph$ (and $\infty$ for the empty refusal) |
| $end(\aleph)$ | the least upper bound of times mentioned in $\aleph$ |

*CSP processes*

| | |
|---|---|
| $P$ | untimed CSP process |
| $Q$ | timed CSP process |
| $STOP$ | deadlock |
| $a \to P$ | prefix |
| $a : A \to P(a)$ | prefix choice (guarded choice) |
| $a_1 \to P_1 \mid a_2 \to P_2$ | guarded choice |
| $c!v \to P$ | output |
| $c?v \to P(v)$ | input |
| $SKIP$ | successful termination |
| $DIV$ | divergence |
| $CHAOS$ | the most non-deterministic divergence-free process |
| $RUN$ | the process that will deterministically perform any event |
| $N$ | process variable |
| $N = P$ | recursive definition |
| $P_1 \Box P_2, \quad \Box_{i \in I} P_i$ | external choice |
| $P_1 \sqcap P_2, \quad \bigsqcap_{i \in I} P_1$ | internal choice |
| $P_1 \; {}_A\|_B \; P_2, \quad \big\|^{A_i}_{i \in I} P_i$ | alphabetized parallel |
| $P_1 \mid\mid\mid P_2, \quad \big\|\big|_{i \in I} P_i$ | interleaved parallel |
| $P_1 \underset{A}{\|} P_2$ | interface parallel |
| $P \setminus A$ | hiding |
| $f(P)$ | renaming |
| $l : P$ | labelling |
| $f^{-1}(P)$ | backward renaming |
| $P_1 \gg P_2$ | chaining |
| $P_1 ;\ P_2$ | sequential composition |
| $P_1 \triangle P_2$ | interrupt |
| $P_1 \triangle_e P_2$ | interrupt on $e$ |
| $a \overset{d}{\to} Q$ | timed prefix |
| $Q_1 \overset{d}{\triangleright} Q_2$ | timeout |
| $WAIT\ d;\ Q$ | delay |
| $Q_1 \triangle_d Q_2$ | timed interrupt |

*CSP notation*

| | |
|---|---|
| $P_1 = P_2$ | $P_1$ and $P_2$ have the same behaviours |
| $P_1 \sqsubseteq P_2$ | $P_1$ is refined by $P_2$ |
| $P_U \sqsubseteq_T Q$ | $P$ is timewise refined by $Q$ |
| $P$ **sat** $S$ | all the observations of process $P$ meet specification $S$ |
| $P$ ref $X$ | $P$ refuses $X$ |
| $P \uparrow$ | $P$ is divergent |
| $P \downarrow$ | $P$ is stable |
| $P_1 \xrightarrow{\mu} P_2$ $(\downarrow \mu)$ | $P_1$ can perform a $\mu$ transition to $P_2$ |
| $P_1 \overset{tr}{\Longrightarrow} P_2$ | $P_1$ can perform the sequence $tr$ and become $P_2$ |
| $Q_1 \overset{d}{\rightsquigarrow} Q_2$ $(\wr\, d)$ | $Q_1$ can perform a $d$ evolution to $Q_2$ |

# Index

ACP, 72
actions with duration, 266
*adj*$(i)$, 159
admissible specification, 249, 369
*A*-independent, 201
ℵ (aleph), 339
algebraic laws, 89
algebraic semantics, 256
alphabetized parallel, 29–42
  FDI semantics, 232
  inference rule, 146, 198
  laws, 102, 352
  operational rules, 31
  stable failures, 180
  timed failures, 350
  timed inference rule, 377
  timed operational rules, 289
  traces, 97
alphabets, 74, 75
alternating bit protocol, 205–207, 253, 443
  timed, 427
ATP, 329

backward renaming, 63–64
  FDI semantics, 238
  inference rule, 151, 202
  laws, 110
  operational rules, 63
  timed failures, 353
  timed inference rule, 381
  timed operational rules, 303
  traces, 111
Baeten, J. C. M., 329, 442
Barrett, Geoff, 257
Bergstra, J. A., 72, 329, 442
bisimulation, 73
  strong, 21
Bolognesi, T, 329
bookshop, 47
boss, 57
Brookes, Stephen, 72, 256
buffer, 17, 54, 247, 285
  and timewise refinement, 422
  liveness, 194
  process-oriented specification, 208
  specification, 141, 194
  timed law, 425
  timed specification, 426
buses, 279

Cauchy sequence, 358
CCS, 72
chain of processes, 292
chaining, 64–66
chaining timed buffers, 423
chains, 39
channel, 9
*CHAOS*
  FDI semantics, 230
  inference rule, 197
  stable failures, 178
  traces, 178
cheese shop, 216
circular saw, 367, 372
communications protocol, 54
compactness, 226, 423
complete lattice, 257
complete metric space, 358
complete partial order, 257
compositional semantics, ix, 89
compound events, 9
compression, 475
congruence, 134–135
  associated equivalence, 134
consistency with an execution, 341
constancy of offers, 297, 315
contraction mapping, 358
CSP
  ASCII form, 477–479
CSP script, 468

Davies, Jim, 442
deadlock, v, vii, 35–37
deadlock-freedom, 247

timed, 426
debugging, 470
delay, 275
laws, 348, 352, 353
timed failures, 347
timed inference rule, 376
delayed prefix, 276
delayed prefix choice, 276
$\delta$ delay, 441
dequeueing, 58
design, vi
determinism, 224
determinization, 471
Dijkstra, E. W., 77
dining philosophers, 77
discrete timed CSP, 452
discrete timed traces, 459
distance function, 357
distributed sum, 158–166, 210–216, 257
distributivity, 114
laws, 114–115
*DIV*
FDI semantics, 229
inference rule, 196
stable failures, 177
traces, 177
divergence, 59, 173, 219–220, 247
and hiding, 59
in recursive calls, 186
interacting with, 223
divergence-freedom, 173, 246
by construction, 249
divergent trace, 220
duration calculus, 329
dynamic specification, 29

encapsulation, 53
environmental assumptions, 371–373
ESTEREL, 329
ET-LOTOS, 329
event guarded, 119
event prefix, *see* prefix
event renaming, *see* renaming
events, 2
eventual non-retraction, 415
on a set, 416
evolutions, 267, 268
exception handling, 276
executions, 85, 88, 313–315
structure, 326
external choice, 18–22
FDI semantics, 231
inference rule, 145, 197
laws, 93, 179, 180, 231
operational rules, 19
stable failures, 179
timed and nondeterminism, 280
timed failures, 349
timed inference rule, 376
timed operational rules, 278
traces, 93

failures, 221
fairness, 248
FDI model, 222
properties, 222–226
FDI timewise refinement, 434–436
FDR, vi, 467–482
Fencott, C., 328
FFN, 226
finite nondeterminism, 226
and recursion, 244
finite timed failures model, 357
finite variability, 325
Fischer, 388, 443
Fischer's algorithm, 389
fixed point, 116
fixpoint operator, 73
flatten, 407
Ford, Henry, 18
FTF model, 357
furniture removers, 33

garage, 43
global clock, 266
Groote, J. F., 257
guarded function, 244
guardedness, 119

handshake synchronization, 29
Hanssen, H., 328

He Jifeng, viii
Hennessy, Matthew, 329
hiding, 53–59
 across choice, 57–59
 FDI semantics, 235
 inference rule, 149, 200
 laws, 108
 operational rules, 54
 stable failures, 183
 timed failures, 352
 timed inference rule, 380
 timed operational rules, 297
 traces, 108
hierarchy of semantic models, 440
Ho-Stuart, C., 329
Hoare, C. A. R. (Tony), viii, 72, 77, 256

ill-timed processes, 387
implementation, vi
indexed alphabetized parallel, 37–42
 inference rule, 146, 199
indexed external choice, 20
 FDI semantics, 231
 inference rule, 145, 197
 laws, 95
 operational rules, 20
 stable failures, 179
 traces, 95
indexed interface parallel, 49, 295
indexed interleaving, 48
indexed internal choice, 24
 FDI semantics, 232
 inference rule, 145, 198
 operational rules, 24
 stable failures, 180
 timed failures, 350
 timed inference rule, 377
 timed operational rules, 282
 traces, 96
indexed parallel
 timed inference rule, 379
inference rules, 4–5
infinite nondeterminism, 236, 238, 241
 and recursion, 242
infinite trace, 221
infinitely fast processes, 286
input, 10
 inference rule, 144, 196
 operational rules, 10
 traces, 92
instantaneous events, 266
int, 460
interface, 1, 30
interface expansion, 63
interface parallel, 48–50
 FDI semantics, 234
 inference rule, 148, 200
 laws, 105, 106, 182, 235, 352
 operational rules, 49
 relationship with parallel operators, 107
 stable failures, 182
 timed failures, 351
 timed inference rule, 380
 timed operational rules, 295
interleaves, 103
interleaving, 42–48
 FDI semantics, 233
 inference rule, 147, 199
 laws, 104, 182, 234, 352
 operational rules, 43
 stable failures, 181
 timed failures, 351
 timed inference rule, 379
 timed operational rules, 293
 traces, 103
internal choice, 18, 22–25
 distributivity laws, 114
 FDI semantics, 232
 inference rule, 145, 198
 laws, 96
 operational rules, 23
 stable failures, 179
 timed failures, 349
 timed inference rule, 377
 timed operational rules, 282
 timed resolution, 281
 traces, 95
interrupt, 70–72
 and nondeterminism, 70

FDI semantics, 239
inference rule, 153, 204
laws, 114
operational rules, 70
timed failures, 354
timed inference rule, 382
timed operational rules, 305
traces, 114
interrupt event, 204
*ITRACE*, 221

Jackson, Dave, 442
Jategoankar, Lalita, 257
Jeffrey, Alan, 329

Klop, J. W., 72
Kruskal's algorithm, 299

labelled transition, 3
labelling, 61
traces, 111
Langerak, Rom, 464
Larsen, Kim, 442
lattice, 257
law of the excluded middle, 5
laws
alphabetized parallel, 102, 352
backward renaming, 110
delay, 348, 352, 353
distributivity, 114
external choice, 93, 179, 180, 231
hiding, 108
indexed external choice, 95
interface parallel, 105, 106, 182, 235, 352
interleaving, 104, 182, 234, 352
internal choice, 96
interrupt, 114
prefix choice, 91
recursion, 118, 189, 244, 246
refinement, 167
renaming, 110
sequential composition, 113
timed interrupt, 355
timeout, 348
Leduc, Guy, 329
Léonard, Luc, 329
Leveson, N., 443
Liang Chen, 328
lift, 2
livelock, v
liveness, 140, 171, 370
LOTOS, 72
Lowe, Gavin, 442

mail forwarder, 46
mail system, 22
marathon, 49
maximal parallelism, 266
maximal progress, 266
and external events, 266
and hiding, 296
and internal events, 266
in timeout, 273
may testing, 131–133
refinement, 362
timed, 362
menu choice, *see* prefix choice
metric space, 257
Meyer, Albert, 257
Milner, Robin, 72
model-checking, 209
modus ponens, 5
Moller, Faron, 328
$\mu$, 73
Mukkaram, Abida, 464
must testing, 254–256
refinement, 362
timed, 362
mutual exclusion, 388, 406
mutual recursion, 13–18, 124–127
operational rules, 14, 189
timed failures, 360
timed operational rules, 286
traces, 125
unique fixed points, 128–131

negative premisses, 297
Newton's method, 66
Newtonian time, 266
Nicollin, X., 328

non-divergent, 173
non-pre-emptive, 442
non-retraction, 412–418
  and timewise refinement, 414, 416
  eventual, 415
    on a set, 416
nondeterminism, v, vii
nondeterministic choice, *see* internal choice
nondeterministic delay, 283, 284
nondeterministic delayed prefix, 283
normalization of state space, 471

observation, ix
OCCAM, 72
office junior, 71
one time stack, 69
operational semantics, 3, 5, 73
Ouaknine, Joel, 463
output, 9
  inference rule, 144, 196
  operational rules, 10
  traces, 91

parallel, *see* alphabetized parallel,
    *see* interface parallel,
    *see* interleaving
parallel composition
  timewise refinement, 411
PARTY, 329
party, 308
Petri nets, 329
Phillips, I., 464
Pi-calculus, 72
pipeline, 33
pipes, 64
Plotkin, Gordon, 73
point nondeterminism, 338
polling, 285
polling for input, 59
postman, 276
Prasad, K. V. S., 442
prefix, 6
  FDI semantics, 228
  inference rule, 143, 195
  operational rules, 6
  stable failures, 177
  timed failures, 346
  timed inference rule, 374
  timed operational rules, 269
  traces, 90
prefix choice, 7
  FDI semantics, 228
  inference rule, 144, 196
  laws, 91
  operational rules, 9
  stable failures, 177
  timed failures, 346
  timed inference rule, 375
  timed operational rules, 272
  traces, 90
Prim's algorithm, 299
process, 1
process algebra, 72
process creation, 46
process-oriented specification, 166–168, 207–
    210, 252–254
promptness, 418–422
  on a set, 420–422
  preserving, 421
property-oriented specification, 139, 246–
    252

Quemada, J., 329

railway crossing, 429, 443
railway network, 79
random numbers, 24
real-time, 266
recursion, 11–13, 115–118
  FDI semantics, 241, 242
  inference rule, 204
  laws, 118, 189, 244, 246
  operational rules, 12, 187
  stable failures, 188
  timed failures, 359
  timed operational rules, 284
  timewise refinement, 402, 411, 436
  traces, 116–117
  unique fixed point
    timed law, 359, 361

unique fixed point law, 120
unique fixed points, 118–123
unwinding, 118
well-defined, 257
recursion induction, 153–155, 204–205, 250–252
divergence-freedom, 250
mutual recursion, 156–158
process-oriented, 209
timed, 383
Reed, G. M. (Mike), 440
refinement, 166, 207, 398
laws, 167
may testing, 439
must testing, 439
refinement checking, 468, 474
refinement ordering, 240
refusal, 172
refusal testing, 349–350, 457
model, 457
refusal token, 339
Regan, Tim, 329
renaming, 60–63
and recursive calls, 66
FDI semantics, 237
inference rule, 150, 201, 202
laws, 110
operational rules, 60
relational, 73
stable failures, 184
timed failures, 353
timed inference rule, 381
timed operational rules, 302
traces, 109
responsibility and control, 267
Roscoe, A. W. (Bill), 72, 73, 79, 256, 440
routing, 40
*RUN*
FDI semantics, 230
inference rule, 145, 197
stable failures, 178
traces, 92

safety, 140, 171
**sat**, 139, 193, 367
satisfiability, 140
scheduler, 248
secretary, 57
Segall, A., 257
segments of track, 79
semantic models, relationships between, 227, 247
sequence notation, 86–88
sequential composition, 67–69
FDI semantics, 238
inference rule, 153, 203
laws, 113
operational rules, 68
stable failures, 185
timed failures, 354
timed inference rule, 382
timed operational rules, 303
traces, 112
Shrödinger's cat, 281
Sifakis, J., 328
*SKIP*, 11
FDI semantics, 229
inference rule, 144, 196
operational rules, 11
stable failures, 177
timed failures, 347
timed inference rule, 375
timed operational rules, 273
traces, 92
sorting, 39
soundness, 5
specification, vi, 86, 139
conjoining, 142
independent of a set, 149
liveness, 193
safety, 139
timed, 367
translating, 403
vacuous, 141
weakening, 141
specification macros, 369–373
spin divergence, 316
spin execution, 316
stability value, 440
stable, 172, 268

stable failure, 174
stable failures model, 174–175
stable refusals, 172–174
stack, 217
static specification, 29
*STOP*, 6, 9
  FDI semantics, 228
  inference rule, 142, 195
  operational rules, 6
  stable failures, 176
  timed failures, 345
  timed inference rule, 373
  timed operational rules, 268
  traces, 90
stop and wait protocol, 54, 58
stopwatch, 287
successful termination, *see SKIP*
sum, 149
$swap_{c,d}$, 65
$\mathsf{synch}_A$, 105
synchronous CCS, 328

*t*-active, 324
  and *t*-guarded, 324
*t*-guarded, 321
temporal logic, 442
  real-time, 329
*term*, 149
termination, 73, 316
  in parallel combinations, 30
test, 132
testing, 73
  may testing, 131–133
  timed failures, 361
threads, 46
time additivity, 312
time closure, 313
time determinism, 311
time interpolation, 312
time-abstracting bisimulation, 442
time-guarded, 286, 288, 320
timed ACP, 329, 442
timed CCS, 328
timed computational model, 266
timed CSP, 265
timed event prefix, 270
  timed operational rules, 270
timed failures, 334, 342–343
timed failures model, 343–345, 356
  properties, 344
timed failures stabilities model, 440
timed interrupt
  laws, 355
  timed failures, 355
  timed inference rule, 383
  timed operational rules, 308
timed LOTOS, 329
timed prefix
  timed failures, 346
  timed inference rule, 374
timed process algebra, 327
timed refusal, 334
  notation, 340–341
timed refusals, 338
timed synchronization, 289
timed trace, 334
  notation, 334–335
  of an execution, 336
timeout, 273
  laws, 348
  timed failures, 347
  timed inference rule, 376
  timed operational rules, 274
  untimed, 58
  variables in time expression, 277
timestop, 316, 317
  absence, 325
timewise refinement
  and non-retraction, 414, 416
  finite interfaces, 422–423
  preservation by operators, 399–401, 410–411
  stable failures, 408
  trace, 398
*tock* events, 445
tock-CSP, 452
tocktimed, 459
Tofts, C., 328
trace equivalence, 89
trace specification

timed translation, 403
traces, 85
traces, 88
traces model, 89
properties, 90
transitions, 3–4
in timed context, 267

urgency, 296, 315
in timed failures, 342

Vaandrager, Frits., 257
Valmari, A., 257
verification, compositional, 141

Wang Yi, 328, 442
watchdog timer, 277
well-behaved, 315
well-timed, 325, 448

zeno behaviour, 319–320
zeno divergence, 317
zeno execution, 317

# Index of Processes

2*COPY*, 65

*AB*, 296
*ACCUMULATOR*, 16
*ALT*, 208
*ATT*, 32, 137
*AUTH*, 185

*B*, 479
*BAG*, 209
*BOOK*, 47
*BOUNCE*, 128
*BOX*, 36
*BROWSING*, 35
*BUFF*, 387
*BUFFER*, 17, 158
*BUS*_1, 20, 91, 94
*BUS*_111, 19
*BUS*_2, 20, 94
*BUS*_37, 19
*BUS_STOP*, 304

*C*1 (constraint process), 290
*C*2 (constraint process), 291
*C*3 (constraint process), 291
*C*4 (constraint process), 291
*CALC*, 33
*CARL*, 289, 378
*CASHIER*, 47
*CH*, 236
*CHAIN*, 39, 61
*CHANGE*, 32, 121
*CHAOS*, 178
*CHIT*, 47
$CHOP_j$, 78
*CHOPSTICKS*, 78
$C_{i,j}$, 39
*CLOCK*, 309
*COFFEE*, 457
*COLLEGE*, 78
*CON*, 79
*CONTROLLER*, 430
$CONTROLLER_{tock}$, 448

*COOKED*, 169
*COPIER*, 306
*COPIER*2, 307
*COPY*, 13, 61, 123
*COUNT*, 156
$COUNT(i)$, 16
$COUNT(n)$, 130
*COUNTER*, 16
$COUNTUP(n)$, 130
*CROSS*, 13, 14, 194
*CROSSING*, 430
$CROSSING_{tock}$, 449
*CUST*, 32, 137

*DAVE*, 37, 181
*DELIVERY*, 277
*DISPENSE*, 381
*DISTSUM*, 161, 216, 254
*DIV*, 177
*DOOR*, 170

*EASEL*, 36
*ELEANOR*, 306
*EMERGENCY*, 306
*EMPTY*, 79
*ENT*, 14

*FASTLIGHT*, 318
*FAX*, 45
*FAXES*, 45
*FIRST*, 79
*FISCHER*, 390
*FOOT*, 66
*FORECOURT*, 43

*GATE*, 430
$MOVE_{tock}$, 449
$GATE_{tock}$, 449
*GROUP*, 38

*HARDBUFF*, 285
*HEAD*, 66
$HEATER(i)$, 15

*HELEN*, 289, 378
*HIGH*, 124
*HOUR*, 66

*IN*, 388
*INPUTS*, 409
*ISABELLA*, 36, 60

*JANET*, 62
*JOURNEY*, 306
*JUNIOR*, 71
*JUNIOR*2, 72

*K*1 (constraint process), 292
*K*2 (constraint process), 292
*KATE*, 36, 61, 306
*KATE_BED*, 291
*KRUSKAL*, 299

*LATECARL*, 290
*LEFT*, 128
*LEVEL*, 322
*LIGHT*, 11, 12, 14, 117, 156, 205, 251
$LINE_n$, 45
*LOW*, 124

*MACHINE*, 32, 121
$MACHINE'$, 122
*MAIL*, 22
*MAIL_SERVICE*, 55
*MAILER*, 42
*MARATHON*, 50
*MASTER*, 54, 137
$MED(c)$, 206
*MID*, 124
*MORNING*, 443

$NBUFF_T$, 208
*NETWORK*, 161, 213, 254
*NEWTON*, 66
$N_l$, 42
*NODE*, 46, 154, 211
$NODE(i)$, 162, 170, 254
$NODE(x)$, 67
*NONREADER*, 76
*NVMSPEC*, 472

*OFFER*, 58, 277, 301
*OFFERS*, 418
*OFFICE*, 62, 201
*ON*, 14, 156
$ONE\_SHOT_d$, 282
*ONE_SHOT*2, 284
$ONE\_SHOT_{[3,5]}$, 283
*OUT*, 388
$OUT(x)$, 46
$OUT_x$, 137

*PAINT*, 62
*PAINTING*, 36
*PARITY*, 59
$PERSON_n$, 38
*PETE*, 37, 181
$PETE'$, 37
$PHIL_i$, 78
*PHILS*, 78
*POLL*, 59, 301
*POLL*1, 236
*POLLING*, 285, 409
*PRIM*, 299
*PRINT*, 300, 380, 461
*PRINT*1, 183
*PRINTER*, 461
*PRINTER*0, 6, 269
*PRINTER*1, 7
*PRINTER*2, 8
*PRINTER*3, 10
*PRINTQ*, 57
*PUMP*1, 43
*PUMP*2, 43
*PURCHASE*, 69

*QS*, 390

*RACE*, 32, 49
*RAW*, 169
*REACT*, 272
*REACT*2, 283
*READER*, 76
*RECEIPT*, 47
*RECORD*, 137
*RELAY*, 21
*RELAY*2, 21

*RIGHT*, 128
*ROUTER*, 23
*RUN*, 92
*RUNNER*, 49

*SAFETY*2, 451
*SALE*, 64, 202
*SAWP*, 123
*SAWP*2, 58
*SEC*, 57
*SECOND*, 79
*SECURE_SHOP*, 34
*SECURITY*, 34
*SERVICE*, 19, 45
*SET*, 17, 67
*SHOP*, 35
*SHOPPING*, 34
*SIMON*, 405
*SORTER*, 40
*SPAWN*, 131
*SPEEDO*, 318
*SPEEDO′*, 319
*SPENDING*, 69
*SPOOL*, 300, 380, 461
*SPY*, 54, 137
*SQRT*, 66
*STACK*, 18, 76, 170
$STACK_1$, 217
$STACK_2$, 217
*START*, 128
*STOPPEDWATCH*, 288
*STOPWATCH*, 288
*STORE*, 69
*SYLVIE*, 62
*SYSTEM*, 430
$SYSTEM_{tock}$, 450

*TASKS*, 71
*TCHAIN*, 293
*TCONTROLLER*, 432
*TCOPY*, 286
*TEA*, 457
*TEAM*, 37
*TFAX*, 294
*TFAXES*, 294
*TGATE*, 432
*THROUGH*, 388
*TICKET*, 32, 121
*TILL*, 35
*TIMED_BUS_*1, 280
*TIMED_BUS_*111, 279
*TIMED_BUS_*2, 280
*TIMED_BUS_*37, 279
*TIMESTOP*, 318
*TINPUTS*, 409
*TINPUTS*2, 416
*TINPUTS*3, 417
*TMED*, 428
*TMORNING*, 443
*TOFFER*, 301
*TOT*, 161, 254
$TOT(i)$, 16
*TOY*, 322
*TPERSON*, 296
*TPOLL*, 301
*TPRESENT*, 296
*TPRINTER*1, 274
$TQ_i$, 292
*TRAIN*, 430
$TRAIN_{tock}$, 450
*TRANS*, 287, 417
*TRANSMIT*, 286
*TSWITCH*, 384
*TTRAIN*, 432
*TTRANS*, 417

*UFISCHER*, 406
*UOFFERS*, 418
*UWITHDRAWS*, 418

*VAR*, 71
$VAR_x$, 137
$V_x$, 137
*VM*1, 191
*VM*2, 191
*VM*3, 191
*VMIMP*, 473
*VMIMP*2, 474
*VMSPEC*, 472

*WAITER*, 276

*WAITER*1, 413
*WAITER*2, 414
*WAITER*3, 414
*WATCH*, 34, 364
*WATCHDOG*, 277
*WITHDRAWS*, 418
*WOBBLE*, 128

$Z(n)$, 319