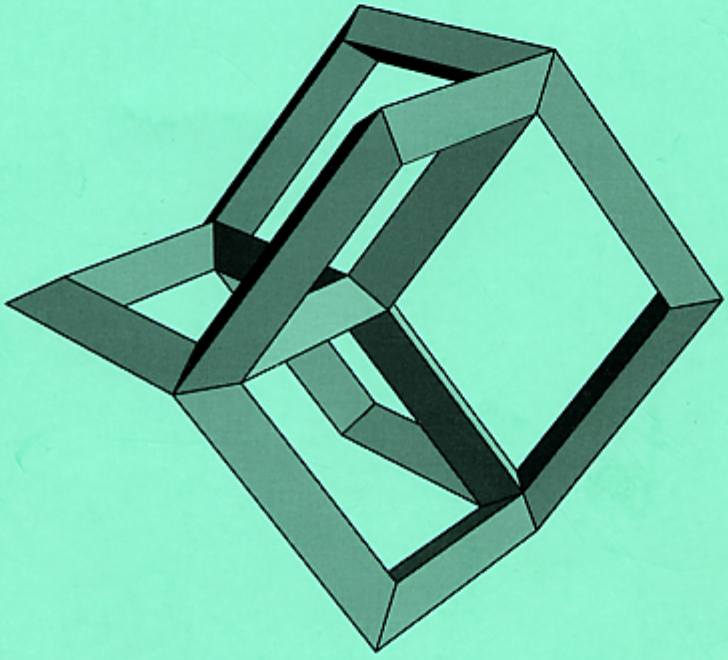
A Theory of

Delay-Insensitive Systems



Tom Verhoeff

A Theory of Delay-Insensitive Systems

Tom Verhoeff

Eindhoven University of Technology
Department of Mathematics and Computing Science

Cover: Koos Verhoeff's impression of the state graph in Figure 3.5, rendered in PostScript with Mathematica by the author. The beams have a triangular cross section. The sculpture enjoys the 24 symmetries of the group S_4, which is also the symmetry group of the tetrahedron. The Mathematica program on the back can be used to generate other views.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Verhoeff, Tom

A theory of delay-insensitive systems / Tom Verhoeff. -
Eindhoven : Eindhoven University of Technology,
Department of Mathematics and Computing Science. - Ill.
Proefschrift Eindhoven. - Met lit. opg., reg. - Met
samenvatting in het Nederlands.
ISBN 90-386-0353-3
Trefw.: communicerende processen ; wiskundige modellen.

A Theory of Delay-Insensitive Systems

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR
AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN,
OP GEZAG VAN DE RECTOR MAGNIFICUS,
PROF.DR. J.H. VAN LINT,
VOOR EEN COMMISSIE AANGEWEZEN DOOR
HET COLLEGE VAN DEKANEN
IN HET OPENBAAR TE VERDEDIGEN OP
VRIJDAG 20 MEI 1994 OM 16.00 UUR

DOOR

TOM VERHOEFF

GEBOREN TE RIJSWIJK, Z.H.

Dit proefschrift is goedgekeurd door

de promotor

prof.dr. M. Rem

en de copromotor

dr.ir. J.T. Udding

Acknowledgment: The research for this dissertation was partially supported by EXACT (ESPRIT Project 6143) and ACiD (ESPRIT Working Group 7225).

To my parents: Bertha & Koos

'Eadum sunt, quorum unum potest substitui alteri salve veritate.'
[Things are equal that can be substituted for
one another without changing correctness.]
Gottfried Wilhelm Leibniz, 1646–1716

Contents

Summary

This research finds its motivation in the design of digital integrated circuits ("chips") and systems comprised of multiple chips. Chips have penetrated all aspects of our life. The timing problem plays a fundamental role in the design of such circuits. Several methods for solving the timing problem are available. In delay-insensitive circuits the timing problem is solved by seeing to it that a circuit's correctness does not depend on assumptions about delays in the connecting wires between the elementary building blocks or about the response times of these building blocks. Delay-insensitive circuits offer the potential of numerous advantages. Because they are relatively unknown, further research is needed on finding the right balance of these advantages against possible penalties in circuit performance and area.

Chip design is abstracted to the design of systems consisting of processes that communicate via channels. We present two families of mathematical models for such systems of communicating processes, aimed at the study of delay-insensitivity. The results concerning the first family are not new, but those for the second family are. Also new is the framework for the models. It is based on the testing paradigm with three special ingredients: (i) systems also play the role of test (environment), (ii) there is a predicate that characterizes autonomously correct systems, and (iii) a system passes a test when the composite system comprising the system-under-test and the test environment is autonomously correct. An important correctness concern for delay-insensitive systems is absence of interference (under all possible delays). The two families differ in the nature of processes from which systems are built and in the choice of correctness concerns.

The development of models sketched below applies to both families. On the space $\mathcal{SYS}$ of systems, a composition operator *par* is given that combines two systems into one larger system. The testing paradigm induces a refinement relation *sat* and an equivalence relation *equ* on $\mathcal{SYS}$. System S is a refinement of system T when the set of tests that S passes contains that of T. Two systems are equivalent when they refine each other, that is, when they pass the same tests. This yields a pre-abstract model $\langle \mathcal{SYS}; par, sat \rangle$, for which *equ* is a congruence relation. A quotient model is then obtained by dividing out *equ*. An isomorphic model is $\langle \mathcal{DI}; \|, \sqsupseteq \rangle$, where $\mathcal{DI}$ is a set of processes. This is a (fully) abstract model. On this model a reflection operator is defined, in terms of which the design equation can be solved.

The first family of models reformulates knowledge that was developed by early workers including Muller, Seitz, and Clark et al., and cast in terms of Trace Theory by van de

Snepscheut, Udding, Ebergen, Schols, Verhoeff, and Dill among others. In its context we also present several applications. Furthermore, we point out some limitations, such as the impossibility to deal with progress. The second family improves and extends the first.

New in the second family of models is the possibility to express progress properties of processes. This is done by dividing the allowed states of a process into three categories: (i) ∇-states where the obligation for progress lies with the process (by sending output), (ii) Δ-states where the obligation for progress does not lie with the process but with the environment (by providing input), and (iii) $\Box$-states without progress obligation. A system suffers from deadlock when there exists a reachable state such that no process is in a ∇-state and at least one process is in a Δ-state. Absence of such deadlock is imposed as an additional correctness concern. The result is a pre-abstract model that extends the pre-abstract model of the first family.

Aforementioned set $\mathcal{DI}$ of the corresponding fully abstract model can be characterized in several ways. In case of the second family, the characterization of $\mathcal{DI}$ that we give by means of extended JTU-Rules is new. We also give a new abstract model for the second family in terms of enhanced characteristic functions. The enhancement consists of making the codomain of these characteristic functions a simple algebra of five objects rather than the two-valued Boolean algebra. These characteristic functions enable us to formulate the extended JTU-Rules concisely.

Finally, we give a classification of nondeterminism related to output. Determinism is defined on the basis of refusal sets, which are familiar from the failures model for Hoare's CSP. Refusal sets, however, are a derived concept in our model and not fundamental as in the failures model. Our set of deterministic processes is closed under composition. New is the distinction that we make between static and dynamic nondeterminism. Static nondeterminism corresponds to freedom in a specification that a designer may still eliminate. Dynamic nondeterminism cannot be eliminated because it depends on the interaction with the environment. An arbiter is a typical example of a process with dynamic nondeterminism. The set consisting of the deterministic and the statically nondeterministic processes is also closed under composition.

This research is of importance because a piece of knowledge in the field of delay-insensitive systems has been formulated and expanded into a uniform theory. The theory provides new insights in this field and improves our ability to transfer knowledge. Finally, the theory should be of help for choosing building blocks and for the development of better design methodologies and tools.

Samenvatting

Dit onderzoek vindt zijn motivering in het ontwerp van digitale geïntegreerde schakelingen ("chips") en systemen opgebouwd uit meerdere chips. Chips zijn doorgedrongen tot alle aspecten van ons leven. Het timing-probleem speelt een fundamentele rol bij het ontwerp van zulke schakelingen. Verschillende oplossingsmethoden zijn beschikbaar voor dit probleem. In een vertragingsongevoelige schakeling wordt het timing-probleem opgelost door ervoor te zorgen dat de correctheid niet afhangt van veronderstellingen omtrent vertragingen in de verbindingsdraden tussen de elementaire bouwstenen of omtrent de reactietijden van deze bouwstenen. Vertragingsongevoelige schakelingen beloven tal van voordelen. Hun relatieve onbekendheid vereist verder onderzoek om de juiste balans tussen deze voordelen en mogelijke nadelen qua prestatie en oppervlakte te vinden.

Chipontwerp abstraheren we tot het ontwerp van systemen bestaande uit processen die via kanalen met elkaar communiceren. We presenteren twee families van wiskundige modellen voor zulke systemen van communicerende processen om vertragingsongevoeligheid te bestuderen. De resultaten met betrekking tot de eerste familie zijn niet nieuw, maar voor de tweede familie wel. Ook nieuw is de opzet van de modellen. Deze is gebaseerd op het testing-paradigma met drie extra ingrediënten: (i) systemen vervullen ook de rol van test(omgeving), (ii) er is een predikaat dat autonoom correcte systemen karakteriseert en (iii) een systeem slaagt voor een test indien het samengestelde systeem bestaande uit het systeem-onder-test en de testomgeving autonoom correct is. Een belangrijke correctheidseis voor vertragingsongevoelige systemen is afwezigheid van interferentie (bij alle mogelijke vertragingen). De twee families verschillen in de aard van de processen waaruit systemen zijn opgebouwd en in de keuze van correctheidseisen.

De hieronder geschetste ontwikkeling van modellen is voor beide families hetzelfde. Op de ruimte $\mathcal{SYS}$ van systemen wordt een compositie-operator *par* gegeven die twee systemen verbindt tot één groter systeem. Het testing-paradigma induceert een verfijningsrelatie *sat* en een equivalentierelatie *equ* op $\mathcal{SYS}$. Systeem S is een verfijning van systeem T indien de verzameling van tests waarvoor S slaagt die van T omvat. Twee systemen zijn equivalent indien ze verfijningen van elkaar zijn, dat wil zeggen indien ze slagen voor dezelfde tests. Dit levert een pre-abstract model $\langle \mathcal{SYS}; par, sat \rangle$, waarvoor *equ* een congruentierelatie is. Een quotiënt model wordt dan verkregen door uitdelen naar *equ*. Een hiermee isomorf model is $\langle \mathcal{DI}; \|, \sqsupseteq \rangle$, waarbij $\mathcal{DI}$ een verzameling processen is. Dit is een volledig-abstract model. Hierop is een reflectie-operator gedefinieerd in termen waarvan de ontwerpvergelijking opgelost kan worden.

De eerste familie modellen herformuleert kennis die ontwikkeld is door Muller, Seitz en Clark et al., en later in termen van Tracetheorie is geformuleerd door van de Snepscheut, Udding, Ebergen, Schols, Verhoeff en Dill. We geven in deze context ook een aantal toepassingen. Verder wijzen we op enkele tekortkomingen, zoals de onmogelijkheid om voortgang te behandelen. De tweede familie vormt een verbetering en uitbreiding van de eerste.

Nieuw in de tweede familie modellen is de mogelijkheid om voortgangseigenschappen van processen uit te drukken. Dit gebeurt door de toegestane toestanden van een proces in drie klassen op te delen: (i) ∇-toestanden waarbij de verplichting tot voortgang bij het proces ligt (door uitvoer te produceren), (ii) Δ-toestanden waarbij de verplichting tot voortgang niet bij het proces ligt maar bij de omgeving (door invoer aan te bieden) en (iii) $\Box$-toestanden zonder voortgangsverplichting. Een systeem lijdt aan deadlock indien een toestand bereikbaar is waarbij geen enkel proces in een ∇-toestand verkeert en ten minste één proces in een Δ-toestand is. Afwezigheid van deadlock wordt als extra correctheidseis opgelegd. Het resultaat is een pre-abstract model dat een uitbreiding vormt van het pre-abstracte model in de eerste familie.

Bovengenoemde verzameling $\mathcal{DI}$ van het bijbehorende abstracte model kan op een aantal manieren gekarakteriseerd worden. De karakterisering van $\mathcal{DI}$ die we in het geval van de tweede familie in de vorm van uitgebreide JTU-regels geven, is nieuw. We geven voor de tweede familie ook een nieuw abstract model in termen van verrijkte karakteristieke functies met als codomein een eenvoudige algebra op vijf objecten in plaats van de gebruikelijke tweewaardige Boole-algebra. Deze karakteristieke functies stellen ons in staat om de uitgebreide JTU-regels compact te formuleren.

Tenslotte geven we een classificatie van nondeterminisme met betrekking tot uitvoer. Determinisme wordt gedefinieerd op basis van 'refusal sets', die ook bekend zijn van het 'failures' model voor Hoares CSP. 'Refusal sets' zijn echter een afgeleid begrip in ons model en niet fundamenteel zoals bij het 'failures' model. Onze verzameling van deterministische processen is gesloten onder compositie. Nieuw is het onderscheid dat we maken tussen statisch en dynamisch nondeterminisme. Statisch nondeterminisme komt overeen met vrijheid in een specificatie die door de ontwerper geëlimineerd kan worden. Dynamisch nondeterminisme kan niet bij ontwerp geëlimineerd worden omdat het afhangt van de interactie met de omgeving. Een arbiter is een typisch voorbeeld van een proces met dynamisch nondeterminisme. De verzameling bestaande uit de deterministische en de statisch nondeterministische processen is ook gesloten onder compositie.

Dit onderzoek is van belang omdat een stuk kennis op het gebied van vertragingsongevoelige systemen in een uniforme theorie geformaliseerd en vervolgens uitgebreid is. Verder verschaft deze theorie nieuwe inzichten in het vakgebied waardoor kennis hieromtrent beter over te dragen is. Tenslotte dient de theorie te helpen bij het kiezen van bouwstenen en bij het ontwikkelen van betere ontwerpmethoden en -gereedschappen.

Curriculum Vitae

1958	Born on October 24 in Rijswijk, Zuid-Holland (NL)
1960–61	Cleveland, Ohio (USA)
1965–70	Elementary school: van Nijenrodeschool, The Hague (NL)
1970–76	High school: Hertog-Jan College, Valkenswaard (NL) Diploma: Gymnasium-β
1976–85	University: Eindhoven University of Technology (NL) Diploma: Master's Degree in Mathematics
1977–80	Teaching assistant on various subjects at EUT
1980–82	Software developer at Vollwood Computer B.V., Waalre (NL)
1982–85	Research assistant on various projects at EUT
1985–87	Teaching assistant doing Ph.D. research at EUT
1987–94	Assistant professor at EUT, Dept. of Math. and C.S.
1988	Visiting research associate at Washington University, Dept. of C.S. in St. Louis, Mo. (USA)

Current address:

Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven
The Netherlands
E-mail: wstomv@win.tue.nl

Chapter 1

Introduction

Information-processing tools made their appearance a long time ago. For ages, timekeeping relied on the sundial and computing on the abacus. Not until the 17th century did timekeeping and computing benefit from new ideas. Christiaan Huygens built the first pendulum-driven clock in 1656 and later his invention of the balance wheel led to the pocket watch. Computing was revolutionized by John Napier's introduction of logarithms in 1614 and the invention of the slide rule in 1622 by William Oughtred. Slide rules reigned until they were replaced by scientific pocket calculators in the mid-1970s.

An information processor interacts with its environment by signals, and enforces a (useful) relationship between these signals. It is said to be *analog* when the signals vary continuously in space, time, or content, as with the sundial and slide rule. This contrasts with *digital* devices, like the abacus and pocket calculator, based on discrete signals, which in that case are also called symbols. The distinction 'digital' versus 'analog' is somewhat contrived, since there are all sorts of hybrid forms as well. It is a recent accomplishment that, through appropriate converters, all information processing can be translated into the digital realm (think of the Compact-Disc technology). Nowadays digital devices are mostly implemented electronically. An electronic information processor is often called a circuit; a digital circuit is an electronic realization of a symbol manipulator.

A digital electronic circuit consists mostly of switches and interconnection wires. The switches were at first implemented by *electromechanical relays*, the size of an average thumb and a switching time in the order of milliseconds. Later they were replaced by *vacuum tubes*, which were about the same size as a relay but with switching times in the order of microseconds. Shortly after World War II, in 1948, the *transistor* was invented. Fully packaged it had the size of a pea and could switch in the order of ten nanoseconds. A major breakthrough was the development of techniques to integrate a number of transistors and their connecting wires on a single silicon "chip", aptly called an *integrated circuit* (IC). It is stated in [GD85] that 'since 1961 the number of transistors that can be successfully fabricated on a single chip has doubled almost every year'. Currently, the state of the art is represented by 64 Mbit dynamic memories with close to 10^8 transistors and the DECchip 21064 (the 'Alpha'), which is a full 64-bit microprocessor implemented by 1.68 million transistors (see [DEC93]).

Modern circuits are not only complex at the structural level: also their behavior has become much more complex because of the increased degree of parallelism. In older microprocessors all operations were totally sequenced. Since we are approaching the upper limit of what is sequentially achievable, modern circuits must rely on parallelism to gain further speed. For instance, DEC's Alpha chip mentioned above contains separate instruction and data caches, and separate pipelined integer and floating-point execution units, and it involves 'dual instruction issue'.

The large degree of integration on chips can only be realized by complicated and expensive fabrication processes. From a geometric layout, giving the precise location and size of each transistor and wire, a set of enlarged photographic masks is produced. These masks are used in numerous physical and chemical processing steps to transfer the structural details of the layout onto a silicon wafer, the size of a compact disc. Each wafer, containing some hundred copies of the circuit, is cut into individual chips, which are then put into packages, bonded to the external pins, and sealed. IC production involves inherently stochastic steps; that is why tests are required along the way to identify defective circuits.

Needless to say, a circuit must be designed with great care before it is submitted for production. Delay-insensitive circuits are intended to improve our ability to make reliable and efficient circuit designs. The theory of delay-insensitive systems is applicable to information-processing systems in general, also to systems that are not electronically implemented.

1.1 Formal Framework

In this section, we outline the development of our theory of delay-insensitivity. Our major motivation for the study of delay-insensitivity is its relevance to the design of digital integrated circuits, to be explained in more detail in the next chapter. However, this is not the only design context where the notion of delay-insensitivity is applicable. The models presented in later chapters and--especially the methods used to construct them, are largely application independent. Only in a few isolated places are decisions based on the application to digital circuit design. These will be pointed out where relevant. Let us now begin with a few philosophic points.

Three Kinds of Models

A model should encompass everything that one cares to express about what is being modeled. The aim is to omit irrelevant details, though it may not always be clear in advance where to draw the boundary between relevant and irrelevant. A **mathematical model** may be set up as some sets of objects, and some operators and relations on these sets. Such a model can also be viewed as an algebra. A well-known example is the real number system, with the set of real numbers as objects, addition and multiplication as operators, and the usual ordering as relation. In the case of delay-insensitive digital circuits, the objects are networks of communicating processes, for which parallel composition is an operator, and

satisfactory substitutability, also known as refinement, is a relation.

We distinguish three levels of abstraction when using models. These are, in order of increasing abstraction:

pre-abstract, (fully) abstract, and axiomatic.

At the **pre-abstract** level, there are possibly irrelevant distinctions between objects; that is, we consider some distinct objects equivalent for the intended application of the model. At the **fully abstract** level, distinct objects are inequivalent, but the objects themselves still may have irrelevant structural detail. At the **axiomatic** level, the objects have no explicit structure; they are implicitly characterized by axioms on their operators and relations. When moving from a pre-abstract to a fully abstract model, one abstracts from irrelevant object distinctions, by identifying equivalent objects. When moving from a fully abstract model to an axiomatization, one abstracts from irrelevant object structure, that is, from irrelevant distinctions between different *models*, thus identifying isomorphic models.

For example, in the case of the real number system, the model with *Cauchy sequences of rational numbers* as objects is at the pre-abstract level: many Cauchy sequences are equivalent as "real numbers". The model with *Dedekind cuts in the rational numbers* as objects is at the fully abstract level: each real number is modeled by a unique Dedekind cut. But the Dedekind cut itself is irrelevant to the notion of "real numbers", since they can also be defined using, for instance, certain *infinite decimal expansions*. Axiomatically, the real number system can be defined (up to isomorphism) as *the complete ordered field*, which abbreviates a list of axioms. We refer to [End77, ML86] for details.

Of course, even the objects in a pre-abstract model are ultimately defined in terms of axiomatically postulated objects to avoid an infinite regress. This shows that pre-abstract models are also "very" abstract. We use **set theory** as a foundation, albeit in an implicit way. (By the way, even an axiomatization can still have irrelevant structure, in that distinct lists of axioms can define the same class of models. This shows that axiomatic characterizations are not necessarily the "most" abstract descriptions.)

The use of a pre-abstract model is often justifiable by its close relationship to intuition or to physics, thereby lending some plausibility to the definitions of the objects, operators, and relations involved. Fully abstract models can provide additional insight by the way in which they eliminate the irrelevant object distinctions. They are useful for proving fundamental properties that later can serve as axioms. They also embody a (relative) consistency proof of a tentative axiomatization. An axiomatic characterization is useful because it provides a consistent framework for carrying out abstract proofs, which do not rely on ad hoc structural properties of the objects. The natural development of a theory often goes from a pre-abstract model, via a fully abstract model, to an axiomatic characterization. We also follow this line but stop short of the last step.

Testing Paradigm

Partly as an experiment, we deviate from the "standard" development of computational models. The "standard" procedure we have in mind (see, for example, [LS84]), introduces

a set of *syntactic* entities, say programs, and assigns to these programs a "*meaning*" from a set of *semantic* entities. Each "meaning" *satisfies* certain *specifications* taken from yet another set. In this setting, program correctness translates into the question whether the program's "meaning" satisfies the given specification.

The "standard" terminology is, at best, misleading. The suggested distinction between syntax and semantics makes no sense, because the question 'what is the meaning of ...?' is utterly uninteresting [Pop83, pp. 261–265]. (Next thing, one will ask for the meaning of the meaning of ...?) A model should cover *everything* one cares to express and should leave no room for such questions. That is why operators and relations are to be incorporated, including such relations expressing that a program satisfies a specification. Of course, these can be defined in terms of auxiliary concepts, such as labeled state-transition systems or predicate transformers. But these auxiliary concepts hardly deserve the name "meaning". So, we will not introduce "meanings" as separate entities.

Furthermore, we wish to dispense with the distinction that is made between programs and specifications. A program can only operate when placed in some environment, together with which it forms an **autonomous system**. The environment is also taken to be a program. Program correctness is now defined by giving criteria for the correctness of autonomous systems. In the case of digital integrated circuits, correctness criteria – such as absence of computation interference – ultimately derive from physics, that is, from physical models. The relevant **correctness criteria** are captured by relation *pass* on programs, where P *pass* E expresses that program P operates correctly in environment E. When dealing with networks of communicating processes, it is natural to confront program and environment with each other by parallel composition. Of course, if one insists on "standard" terminology, then for a given program P, the set of E's satisfying P *pass* E, could be considered the "meaning" of P.

Operation of a program within an environment can also be interpreted as a form of **observation** [Hoa85, OH86] or **testing** [dNH83, Hen88]: P *pass* E expresses that program P passes the test under environment E. Program E is then called a testing environment, test, observer, or experimenter. We can now define when program P is a **satisfactory substitute** for program Q, denoted by P *sat* Q, namely when P passes at least the same tests as Q does. We note that this is based on a **demonic** attitude towards nondeterminism. If Q is viewed as a specification, then P *sat* Q may also be interpreted as 'P satisfies Q' or 'P implements Q'. Other common pronunciations of P *sat* Q are: 'P is at least as good as Q', 'P realizes Q', 'P refines Q', and 'P conforms to Q'. Programs P and Q are **(testing) equivalent** when they are satisfactory substitutes for each other, that is, when they pass exactly the same tests.

The appearance of an equivalence notion, instead of an equality, indicates that we are dealing with a pre-abstract model here. The next step is to "factor out" this equivalence and to study the related fully abstract model. The emphasis is on the development and analysis of fully abstract models. As a final step, an axiomatic characterization could be sought, though we will not complete that part of the journey.

1.2 Chapter Overview

The last section of this chapter covers some notational issues. Chapter 2 provides background information on the design of digital electronic circuits and motivates our interest in delay-insensitivity. Chapter 3 introduces an informal model that enables us to pose questions without delving into technical matters too much.

In Chapter 4 we present a formal model concerning delay-insensitive systems, called the DI Model. Actually, the DI Model encompasses two closely related models. We start with a pre-abstract model and subsequently develop a fully abstract model. The pre-abstract model is founded on a set of *processes*. The objects of interest are process networks, called *systems*. The set of systems is sufficiently rich to contain objects that serve as *specification* as well as objects that play the role of *implementation*[1] The distinction between implementation and specification, however, falls outside the scope of the theory; it exists in the user's mind only. Systems can be composed into larger, more complex, systems. This *composition operator* models the connection of subsystems by wires. A correctness criterion on closed systems forms the basis for the *comparison* of systems employing the testing paradigm. It turns out that a related fully abstract model can be obtained—after a minor correction—as a subset of processes. Most of the results in this chapter, and also of the next chapter for that matter, are not new. However, we take a novel approach to the presentation of the model.

Chapter 5 discusses several applications of the DI Model and reveals some of its limitations. We extend the DI Model in Chapter 6 to address one of these limitations, namely by incorporating some form of progress requirement. This Extended DI Model is, again, developed from a pre-abstract model into a fully abstract model. In contrast to the DI Model, all results concerning the Extended DI Model are believed to be new. Chapter 7 is more technical in nature and shows how a fully abstract model can, in fact, be derived from the pre-abstract model. It is based on a small algebra for trace labels. The classification of processes in terms of output nondeterminism is the subject of Chapter 8. This classification helps us to better understand some features of the DI Models. It also gives rise to an interesting distinction between static and dynamic nondeterminism. Such a distinction is intuitively appealing but cannot be made in, for instance, the Failures Model for CSP.

Finally, Chapter 9 completes our treatment of delay-insensitivity. We look back at the results and how they were obtained, and we summarize the relationship with the work of others. We also point out some issues that were ignored. Along the way we suggest topics for further research and development.

1.3 Notational Conventions

Function application is written with an infix dot: $f.x$ is the image of x under application of f. **Function composition** $\circ$ is defined by $(f \circ g).x = g.(f.x)$.

[1] With 'implementation' we do not refer to some *physical* realization, but to a design with more (internal) structure than a specification, for instance, in terms of a network of components.

A slightly unconventional notation for **variable-binding** constructs is used. It will be explained here informally. **Universal quantification** is denoted by

$$(\forall\, \ell : D : E)\ ,$$

where $\forall$ is the quantifier, ℓ is the list of bound variables, D is the domain predicate, and E is the quantified expression. Both D and E will, in general, contain variables from ℓ. Predicate D delineates the domain of the bound variables. Expression E should be well-defined for all values of the bound variables that satisfy D. When D is simply *true* or clear from the context it is often omitted. For instance, when variables x and y range over function f's domain, we can express that f is injective by

$$(\forall\, x, y :: f.x = f.y \ \Rightarrow\ x = y)\ . \tag{1.1}$$

Existential quantification is likewise denoted by quantifier $\exists$. In the case of **set formation** we write

$$\{\ell : D : E\}$$

to denote the set of all values E obtained by substituting values that satisfy D for the variables in ℓ. By way of example, consider for natural number k, the set $\{n : k \leqslant n : k^n\}$ of all powers of k with integral exponent at least k. In the conventional notation this set might be written as $\{k^n | k \leqslant n\}$, where it is unclear which variables are bound.

For expressions E and G, an expression of the form $E \Rightarrow G$ will at times be proved in a number of steps by the introduction of intermediate expressions. For instance, we can prove $E \Rightarrow G$ by proving $E \equiv F$ and $F \Rightarrow G$ for some expression F. This derivation is recorded as

$$\begin{array}{ll} & E \\ \equiv & \quad \{\ \text{hint why } E = F\ \} \\ & F \\ \Rightarrow & \quad \{\ \text{hint why } F \Rightarrow G\ \} \\ & G \end{array}$$

In this way we avoid writing down intermediate expressions like F twice. For example, a proof of '$f \circ g$ is injective if f and g are injective' might go as follows. For x and y in the domain of $f \circ g$, hence in the domain of f, we derive

$$\begin{array}{ll} & (f \circ g).x = (f \circ g).y \\ \equiv & \quad \{\ \text{definition of } f \circ g\ \} \\ & g.(f.x) = g.(f.y) \\ \Rightarrow & \quad \{\ g \text{ is injective: definition (1.1) with } x, y := f.x, f.y\ \} \\ & f.x = f.y \\ \Rightarrow & \quad \{\ f \text{ is injective}\ \} \\ & x = y \end{array}$$

The notation '$x, y := E, F$' stands for the simultaneous substitution of E and F for x and y respectively.

Chapter 2

Motivation

Our interest in delay-insensitivity first arose in the context of digital electronic circuits, especially in the form of integrated circuits. In the first section we take a closer look at one of the main problems encountered in the design of digital integrated circuitry, namely the timing problem. Each of the next three sections discusses a different approach to the timing problem; delay-insensitivity is one of them. The final section is about specifications in which time only plays a role for sequencing.

2.1 Timing Problem

A digital integrated circuit can be viewed as a network of transistors (that is, electronic switches) interconnected by wires. These circuit elements interact by voltage changes, also called signals. The operations that take place are propagation, duplication, and switching of signals. By the very nature of the circuit elements, these operations are **continuous phenomena** expressible in terms of partial differential equations.

Digital circuits, however, are intended to carry out **discrete computations**, as opposed to continuous or analog computations. This is a fundamental issue in the design of digital circuits and the source of a number of problems. The issue can be illustrated with the operation of a simple digital circuit, namely an *OR*-gate.

2.1.1 Example An *OR*-gate has two input ports, say a and b, and one output port, say c (see Figure 2.1). We distinguish two special voltage levels at these ports: *high* and *low*, where *high* exceeds *low*. We say that a port is *true* when its voltage level is at least *high*, and that it is *false* when its voltage level is at most *low*.

The *OR*-gate strives to make its output equal to the disjunction (boolean OR) of its inputs. This can be accomplished with transistors and wires, but we need not know how that is done. By the way, notice that nothing is specified about the *OR*-gate's output in case an input is at a voltage level between *low* and *high*, which does not correspond to a boolean value.

We now consider three computation scenarios of the *OR*-gate. All three start in the stable state where a is *true*, b is *false*, and, consequently, c is *true* (see Figure 2.1).

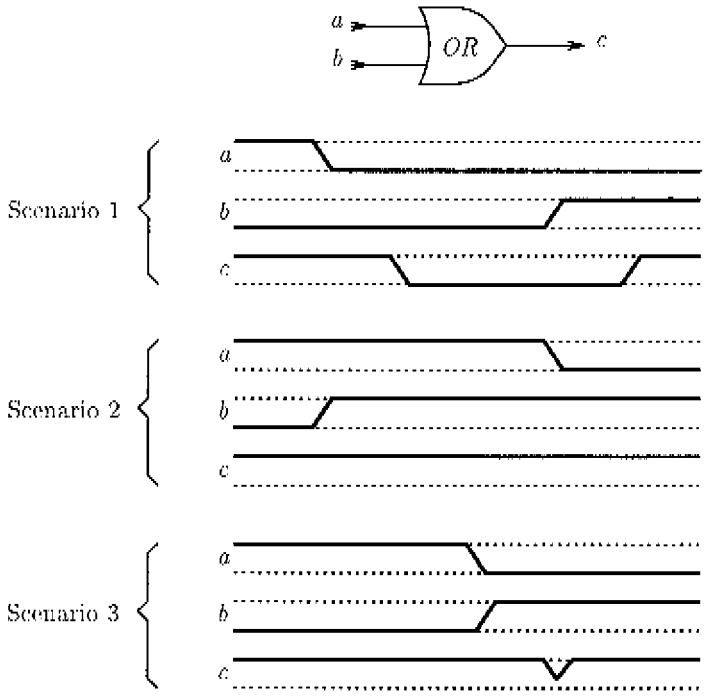


Figure 2.1: Three computation scenarios for the *OR*-gate

In Scenario 1, first a changes to *false*, subsequently c changes to *false*, next b changes to *true*, and then c changes back to *true* again. In Scenario 2, b first changes to *true*, c remains *true*, followed after some time by a changing to *false*, c still remaining *true*. Thus, depending on the relative timing of the changes on a and b, as illustrated in Scenarios 1 and 2, there is either a downward pulse on c or no change at all.

Consider the function that maps $t_b - t_a$, the time from change of a to change of b, into the minimum voltage attained by c after the first change. This function, being defined in terms of partial differential equations, is continuous when some (mild) restrictions on part of the circuitry are met. On account of the Intermediate Value Theorem (known from Analysis for continuous functions), the continuity of this function implies the existence of Scenario 3, in which also a changes to *false* and b changes to *true*—suitably timed with respect to each other — causing c to generate a considerable voltage dip with its minimum somewhere between *low* and *high*. Such a dip may elicit all sorts of complicated behavior at the receiving end.

The lesson is that, in spite of its intended digital simplicity, an *OR*-gate is a subtle piece of circuitry as far as its behavioral analysis is concerned. Note that the argument above holds for *every* realization of an *OR*-gate. ∎

Apparently, the **relative timing** of signals critically influences the behavior of digital circuits. The designer needs to control the relative timing of signals carefully, even when speed is no concern and functional correctness is the only concern. This is the **timing problem**.

2.2 Traditional Solution

Relative timing is directly determined by the **operating speed** or **delay** of the circuit elements involved. The delay characteristics of a circuit depend on such diverse factors as

1. circuit logic and topology,
2. geometric layout,
3. scaling and integration technology,
4. fabrication stochastics,
5. environmental conditions,
6. metastability resolution, and
7. aging.

Here are some examples.

1. Logic: A two-input *OR*-gate is usually faster than a three-input *OR*-gate, even with one of the latter's inputs fixed at *false*. Topology: When a four-input *OR*-gate is built from three two-input *OR*'s, its operating speed will depend on whether they are connected linearly or as a balanced tree.

2. Geometry: Even if the four-input *OR*-gate is implemented as a balanced tree, the exact layout of this tree will also affect the operating speed. Imagine putting each *OR*-gate in a different corner of the chip, and connecting them by very long wires.

3. Technology: Each integration technology has its own characteristics. CMOS switches are relatively slow and economic in operation. Gallium-arsenide transistors are fast but require a cool environment. See [Sei79] for a discussion of scaling.

4. Fabrication: The manufacturing process is not completely controllable. Hence, circuit elements manufactured from the same design by the same technology may vary in characteristics, such as operating speed.

5. Environment: The operating speed depends directly on such factors as temperature and power supply voltage.

6. Metastability: All sufficiently smooth systems—and in nature that includes most systems—with at least two stable states have at least one **metastable** state (see [Hur75, Mar81, KC87b, KC87a]). A metastable state is like the "middle" position of a toggle switch, where it is carefully balanced between on and off (also see Figure 2.2). On the one hand, a metastable state persists when the system is left to itself. On the other hand, a small deviation from such a state will make the system diverge from it and move towards one of its stable states.

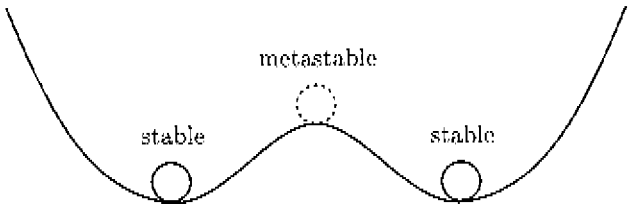


Figure 2.2: Stable and metastable states in the field of gravity

In practice, a macroscopic system like an integrated circuit is never "left to itself": there are always small perturbations due to noise. Hence, such a system will leave a metastable state with probability one. The problem is that there is no upper bound on how long it will stay in or near the metastable state before diverging.

A flip-flop is a (digital electronic) system with two stable states (on/off, set/reset, 0/1, *true/false*, whatever they are called). Under the right circumstances any flip-flop can be brought sufficiently close to, or even into, a metastable state, where it hesitates between 0 and 1 (this is known as the **glitch phenomenon**, see [CM73, Sei80]). The duration of this hesitation is unpredictable and can be arbitrarily large; thus, it translates into variable operating speed.

7. Aging: As circuits grow older, their characteristics, including operating speed, slowly change.

In summary, delays are not easily controllable because they depend on many factors. These dependencies increase the overall complexity of the design task. Nevertheless, the designer somehow needs to immunize circuit designs against delay variability. With current integration technologies this poses a serious problem because of the huge functional complexity of the circuits that may be manufactured.

The **traditional solution** to the timing problem introduces severe constraints on the delays, in order to make their effects tractable. Most often a central **clock** serves as a global event sequencer and time reference, simplifying matters a little by trading two-sided for one-sided bounds ('wait until the next clock tick for the *OR*-gate's output to stabilize'; see [Sei80, p. 225] for an explanation).

The traditional solution, however, has several disadvantages. The constraints imposed on delays directly translate into restrictions on all parameters that influence delays. The

substitution of a functionally equivalent but faster subcircuit (item 1 above) may require a complete redesign. Layout freedom (item 2) is limited. Also rescaling or the employment of new integration techniques (item 3) may require a redesign. Complicated tests must be performed to eliminate circuits suffering from fabrication failures (item 4). Power supply and ambient temperature (item 5) must be kept within strict limits. If a clock is present, then its period must be tuned to accommodate the worst case and it must be properly distributed. When metastability (item 6) plays a role, clock tuning is inherently impossible and one has to settle for a circuit with at best probabilistic reliability.

2.3 Ideal Solution

An **ideal solution** to the timing problem is based on decoupling correctness from delay variations altogether. That is, one sees to it that circuits are correct even under arbitrary, uncorrelated variations in all delays. The resulting circuits are called **delay-insensitive**.

This ideal solution is very attractive since it does not suffer from the disadvantages mentioned in connection with the traditional solution. It promises freedom in subcircuit substitution (item 1), layout (item 2), scaling and integration technology (item 3), and operating conditions (item 5). Testing is still required to filter fabrication failures (item 4), but tests need not be so complicated. The concept of a clock is irrelevant under this ideal solution, so clock tuning and distribution, and the harmful consequences of the glitch phenomenon (item 6) can be avoided.

Of course, delay variability not only affects functional correctness, but also directly relates to **performance**. Therefore, efficiency considerations may reintroduce limits on delay variability even in the ideal solution. What we have gained is a separation of concerns: correctness independent of delays. Of course, performance does depend on delays, but also on the choice of "algorithm".

In general, however, it is hard to build transistor circuits whose correctness is completely independent from delays—if it is at all possible (see [vdS85, p. 77] and [Seg91]).

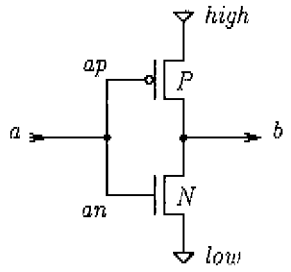


Figure 2.3: CMOS inverter

2.3.1 Example Consider a very basic circuit (see Figure 2.3): the CMOS inverting amplifier with input port a and output port b (see [WE93] for details on the operation of

CMOS transistors). We adopt the same voltage level conventions as with the *OR*-gate of Example 2.1.1. In the stable state where a is *false* and b is *true*, transistor P is conducting and transistor N is non-conducting.

Let us assume that transistor P switches much more slowly than transistor N, or, what amounts to the same thing, that wire ap is much slower than wire an. When a is now changed from *false* to *true*, the inverter temporarily gets into a state where both transistors are conducting. This short-circuits the power supply and possibly destroys the inverter when lasting too long (or, at the least, wastes energy).

Obviously, the correctness of the inverter circuit depends on a suitable matching of delays, constraining such aspects as the circuit's layout, etcetera. Therefore, this inverter circuit is not fully delay-insensitive (assuming that it was intended for computating the boolean *NOT*; see the next section for the importance of a specification). ∎

2.4 Two-Stage Solution

Instead of aiming at correctness independent of all delays, it seems more realistic to accept some dependencies. This is incorporated in the following **two-stage solution** to the timing problem [Cla67, Kel74, Sei80, MFR85, vdS85].

First, a small but sufficiently expressive set of simple **building blocks**, whose correctness may depend on the size of internal delays to some extent, is designed. These building blocks are then used to design larger circuits, whose correctness is independent of the externally observable delays of the building blocks and interconnecting wires.

The two-stage solution yields circuits that still give a fair amount of freedom in layout, substitution, scaling, and integration technology, etcetera, because it localizes the timing problem inside the few sufficiently simple building blocks, where it needs to be solved only once. See Chapter 9 for other advantages and disadvantages of this approach.

Circuits based on the two-stage solution are often called **delay-insensitive**, **speed-independent**[1], or **self-timed**. It does not mean very much if someone states, in isolation, that a circuit is delay-insensitive. Even the correctness of traditional circuits is insensitive to delay variations to some extent (otherwise, they would be quite useless). One should elaborate the statement by indicating (i) *which delays* in the circuit are allowed to change (ii) by *how much*, (iii) in *what relationship* to other delays, and (iv) without affecting correctness with respect to *what specification*. In case of the ideal solution, the answers to (i), (ii), and (iii) are: 'all', 'an arbitrary amount', and 'uncorrelated' respectively; we will come back to (iv) in a moment.

In case of the two-stage solution, the answer is less straightforward. For one thing, delays in all wires interconnecting the building blocks are allowed to change in an arbitrary uncorrelated fashion. Within the building blocks only certain changes are allowed, for example, those changes that affect only the externally observable delays, and, depending on the particular implementation, possibly others as well.

[1] The term 'speed-independent' is usually reserved for a more restricted class of circuits.

However, we will be interested in *design with*, not *design of*, such building blocks. When treating the building blocks as black boxes the two-stage designs will also be considered delay-insensitive.

2.5 Specifications without Time Metric

So far, we have not said much about the kind of specifications against which circuit correctness is verified. It is obvious that time can play only a limited role in specifications, for, otherwise, no delay-insensitive circuit can satisfy it. For instance, it does not make sense to specify a delay-insensitive circuit in which a certain output is to be generated within one microsecond, because the wire connecting the circuit's output port to another circuit's input port may arbitrarily delay the signal anyway.

Although alternatives are possible, we will work with specifications that are completely free of a **time metric** [Sei80, vdS85, Udd84]. In such specifications only the order in which events occur, and not their precise location in time, is of importance. The events in this case are—roughly speaking—rapid monotonic voltage changes bringing about a change in boolean value. These events are called **(voltage) transitions** and they are considered **atomic events**, that is, events cannot "overlap" or occur "simultaneously".

This choice for events brings with it a restriction, which we call the **digital mode (restriction)**, on the allowed signal waveforms. Even if all input signals obey the digital mode restriction, it is still possible that the circuit's output signal violates the digital mode. This is illustrated by the non-digital pulse in Scenario 3 of Example 2.1.1. Such unwelcome signals on input ports are said to constitute **computation interference** [vdS85]. Correct usage of a delay-insensitive circuit puts a restriction on the environment's behavior as well.

2.5.1 Example In terms of orderings on voltage transitions, the *OR*-gate from Example 2.1.1 can be specified by the labeled graph of Figure 2.4. An edge labeled $a\uparrow$ stands

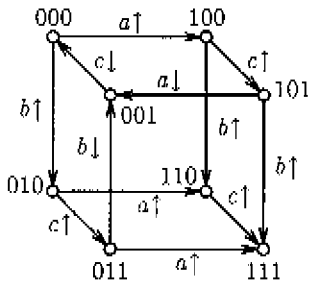


Figure 2.4: State-transition diagram for *OR*-gate

for a transition on port a from *false* to *true*, and $a\downarrow$ for a *true*-to-*false* transition. The vertexes of the graph have been labeled with the state vector abc of port values (0 for *false*, 1 for *true*).

Notice that the *OR*-gate in state 101, which corresponds to the initial state of the scenarios in Example 2.1.1, is capable of processing transitions on either input. But once one input has been received the other cannot be accommodated. In particular, a transition on b in state 001 can cause computation interference for certain settings of the delays involved. This is modeled by the absence of an edge labeled $b\uparrow$ from state 001 in the state-transition diagram. Similarly, there are no edges labeled $a\downarrow$ and $b\downarrow$ from state 111, which is a sink in the graph.

In summary, this specification prescribes restrictions on both the *OR*-gate's behavior and that of its environment. These restrictions capture the most liberal delay-insensitive usage of the *OR*-gate. In practice, the sink part consisting of states 110 and 111 is never exploited. ∎

The digital mode restriction also has consequences for a simple wire. Because of dispersion and dissipation, a sequence of two clean signal transitions at the input port of a wire can result in a non-digital pulse at the output port. This phenomenon is called **transmission interference** [vdS85]. Phrased differently: a wire can reliably process at most one voltage transition at a time.

Absence of computation and transmission interference are correctness concerns for the designer of delay-insensitive circuits. Other correctness criteria, like absence of wiring conflicts and absence of deadlock, will be discussed later.

Chapter 3

Objectives

In this chapter we informally illustrate the objectives we have in mind when developing a theory of delay-insensitive systems. The theory should enable one to argue about such things as equivalence, composition, substitution, satisfaction, and decomposition under a variety of correctness concerns. The next chapter will formalize these concepts. The development of practical tools should be served by our investigation, but is beyond the scope of this work.

The two-stage solution to the timing problem (as discussed in Chapter 2) transforms digital circuit design into the design of networks of communicating processes. The processes are to be taken from a small set of building blocks. How these building blocks are designed is not our concern here; that needs to be done only once and requires intimate knowledge of the particular implementation technology. The choice of building blocks is also left open. In fact, the theory presented here should be helpful in selecting an appropriate set of building blocks.

Communication delays and processing delays are nondeterministic parameters. As we have seen, absence of computation and transmission interference are important correctness criteria. Notice that other applications, for instance involving software, or product flows in factories, also fit in this abstract framework. However, the correctness criteria involved may be different.

3.1 Processes

We will now sketch a simple formalism and informally look at some examples, illustrating the kind of problems that we intend to address.

In this simple formalism, the interactions of a module and its environment are specified by a triple (I, O, V) satisfying the conditions listed below. Such a triple is called a process. I is the process's set of input port names and O is its set of output port names. I and O should be disjoint. V is a set, called the process's trace set, of finite-length sequences over $I \cup O$; it should be non-empty and prefix-closed. The latter means that for each trace in V all its prefixes, (initial segments) are also in V. Trace set V specifies in which order

communication actions can take place, that is, in which order the process can send and receive signals via the ports. This works as follows.

Let t be the sequence of communication actions thus far performed by the process. Note that communication actions are considered atomic events and can therefore be sequenced. Sequence t is also called the current trace of the process. Initially, t is the empty sequence ε, which belongs to V by definition. If $a \in O$ and $ta \in V$ then the process can[1] produce a signal on output port a and its current trace is extended to ta. If $a \in I$ and a signal is received on input port a then the current trace is also extended to ta, regardless of whether $ta \in V$.

When $ta \notin V$ we say that there is (computation) interference at the process. This should be avoided at all costs. It is an obligation of the process's environment to see to it that the current trace remains in V (well, actually, it is the designer's responsibility to use processes in appropriate environments only). Notice that a process cannot directly prevent its environment from supplying an input signal (it may be able to do so indirectly by sending output). Nor can the environment directly force the process to produce an output signal. All the environment can do is wait for an output signal to be sent.

Examples of Processes

We illustrate our process notion by five elementary examples.

3.1.1 Example The **wire** with input port a and output port b (thus, $a \neq b$) is the process specified by triple

$$(\{a\},\ \{b\},\ \{\varepsilon, a, ab, aba, abab, ababa \ldots\}) ,$$

where the trace set consists of all alternations of a's and b's not starting with b. The wire copies each input signal on its output. Its environment should not provide the next input until it has received the output signal. This process is also denoted by $\mathtt{W}(a; b)$. In this notation the semicolon separates the input symbols on the left from the output symbols on the right. ∎

We often find it convenient to define a trace set by a state graph. A **state graph** is a directed graph with one vertex marked as initial state and every edge labeled with a symbol. Moreover, for each vertex, the edges leaving that vertex should have distinct labels. The trace set of such a state graph consists of exactly those symbol sequences obtained by writing down, in order, the labels encountered in the state graph on paths that start at the initial state. The vertexes are also called states. In diagrams, the initial state appears as solidly filled circle. For instance, the trace set of $\mathtt{W}(a; b)$ is also given by the topmost state graph in Figure 3.1.

3.1.2 Example The **I-wire** $\mathtt{I}(a; b)$ with input a and output b is defined in Figure 3.1. It is essentially a wire with an initial signal on it, that is, it can initially produce an output

[1] We are on purpose a bit vague: 'can' here simply means 'is able to', not 'is guaranteed to'.

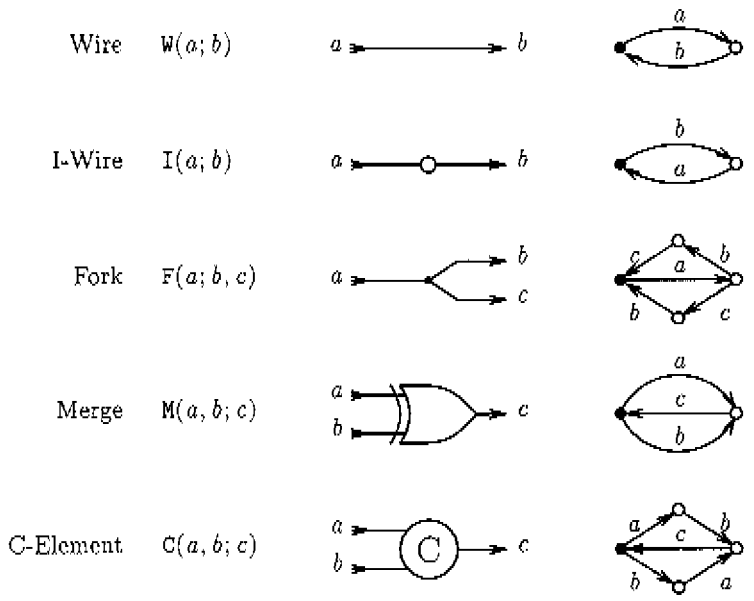


Figure 3.1: Diagrams (middle) and state graphs (right) of some processes

signal, after which it will behave like W(a; b). The environment should wait for the first output before sending the first input. ◂

3.1.3 Example The **fork** F(a; b, c) with input a and two outputs b and c is defined in Figure 3.1. It duplicates each input signal on both output ports. The environment should wait until it has seen both outputs before supplying the next input. ◂

3.1.4 Example The **merge** M(a, b; c) with two inputs a and b, and output c is defined in Figure 3.1. It duplicates any input signal on its output port. The environment should ensure mutual exclusion of the inputs. The next input may be provided only after the occurrence of the output. ◂

3.1.5 Example The **C-element** C(a, b; c) with two inputs a and b, and output c is defined in Figure 3.1. It waits until both inputs have received one signal and then produces an output signal. The order of the inputs is not prescribed. The environment should wait until it has received the output before initiating the next cycle. ◂

3.2 Systems of Processes

Input and output port names of a process are dummies in the sense that they may be renamed to obtain a related process. Thus, both W(a; c) and W(b; c) are instances of a wire

process. The port names will be used to indicate connectivity in networks.

In our simple formalism, a network (or system) is just a set of processes such that each port name occurs at most once as input and at most once as output. Consider a port name that occurs in some of the processes. If the port name occurs exactly once, then it is considered an external port of the system, available for connection to the environment. Otherwise, it occurs exactly twice (once as input and once as output) and the two ports with that same name are considered connected by a wire. We do not include such a wire explicitly in the system as a wire process, but as far as operation is concerned the connection is intended to behave like a wire process. This wire is internal to the system, that is, the communications on it are not observable by the environment.

We are interested in the analysis of the behavior of process networks. We will do so only intuitively in the remainder of this chapter by looking at some examples and raising some questions.

Examples of Systems

A set consisting of a single process is a system. It has no internal wires. We will often identify process P with the singleton system $\{P\}$.

The typewriter font will be used for symbol constants. Thus, a, b, and c are three (distinct) symbols, and W(a;b) and W(b;c) are two instances of wire processes. Figure 3.2 presents four systems that we discuss next. The dashed lines represent internal wires.

3.2.1 Example Set S_1 defined by

$$S_1 \;=\; \{\ \mathtt{W(a;c)},\ \ \mathtt{W(b;c)}\ \}$$

is *not* a system because it is malformed: port name c occurs twice as output. ∎

3.2.2 Example Set S_2 defined by

$$S_2 \;=\; \{\ \mathtt{I(a;b)},\ \ \mathtt{I(b;c)}\ \}$$

is a system. It has an external input port a, an (implicit) internal wire connecting the b-ports, and an external output port c. Even if the environment refrains from sending inputs, this system may suffer from computation interference: I-wire I(a;b) can produce a signal on port b, which subsequently can arrive at I-wire I(b;c) before the latter has produced its c-signal. Therefore, S_2 can misbehave in any environment and, hence, it is completely useless. ∎

3.2.3 Example System S_3 defined by

$$S_3 \;=\; \{\ \mathtt{F(a;b,c)},\ \ \mathtt{C(b,c;d)}\ \}$$

is, in a sense, equivalent to wire W(a;d). We reason as follows. When S_3 is provided with an a-input there is no possibility of interference and the fork may eventually produce b-

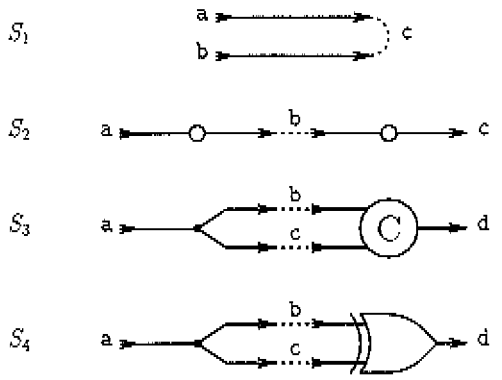


Figure 3.2: Diagrams of four systems S_1 through S_4

and c-outputs. When both b- and c-signals have arrived as inputs at the C-element, it may produce a d-output. After this, and no earlier[2], another cycle can take place.

We will be more specific about the kind of equivalence we have in mind later on. ∎

3.2.4 Example System S_4 defined by

$$S_4 \;=\; \{\; \mathtt{F(a;b,c)},\;\; \mathtt{M(b,c;d)} \;\}$$

is not equivalent to wire $\mathtt{W(a;d)}$. When provided with an a-input the system may suffer from computation interference at the merge, because mutual exclusion of its inputs is then not guaranteed. However, if the environment refrains from supplying inputs altogether, then no interference can ensue and no outputs will be produced. Hence, system S_4 is equivalent to the process $(\{\mathtt{a}\},\{\mathtt{d}\},\{\epsilon\})$. ∎

3.2.5 Example Now consider system S_5 depicted in Figure 3.3 and defined by

$$S_5 \;=\; \{\; \mathtt{C(a,b;c)},\;\; \mathtt{F(c;d,e)} \;\}\;.$$

It has external input ports a and b and external output ports d and e; an internal wire connects the c-ports. Does there exist an equivalent process?

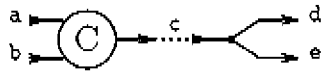


Figure 3.3: Diagram of S_5

A first attempt might lead to process P_5 whose trace set is defined by the state graph given in Figure 3.4. Vertexes labeled with the same number represent a single state of the

[2]If the environment supplies the next a-input before receiving the d-output then there is a possibility of interference.

state graph. Label 0 occurs multiply; labels 1, 2, and 3 are for reference purposes only. Process P_5 indeed captures some aspects of system S_5, but we would not want to consider it equivalent. The argument is as follows.

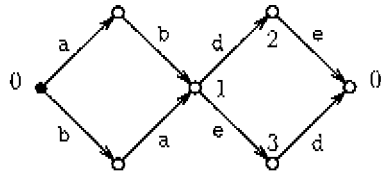


Figure 3.4: State graphs for P_5

According to P_5, interference is possible when the environment supplies the next input after receiving a *single* output from the fork (see states 2 and 3 in the state graph). Admittedly, in this state, the system cannot cope with inputs on *both* a and b, because the C-element is then enabled to output c, which may subsequently interfere with the fork. However, it can handle a *single* external input; the C-element should be "kept quiet" until the other output appeared from the fork. All the environment has to guarantee in this state is mutual exclusion of the inputs.

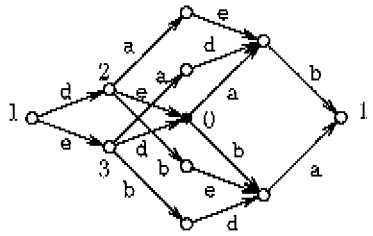


Figure 3.5: State graph for Q_5

These features are incorporated in the state graph of Figure 3.5 (note that the initial state is in the center), which defines the trace set of process Q_5. We claim that Q_5 is equivalent to S_5. This equivalence can be proved within the formal model of Chapter 4.

A different way of arguing against the equivalence of S_5 and P_5 is that an environment that has no possibility of interference with S_5 may suffer from interference with P_5. For instance, the environment obtained by exchanging the roles of input and output in Q_5 will do. On the other hand, it is the case that every environment that has no possibility of interference with P_5 also has no possibility of interference with S_5. Therefore, S_5 can be substituted for P_5 in any context without introducing any possibility of interference. One could also say that S_5 satisfies or implements specification P_5 (without being equivalent to it). ∎

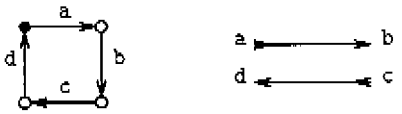


Figure 3.6: State graph for P_6 (left) and diagram of S_6 (right)

3.2.6 Example Consider process P_6 with input ports a and b, output ports c and d, and trace set defined by the state graph in Figure 3.6. It defines a four-phase communication protocol and is readily implemented by two wires, as in system S_6 shown in Figure 3.6 and defined by

$$S_6 = \{ \mathtt{W(a;b)},\ \mathtt{W(c;d)} \} .$$

However, P_6 and S_6 are obviously not equivalent: S_6 can handle an initial input via port c (without interference), but P_6 cannot. ∎

3.2.7 Example Finally, consider system S_7 and processes P_7 and Q_7 defined by

$$\begin{aligned} S_7 &= \{\ (\varnothing, \{\mathtt{a},\mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{b}\}),\ (\{\mathtt{b}\}, \varnothing, \{\varepsilon, \mathtt{b}\})\ \} , \\ P_7 &= (\varnothing, \{\mathtt{a}\}, \{\varepsilon, \mathtt{a}\}) , \\ Q_7 &= (\varnothing, \{\mathtt{a}\}, \{\varepsilon\}) . \end{aligned}$$

System S_7 has one external output port named a, and one internal wire connecting the b-ports. It consists of a process that may produce a signal on either of its output ports a or b but not on both, and a process that is willing to receive a signal on its only input port b and that does nothing afterwards.

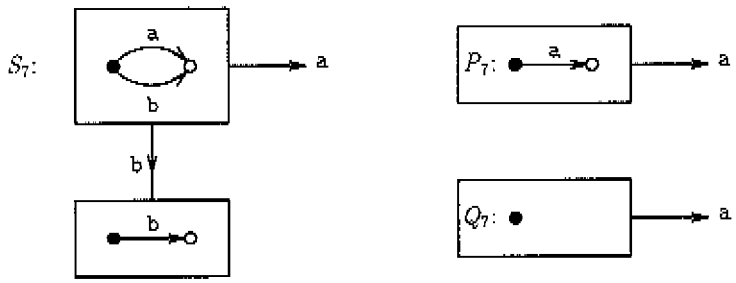


Figure 3.7: Diagrams with state graphs for S_7, P_7, and Q_7

It has been our implicit attitude so far that processes have no obligation to produce output when they are capable of doing so. Consequently, S_7 is equivalent to P_7 and not equivalent to Q_7. We could postulate that, in an environment supplying no input, a process capable of producing output will eventually do so; further input, however, might of might not remove the obligation of the process to produce output. In that case, S_7 can no longer

be considered equivalent to P_7, because P_7 then is a process that is guaranteed to produce a a-signal, whereas S_7 might produce an a-signal or might fail to produce it (namely if internally the choice for b was made). S_7 then is equivalent to a "mixture" of P_7 and Q_7. An extension of the model in Chapter 6 will provide a better way to deal with this (see Example 6.2.1). ◢

3.3 Questions to Be Addressed

The examples above raise such questions as:

- When are two systems equivalent?
- What correctness concerns should we consider?
- When is one system a satisfactory substitute for another?
- Can we define a composition operator that takes a pair of processes and yields a process that is equivalent to the pair?
- In case such a composition operator cannot be defined, how should we extend the space of processes so that composition can be defined?
- Does a finite set of building blocks suffice to implement all interesting systems?

Chapter 4

DI Model

In this chapter we present and analyze a model, called the **DI Model**, that formalizes the concepts introduced in the preceding chapter. We start by defining processes and systems. System structure and operation are treated separately. Next we define a pre-order on systems expressing when one system is "at least as good" as another. The composition and comparison of systems are the key concepts of the DI Model. Subsequent sections analyze these concepts to get a better understanding of the DI Model. For that purpose we present a partial order on processes, composites and canonical representatives, DI processes, and the JTU-Rules. The final two sections deal with the computation of composites and the solution of the design equation.

It is not our aim to cover all mathematical details of the DI Model. The Extended DI Model of Chapter 6 is treated in more detail; especially Chapter 7 delves into the mathematics of these models. In the current chapter, most results are stated without proof. In some cases the proofs are simple, but more often they are rather tedious or even complicated. Some of the proofs have appeared in [UV88, CUV89a, Ver89]. We also present some proofs in Appendix B.

4.1 Processes

Let Σ be an infinite set of **symbols**. Typically, variables a, b, and c range over Σ, and symbols a, b, and c are (distinct) constants in Σ. A finite subset of Σ is called an **alphabet**.

A **process** P is a triple $(\mathbf{i}P, \mathbf{o}P, \mathbf{t}P)$ such that

1. $\mathbf{i}P$ and $\mathbf{o}P$ are disjoint alphabets: $\mathbf{i}P \cap \mathbf{o}P = \emptyset$,
2. $\mathbf{t}P \subseteq (\mathbf{i}P \cup \mathbf{o}P)^*$,
3. $\mathbf{t}P$ is non-empty and prefix-closed.

We call $\mathbf{i}P$ the **input alphabet**, $\mathbf{o}P$ the **output alphabet**, and $\mathbf{t}P$ the **trace set** of P. We define the **alphabet** $\mathbf{a}P$ of P by $\mathbf{a}P = \mathbf{i}P \cup \mathbf{o}P$. The set of all processes is denoted by $\mathcal{PROC}$. Typically, variables P, Q, and R range over $\mathcal{PROC}$.

The input and output alphabets specify the structural properties of a process. Its trace set specifies the behavioral properties. The current "state" of process P is characterized by a trace $t \in \mathbf{t}P$. The intention of the trace set is as follows.

- For $a \in \mathbf{o}P$, we have $ta \in \mathbf{t}P$ if and only if P can send an output signal via a in state t.
- For $a \in \mathbf{i}P$, we have $ta \in \mathbf{t}P$ if and only if P can receive an input signal via a in state t.

This intention will become clearer when we define the operation of a system below. The trace set prescribes restrictions (for proper operation) on the process itself *and* on its environment. It can be viewed as a combination of pre- and post-conditions.

The **reflection** operator $\backsim$ is defined on $\mathcal{PROC}$ by

$$\backsim P \quad = \quad (\mathbf{o}P, \mathbf{i}P, \mathbf{t}P) \; . \tag{4.1}$$

Thus, reflection interchanges input and output alphabet; the trace set is not affected. Reflection is its own inverse. The **empty process** $(\varnothing, \varnothing, \{\varepsilon\})$ is the only process equal to its own reflection.

For trace set V and trace t, trace set V/t (pronounced as 'V **after** t') is defined by

$$V/t \quad = \quad \{u : tu \in V : u\} \; . \tag{4.2}$$

This operator is well-known from the theory of automata and formal languages. The after-operator is lifted to processes as follows. For process P and trace $t \in \mathbf{t}P$, process P/t is defined by

$$P/t \quad = \quad (\mathbf{i}P, \mathbf{o}P, \mathbf{t}P/t) \; . \tag{4.3}$$

Thus, 'aftering' preserves input and output alphabet, and affects only the trace set (which is indeed again non-empty and prefix-closed on account of $t \in \mathbf{t}P$; for $t \notin V$ we have $V/t = \varnothing$). Observe that $P/\varepsilon = P$ and $P/t/u = P/tu$ provided that $tu \in \mathbf{t}P$. Reflection and 'aftering' commute, in the sense that for trace $t \in \mathbf{t}P$, we have

$$\backsim(P/t) \quad = \quad (\backsim P)/t \; . \tag{4.4}$$

Hence, we may omit the parentheses.

Other examples of processes are the ones given in Chapter 3: wire $\mathtt{W}(a;b)$, I-wire $\mathtt{I}(a;b)$, fork $\mathtt{F}(a;b,c)$, merge $\mathtt{M}(a,b;c)$, and C-element $\mathtt{C}(a,b;c)$. We have such identities as

$$\begin{aligned}
\mathtt{I}(a;b) \quad &= \quad \backsim \mathtt{W}(b;a) \; , \\
\mathtt{W}(a;b)/a \quad &= \quad \mathtt{I}(a;b) \; , \\
\mathtt{I}(a;b)/b \quad &= \quad \mathtt{W}(a;b) \; , \\
\mathtt{C}(a,b;c) \quad &= \quad \backsim \mathtt{F}(c;a,b)/c \; .
\end{aligned}$$

For a given process, different traces may correspond to equivalent "states", in the sense that the future looks the same after these traces. Formally, traces t and u of process P are

equivalent in this sense when $P/t = P/u$. For instance, ε and ab are equivalent traces of wire $\mathtt{W}(a;b)$. The processes P/t where t ranges over $\mathbf{t}P$ may be viewed as the (abstract) states of P, and $P = P/\varepsilon$ as the initial state. Occurrence of signal a takes P from state P/t to P/ta provided that $ta \in \mathbf{t}P$. Accordingly, the **minimal state graph** of process P is defined as the edge-labeled directed graph

$$(\{t : t \in \mathbf{t}P : P/t\},\ \{t, a : ta \in \mathbf{t}P : (P/t, a, P/ta)\}) , \tag{4.5}$$

with P/ε as initial state. It is called the minimal state graph because there exist no state graphs for P with fewer vertices. It is unique up to graph isomorphism. The state graphs that we have given in Chapter 3 are all minimal.

4.2 Structure of Systems

A **system** is a finite set, say S, of processes such that for all $a \in \Sigma$ we have

1. $\#\{P : P \in S \wedge a \in \mathbf{i}P : P\} \leqslant 1$ and
2. $\#\{P : P \in S \wedge a \in \mathbf{o}P : P\} \leqslant 1$,

where $\# V$ is the size of set V. The set of all systems is denoted by $\mathcal{SYS}$. Typically, variables S, T, and U range over $\mathcal{SYS}$.

For system S we define a number of alphabets as follows:

$$\begin{array}{lcll}
\mathbf{i}S & = & \bigcup\{P : P \in S : \mathbf{i}P\} & (\textbf{input alphabet}), \\
\mathbf{o}S & = & \bigcup\{P : P \in S : \mathbf{o}P\} & (\textbf{output alphabet}), \\
\mathbf{a}S & = & \mathbf{i}S \cup \mathbf{o}S & (\textbf{alphabet}), \\
\mathbf{n}S & = & \mathbf{i}S \cap \mathbf{o}S & (\textbf{internal alphabet}), \\
\mathbf{x}S & = & \mathbf{i}S \div \mathbf{o}S & (\textbf{external alphabet}), \\
\mathbf{xi}S & = & \mathbf{i}S \setminus \mathbf{o}S & (\textbf{external input alphabet}), \\
\mathbf{xo}S & = & \mathbf{o}S \setminus \mathbf{i}S & (\textbf{external output alphabet}),
\end{array}$$

where $\bigcup V$ denotes the union of the elements of set V[1], and binary operators $\div$ and $\setminus$ on sets denote the symmetric and asymmetric set difference respectively. System S is called **closed** when $\mathbf{x}S = \varnothing$.

Consider processes P and Q in system S. If symbol a occurs in $\mathbf{o}P \cap \mathbf{i}Q$ then the a-port of P drives the a-port of Q via an implicit internal wire. If symbol a occurs in $\mathbf{o}P \setminus \mathbf{i}S$ then the a-port of P is an external output port of S. If symbol a occurs in $\mathbf{i}Q \setminus \mathbf{o}S$ then the a-port of Q is an external input port of S.

4.2.1 Example Trivial examples of systems are $\varnothing$ (without processes and closed), and for process P also $\{P\}$ and $\{P, \backsim P\}$. The latter is a closed system. Note that for $P = \backsim P$, system $\{P, \backsim P\}$ consists of only *one* (empty) process.

[1] The elements of V are themselves also sets.

System S defined by

$$S \;=\; \{\ (\{\mathtt{a},\mathtt{b}\},\{\mathtt{c}\},\{\varepsilon\}),\ \ (\{\mathtt{c}\},\{\mathtt{d},\mathtt{e}\},\{\varepsilon\})\ \} \tag{4.6}$$

has two external input ports a and b, an internal wire c, and two external output ports d and e. ∎

To express that one output drives two (or more) inputs, one needs to introduce explicit forking processes. Similarly, to express that two (or more) outputs drive a single input, one needs to introduce, for instance, explicit merging processes.

The internal symbols of a system are considered dummies. We call two systems **isomorphic** whenever they can be transformed into each other by systematically renaming *internal* symbols. For instance, system T defined by

$$T \;=\; \{\ (\{\mathtt{a},\mathtt{b}\},\{\mathtt{x}\},\{\varepsilon\}),\ \ (\{\mathtt{x}\},\{\mathtt{d},\mathtt{e}\},\{\varepsilon\})\ \} \tag{4.7}$$

is isomorphic to system S defined in (4.6), because T can be obtained from S by renaming internal symbol c to x. Being isomorphic is an equivalence relation. From now on we abstract from this equivalence, that is, two isomorphic systems will be treated as equal. Nevertheless, we will continue to work with representatives of the equivalence classes.

Systems S and T are called **connectable** whenever

$$\mathbf{xi}S \cap \mathbf{xi}T = \varnothing \quad \text{and} \quad \mathbf{xo}S \cap \mathbf{xo}T = \varnothing\ . \tag{4.8}$$

The internal symbols of connectable systems S and T can be renamed systematically, yielding systems S' and T' respectively, such that

$$\mathbf{a}S' \cap \mathbf{n}T' = \varnothing \quad \text{and} \quad \mathbf{n}S' \cap \mathbf{a}T' = \varnothing\ .$$

In that case, $S' \cup T'$ is again a system and it is independent of the particular renamings involved. It is called the **composition** of S and T and is denoted by $S\ par\ T$. Thus, *par* is a partial binary operator on $\mathcal{SYS}$ defined only for connectable systems.

Composition of two systems introduces an internal wire for each symbol that is an external output of one system and an external input of the other. Composition is commutative and has the empty system $\varnothing$ as unit. It is associative *provided* that no symbol occurs in the external alphabets of more than two of the composed systems. The next example shows what can go wrong when a symbol occurs in more than two alphabets. The situation is similar to that of the blending operator in [vdS85].

4.2.2 Example Consider processes P, Q, and R defined by

$$\begin{aligned} P &- (\{\mathtt{a}\},\varnothing,\{\varepsilon,\mathtt{a}\})\ , \\ Q &- (\varnothing,\{\mathtt{a}\},\{\varepsilon,\mathtt{a}\})\ , \\ R &= (\{\mathtt{a}\},\varnothing,\{\varepsilon\})\ . \end{aligned}$$

Process P has one input port a and is willing to receive a signal. Process Q has one output port a and can send a signal ($Q = \backsim P$). Process R has one input port a but cannot process

a signal. Note that symbol a occurs in all three processes. On the one hand, in system S_1 defined by

$$S_1 \;=\; (\{P\}\ par\ \{Q\})\ par\ \{R\}\ ,$$

an internal wire is introduced between P and Q (it has to be renamed from a to something else, say x, for composition with $\{R\}$, see Figure 4.1); furthermore, a is an external port

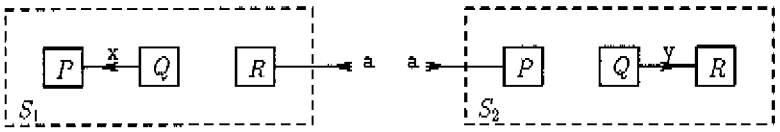


Figure 4.1: Diagrams for S_1 (left) and S_2 (right)

of S_1 connected to R. On the other hand, in system S_2 defined by

$$S_2 \;=\; \{P\}\ par\ (\{Q\}\ par\ \{R\})\ ,$$

a is an external port connected to P, whereas an internal wire, say y, is introduced between Q and R (see Figure 4.1). Systems S_1 and S_2 are non-isomorphic (since they cannot be transformed into each other by renaming internals). It will turn out that they are not equivalent in a broader sense either (see Section 4.4). ∎

Apparently, the internal connection pattern can vary with the order of composition. Note that in the example above, $\{P, Q, R\}$ is not a system and $\{P\}\ par\ \{R\}$ is not defined because P and R are not connectable due to an input conflict on a. It is straightforward to prove that

'$(S\ par\ T)\ par\ U$ is defined and closed'

≡

'$S\ par\ (T\ par\ U)$ is defined and closed' .

Furthermore, if $(S\ par\ T)\ par\ U$ is closed, then no symbol occurs in more than two external alphabets.

4.2.3 Note Let S and T be two connectable systems. Intuitively one would expect the number of processes in $S\ par\ T$ to equal the sum of the numbers of processes in S and T. This is, in general, the case; the only exception occurs when the empty process $(\varnothing, \varnothing, \{\varepsilon\})$ is a member of both S and T. In that case, one copy of the empty process vanishes under composition. This does not invalidate the model, since the empty process is "harmless" anyway. One way to overcome this flaw is to define systems as bags of processes instead of sets (see [Ver94]). We have not used bags because of the additional burden of using a bag calculus. ∎

4.3 Operation of Systems

In this section we consider how systems operate. The operation of a system involves the interaction of its processes and interconnecting wires. First we define system operation under the assumption that the interconnecting wires have no delay, that is, the sending of a transition onto a wire coincides with its reception at the other end. This is called **isochronic operation** (for lack of a better name). Then we define system operation assuming that wires may incur delays; this we call **anisochronic operation**.

The isochronic operation of system S is characterized by its reachable traces. The set *reach*.S of **reachable traces** of S is defined inductively as the $\subseteq$-least trace set satisfying

- $\varepsilon \in$ *reach*.S and
- if $t \in$ *reach*.S, $P \in S$, $a \in \mathbf{o}P$, and $ta\lceil\mathbf{a}P \in \mathbf{t}P$ then $ta \in$ *reach*.S.

Here, $ta\lceil\mathbf{a}P$ (pronounced as 'ta **projected on** $\mathbf{a}P$') denotes the trace obtained from ta by removing all symbols not in $\mathbf{a}P$ (see [vdS85]). Note that *reach*.$S \subseteq (\mathbf{a}S)^*$.

The empty trace ε models the "initial state" of system S. The (global) state ta of S induces the (local) state $ta\lceil\mathbf{a}P$ at process P. State changes can occur whenever a process can produce output in its current (local) state. If for process $P \in S$, symbol $a \in \mathbf{o}P$, and trace $t \in$ *reach*.S, we have $ta\lceil\mathbf{a}P \in \mathbf{t}P$, then we say that '*output* a **is enabled** in P after t'. If for process $Q \in S$, symbol $a \in \mathbf{i}Q$, and trace $t \in$ *reach*.S, we have $ta\lceil\mathbf{a}Q \in \mathbf{t}Q$, then we say that '*input* a **is acceptable** for Q after t'. When an enabled output is actually produced, it appears as input at the receiving end, *regardless* of whether that input is acceptable for the receiver in the current state.

Not all reachable traces are regarded as equally desirable. Reachable trace t is called **interfering** when there exists a process $P \in S$ with $t\lceil\mathbf{a}P \notin \mathbf{t}P$, that is, when the local state that t induces at P is *not* in agreement with P's specification. The set of interfering traces of S is denoted by *intf*.S. Note that $\varepsilon \notin$ *intf*.S, since $\varepsilon\lceil\mathbf{a}P = \varepsilon \in \mathbf{t}P$ for every process P.

System S is said to be **free of interference** when

$$\mathit{intf}.S \;=\; \varnothing\ , \tag{4.9}$$

that is, when it has no interfering traces.

4.3.1 Example Consider process P and system S defined by

$$P \;=\; (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{ab}\})\ ,$$
$$S \;=\; \{P, \backsim P\}\ .$$

Process P has no input ports and two output ports a and b. It can send a signal along a followed by a signal along b. Process $\backsim P$ is willing to accept two inputs, first along a then along b. The reachable traces of S are now given by

$$\mathit{reach}.S \;=\; \{\varepsilon, \mathtt{a}, \mathtt{ab}\}\ .$$

None of these traces is interfering and, hence, S is free of interference. ∎

4.3.2 Example In case system S is given by

$$S \;=\; \{\; \mathtt{I(a;b)},\;\; \mathtt{I(b;a)} \;\}\;,$$

we have

$$\mathit{reach}.S \;=\; \{\varepsilon, \mathtt{a}, \mathtt{b}\}\;.$$

The only non-interfering trace is ε, because traces starting with a are interfering on account of $\mathtt{I(a;b)}$ and those starting with b on account of $\mathtt{I(b;a)}$. Either process can initially send an output but not accept an input. Once an output has been produced there is interference and thereafter neither process can produce further output. Thus, S is *not* free of interference. ∎

In order to define system operation under the assumption that connecting wires may incur delays, we add explicit wire processes to model the behavior of the (implicit) internal wires. For that purpose, the symbols in S must first be renamed. Given process P, define renaming ρ_P of $\mathbf{a}P$ by

$$\rho_P.a \;=\; \begin{cases} a? & \text{if} \quad a \in \mathbf{i}P \\ a! & \text{if} \quad a \in \mathbf{o}P \end{cases} \tag{4.10}$$

Such a renaming is lifted to processes: process $\rho_P.P$ is obtained from P by replacing each symbol a in P, (that is, in the alphabets and in the traces) by $\rho_P.a$. We now define system $\vec{S}$, called the **wired** version of S, by

$$\begin{aligned} \vec{S} \;=\; & \{P : P \in S : \rho_P.P\} \cup \{a : a \in \mathbf{n}S : \mathtt{W}(a!;a?)\} \cup \\ & \{a : a \in \mathbf{xi}S : \mathtt{W}(a;a?)\} \cup \{a : a \in \mathbf{xo}S : \mathtt{W}(a!;a)\}\;. \end{aligned} \tag{4.11}$$

The definition of $\vec{S}$ is a little more general than required at this point: there are wires to and from external ports as well. This is useful later on in Theorem 4.4.2. If S is closed then $\vec{S}$ is closed as well, and the last two sets of additional wires in the definition of $\vec{S}$ are empty. Note that symbol $a!$ is the *output* port of some renamed "ordinary" process, connected to the *input* port of wire $\mathtt{W}(a!;a?)$ or $\mathtt{W}(a!;a)$. Isochronic operation of $\vec{S}$ will be referred to as **anisochronic operation** of S.

4.3.3 Example Consider again process P and system S from Example 4.3.1. System $\vec{S}$ in this case amounts to

$$\{\;\; (\varnothing, \{\mathtt{a!}, \mathtt{b!}\}, \{\varepsilon, \mathtt{a!}, \mathtt{a!b!}\}),\;\; (\{\mathtt{a?}, \mathtt{b?}\}, \varnothing, \{\varepsilon, \mathtt{a?}, \mathtt{a?b?}\}),\;\; \mathtt{W(a!;a?)},\;\; \mathtt{W(b!,b?)} \;\}\;.$$

Note that S is closed and that both symbols a and b are internal, giving rise to two additional wire processes in $\vec{S}$. The reachable traces of $\vec{S}$ are readily computed as

$$\begin{aligned} \mathit{reach}.\vec{S} \;=\; \{\;\; & \varepsilon,\;\; \mathtt{a!},\;\; \mathtt{a!b!},\;\; \mathtt{a!b!a?},\;\; \mathtt{a!b!a?b?}, \\ & \qquad\qquad\quad \mathtt{a!b!b?},\;\; \mathtt{a!b!b?a?}, \\ & \qquad\quad \mathtt{a!a?},\;\; \mathtt{a!a?b!},\;\; \mathtt{a!a?b!b?} \;\;\}\;. \end{aligned}$$

Thus a!a?b!b? is in $\mathit{reach}.\vec{S}$ and it is not interfering. However, a!b!b? is reachable and interfering. Hence, $\vec{S}$ is not free of interference. ∎

The example above gives a system S that is free of interference, whereas $\vec{S}$ is not. The reason for the difference is that under anisochronic operation the additional wires in $\vec{S}$ need not preserve the order in which signals are sent. One thing that complicates the design of delay-insensitive systems is precisely that no assumptions are to be made about wire delays (other than that they are not negative).

4.3.4 Example For system S of Example 4.3.2 we have

$$\vec{S} \;=\; \{\; \mathtt{I(a?;b!)},\;\; \mathtt{I(b?;a!)},\;\; \mathtt{W(a!;a?)},\;\; \mathtt{W(b!;b?)} \;\}\;.$$

Each trace of the form $(\mathtt{b!a!b?a?})^n$ (for any $n \geqslant 0$) is a non-interfering trace of $\vec{S}$. Furthermore, these traces put the system in a state equivalent to the initial state (in the sense that the future possibilities are the same). However, traces of the form $(\mathtt{b!a!b?a?})^n\mathtt{b!b?}$ are also traces of $\vec{S}$, but they are interfering. Thus, $\vec{S}$ has both an infinite number of interfering and non-interfering traces ($\vec{S}$ is obviously not free of interference). Recall that ε is the only non-interfering trace of S. ∎

4.3.5 Example Consider process $P = (\varnothing, \{\mathtt{a}\}, \{\varepsilon, \mathtt{a}, \mathtt{aa}\})$ and system $S = \{P, \backsim P\}$. Process P has one output port a on which it sends two signals in succession. Process $\backsim P$ can accept two inputs on a. Therefore, system S is free of interference; but $\vec{S}$ is not, since P causes interference at the additional wire $\mathtt{W(a!;a?)}$ in $\vec{S}$. ∎

This example shows another system S that is free of interference, but for which $\vec{S}$ is not. The reason now is not that wires may disturb signal order, but that a wire can safely transmit only one signal at a time. Interference caused at a wire input is called **transmission interference**.

We would like to make some remarks on our approach to system behavior.

4.3.6 Note First of all, our (operational) semantics is based on **interleaving** of concurrent atomic events. That is, events that are not "causally" related are put in some (arbitrary) order in the execution sequence. In this model the only effect of variation in *delays* (within the modeled processes) can be variation in *order* (of symbols in traces).

Secondly, we have not taken the trouble to model the behavior of a system accurately after the occurrence of interference. The reason is that interference is to be avoided according to the correctness criterion introduced in the next section. Once there is interference, we do not care what happens afterwards; the game has been lost anyway.

Thirdly, the definitions of *reach* and *intf* can also be applied to systems that are not closed. Consider, for example, process P defined by

$$P \;=\; (\{\mathtt{a}\}, \{\mathtt{b}\}, \{\varepsilon, \mathtt{b}, \mathtt{ba}\})\;.$$

System $\{P\}$ is not closed and $reach.\{P\}$ equals $\{\varepsilon, \mathtt{b}\}$, because there is no process to provide input a. Note that $reach.\{P\} \neq \mathtt{t}P$. Furthermore, $intf.\{P\}$ is empty: there is no process that fails to accept output b. Thus, $\{P\}$ is free of interference. In a sense the definitions of *reach* and *intf* applied to a system that is not closed assume some very

benign, but unrealistic, environment that accepts all outputs and sends no inputs. We are not interested in this interpretation. The next section will deal with systems that are not closed in a different way. ◢

We finish this section with two theorems, whose proofs illustrate how tedious the details of our model can be. Theorem 4.3.7 gives an alternative characterization of interference based on weaving. Theorem 4.3.8 relates interference of S and $\vec{S}$. The weaving operator *weave* (see [vdS85]) may be defined for systems by

$$weave.S \;=\; \{t : t \in (\mathbf{a}S)^* \wedge (\forall P : P \in S : t\lceil \mathbf{a}P \in \mathbf{t}P) : t\} \;. \tag{4.12}$$

For process P, we have $weave.\{P\} = \mathbf{t}P$. For closed system S, we have

$$weave.S \;\subseteq\; reach.S \;, \tag{4.13}$$

since in a closed system each symbol is an output of some process. Statement (4.13) does not necessarily hold when S is not closed, as witnessed by system $\{P\}$ of Note 4.3.6. Thus, closed system S is free of interference if and only if

$$reach.S \;=\; weave.S \;. \tag{4.14}$$

The next theorem characterizes interference in terms of $weave.S$ rather than the inductively defined set $reach.S$.

4.3.7 Theorem Closed system S is free of interference if and only if

$$(\forall t, a, P : t \in weave.S \wedge P \in S \wedge a \in \mathbf{o}P \wedge ta\lceil \mathbf{a}P \in \mathbf{t}P : ta \in weave.S) \;. \tag{4.15}$$

Proof: See Appendix B.0.1. ◢

Note that the universal quantification (4.15) is also equivalent to

$$(\forall t, a, P, Q : t \in weave.S \wedge P \in S \wedge Q \in S \wedge a \in \mathbf{o}P \cap \mathbf{i}Q \wedge ta\lceil \mathbf{a}P \in \mathbf{t}P \\ : ta\lceil \mathbf{a}Q \in \mathbf{t}Q) \;,$$

since each output is an input to some process and $ta\lceil A = t\lceil A$ whenever $a \notin A$. When system S is not free of interference, this can be shown by exhibiting processes P and Q in S, symbol a in $\mathbf{o}P \cap \mathbf{i}Q$, and a trace $t \in weave.S$, such that output a is enabled in P after t and input a is not acceptable for Q after t. This can be phrased concisely as 'P **causes interference** at Q on port a after trace t'.

4.3.8 Theorem For closed system S we have

'$\vec{S}$ is free of interference' $\;\Rightarrow\;$ 'S is free of interference' .

Proof idea: When wire delay plays a role, that delay may be zero as well, which corresponds to isochronic operation. Hence, any interfering trace $t \in reach.S$ corresponds to some interfering trace $t' \in reach.\vec{S}$ (replace each occurrence of symbol a in t by a!a? to obtain t'). ◣ ◢

In general, the reverse implication does not hold as illustrated by Examples 4.3.1, 4.3.3, and 4.3.5. Theorem 4.7.7, however, states a condition under which the reverse implication does hold.

4.4 Correctness, Satisfaction, and Equivalence

We consider only the operation of closed systems, that is, of systems whose ports are all properly connected. We disallow dangling inputs and outputs because these may pick up or radiate stray signals, or may cause other types of malfunctioning. If an input is to be "kept quiet" then one should express that by hooking it up to a "quiet" process. Furthermore, we insist on absence of interference during operation.

This is captured in the following definition of our correctness concern. System S is **correct** (as an autonomous system, that is, requiring no additional environment for its operation), denoted by $Correct.S$, when $\vec{S}$ is closed and free of interference:

$$Correct.S \quad \equiv \quad \text{`}\vec{S} \text{ is closed and free of interference'} \; .$$

Because it is based on $\vec{S}$, this definition involves anisochronic operation of S, that is, with additional wires. Thus, correctness requires absence of interference for all possible delays. Isochronic operation plays a role again in Theorem 4.7.7.

We say that system S is a **satisfactory replacement** for system T whenever T (being part of any larger system) can be replaced by S without disturbing the correctness (of the larger system). We denote this by $S \; sat \; T$. Note that if T is part of some larger system, then the larger system can be written as $T \; par \; U$ for some system U. Formally, relation sat on $\mathcal{SYS}$ is defined by

$$S \; sat \; T \quad \equiv \quad (\forall \, U \; : \; U \in \mathcal{SYS} \; : \; Correct.(S \; par \; U) \; \Leftarrow \; Correct.(T \; par \; U)) \; . \qquad (4.16)$$

We postulate that $Correct.(S \; par \; U)$ does not hold if $S \; par \; U$ is not defined.

We can interpret $S \; sat \; T$ also as 'system S satisfies specification T'. Consequently, sat is also called a **satisfaction** or **refinement** relation. Relation sat is a pre-order, that is, sat is reflexive and transitive (but not necessarily antisymmetric). Hence, **equivalence** of systems, denoted by equ, can be defined by

$$S \; equ \; T \quad \equiv \quad S \; sat \; T \; \wedge \; T \; sat \; S \; , \qquad (4.17)$$

that is, S and T are equivalent when they are satisfactory replacements for each other. Relation equ is an equivalence relation on $\mathcal{SYS}$ (it is reflexive, transitive, and symmetric). Of course, we have

$$S \; equ \; T \quad \equiv \quad (\forall \, U \; : \; U \in \mathcal{SYS} \; : \; Correct.(S \; par \; U) \; \equiv \; Correct.(T \; par \; U)) \; . \qquad (4.18)$$

Furthermore, equ is a **congruence** with respect to par and sat, that is, composition and satisfaction "do not cross equ-class boundaries". Formally, this is expressed as follows. For systems S, S', T, and T' with $S \; equ \; S'$ and $T \; equ \; T'$ we have

$$S \; par \; T \quad equ \quad S' \; par \; T' \quad \text{and}$$
$$S \; sat \; T \quad \equiv \quad S' \; sat \; T' \; .$$

[For a proof of the first equivalence see Appendix B.] Consequently, we can abstract from equ-equivalence in the algebra $\langle \mathcal{SYS}; par, sat \rangle$ and obtain the so-called **quotient algebra** $\langle \mathcal{SYS}; par, sat \rangle / equ$.

Verifying that S *sat* T, or S *equ* T, holds according to the definition is a cumbersome task, because of the quantification over all systems in (4.16). Proving that S *sat* T does *not* hold can be done by exhibiting a suitable system U such that T *par* U is correct but S *par* U is not correct.

4.4.1 Example Reconsider systems S_1 and S_2 introduced in Example 4.2.2. We claim

$$S_1 \textit{ sat } S_2 \quad \wedge \quad \neg(S_2 \textit{ sat } S_1) \ ,$$

and, hence, S_1 is not equivalent to S_2.

To prove the second conjunct, consider system $U = \{\backsim R\}$. System U has one output port a and it will not send a signal. Observe that both systems S_1 *par* U and S_2 *par* U are well-defined and closed. However, S_1 *par* U is correct ($(S_1 \textit{ par } U)^{\neg}$ is free of interference), but S_2 *par* U is not (since in $(S_2 \textit{ par } U)^{\neg}$, process $\rho_Q.Q$ causes interference at $\rho_R.R$).

The first conjunct is, in this particular case, not difficult to prove, because for any system U such that S_2 *par* U is well-defined and closed, S_2 *par* U is not free of interference (as indicated above). Hence, for all systems U we have $\neg \mathit{Correct}.(S_2 \textit{ par } U)$, and this trivially yields S *sat* S_2 for any system S. In a sense, S_2 is the worst imaginable system: no environment can keep it from running into interference. ◢

4.4.2 Theorem For any system S we have

$$S \quad \mathit{equ} \quad \vec{\vec{S}} \ .$$

Proof idea: By operational reasoning, the additional wires in $\vec{\vec{S}}$ may be coalesced with the wires that are introduced for anisochronic operation of $\vec{\vec{S}}$, that is, when considering isochronic operation of $(\vec{S} \textit{ par } U)^{\neg}$. ◢

4.4.3 Note It is possible to define *Correct*, *sat*, and *equ* for *isochronic* operation as well:

$$\begin{aligned}
\mathit{Correct}^{\mathit{iso}}.S &\equiv \text{`}S \text{ is closed and free of interference'} \ , \\
S \ \mathit{sat}^{\mathit{iso}} \ T &\equiv (\forall\, U : U \in \mathcal{SYS} : \mathit{Correct}^{\mathit{iso}}.(S \textit{ par } U) \Leftarrow \mathit{Correct}^{\mathit{iso}}.(T \textit{ par } U)) \ , \\
S \ \mathit{equ}^{\mathit{iso}} \ T &\equiv (\forall\, U : U \in \mathcal{SYS} : \mathit{Correct}^{\mathit{iso}}.(S \textit{ par } U) \equiv \mathit{Correct}^{\mathit{iso}}.(T \textit{ par } U)) \ .
\end{aligned}$$

The proof of the preceding theorem, can then be based on repeated application of the following equivalence for distinct symbols a, b, and c:

$$\{\ \mathtt{W}(a;b),\ \mathtt{W}(b;c)\ \} \quad \mathit{equ}^{\mathit{iso}} \quad \{\ \mathtt{W}(a;c)\ \} \ . \tag{4.19}$$

◢

4.5 Partial Order on Processes

In the next section, we will pick a canonical representative from each *equa*-equivalence class of systems. To prepare the road, this section focuses on processes. Relation $\sqsubseteq$ on $\mathcal{PROC}$ is defined by

$$\begin{aligned} P \sqsubseteq Q \;\; = \;\; & \mathbf{i}P = \mathbf{i}Q \wedge \mathbf{o}P = \mathbf{o}Q \wedge \\ & (\forall\, t, a : a \in \mathbf{i}P \wedge ta \in \mathbf{t}P \wedge t \in \mathbf{t}Q : ta \in \mathbf{t}Q) \wedge \\ & (\forall\, t, a : a \in \mathbf{o}P \wedge t \in \mathbf{t}P \wedge ta \in \mathbf{t}Q : ta \in \mathbf{t}P) \end{aligned} \tag{4.20}$$

In words, the requirements for $P \sqsubseteq Q$ can be expressed as follows.

1. P and Q have the same input alphabets and the same output alphabets.
2. For all states $t \in \mathbf{t}P \cap \mathbf{t}Q$, the set of inputs that P can receive in state t is included in the set of inputs that Q can receive in state t ("P can receive no more inputs than Q").
3. For all states $t \in \mathbf{t}P \cap \mathbf{t}Q$, the set of outputs that P can send in state t contains the set of outputs that Q can send in state t ("P can send at least the outputs that Q can send").

$P \sqsubseteq Q$ expresses that Q is "at least as good as" P with respect to interference under isochronic operation.

Relation $\sqsubseteq$ is a partial order on $\mathcal{PROC}$, that is, it is reflexive, antisymmetric, and transitive (this is a non-trivial result; see Appendix B for a proof).

The $\sqsubseteq$-minimal elements of $\mathcal{PROC}$ are processes P with

$$\mathbf{t}P \;\; = \;\; (\mathbf{o}P)^* \;,$$

and the $\sqsubseteq$-maximal elements are processes P with

$$\mathbf{t}P \;\; = \;\; (\mathbf{i}P)^* \;.$$

Reflection turns the order around:

$$P \sqsubseteq Q \;\; \equiv \;\; \backsim P \sqsupseteq \backsim Q \;. \tag{4.21}$$

Furthermore, $\sqsubseteq$ induces a complete lattice structure in each set of processes with the same input-output alphabets. To express this more precisely we define $\mathcal{PROC}(I, O)$ for alphabets I and O by

$$\mathcal{PROC}(I, O) \;\; = \;\; \{P : P \in \mathcal{PROC} \wedge \mathbf{i}P = I \wedge \mathbf{o}P = O : P\} \;.$$

Poset $\langle \mathcal{PROC}(I, O); \sqsubseteq \rangle$ is a complete lattice, in the sense that every subset V has a greatest lower bound (denoted by $\sqcap\, V$) and a least upper bound (denoted by $\sqcup\, V$). See [UV88] for proofs.

Note that processes in distinct $\mathcal{PROC}(I, O)$ are incomparable under $\sqsubseteq$ and do not have common lower or upper bounds. Hence, $\langle \mathcal{PROC}; \sqsubseteq \rangle$ is not a lattice. Therefore, we introduce $\bot$ (pronounced 'bottom') and its reflection $\top$ (pronounced 'top') as additional processes, where $\bot$ is the $\sqsubseteq$-least process and $\top$ the $\sqsubseteq$-greatest. The expanded set of processes is again denoted by $\mathcal{PROC}$. Consequently, $\langle \mathcal{PROC}; \sqsubseteq \rangle$ now is a complete lattice.

Processes $\bot$ and $\top$ have no input and output alphabets and no trace set, and we consider them to be members of all $\mathcal{PROC}(I, O)$. For connection purposes (that is, under composition by *par*), however, they should be regarded as having empty input-output alphabets. The rules for system correctness become slightly more complicated and are postulated in the next paragraph.

A system that contains process $\top$ is correct no matter what else it contains (even $\bot$). Process $\top$ acts as a miraculous panacea. A system that does not contain process $\top$ but that does contain $\bot$ is incorrect, no matter what else it contains. Process $\bot$ spoils everything, except when $\top$ is present. At this point it is not clear why we choose to let top win over bottom. You may choose otherwise, but then top will turn out to be equivalent to the empty process and later we need a "real top" to define canonical representatives (worst friend of bottom) and to make factorization work in all cases.

Without proof we state two important properties of $\sqsubseteq$.

4.5.1 Theorem Predicate *Correct* is $\sqcap$-continuous (hence, $\sqsubseteq$-monotonic) in the following sense. For system S and process set $W \subseteq \mathcal{PROC}$ we have

$$(\forall\, P : P \in W : \mathit{Correct}.(S \mathit{\ par\ } \{P\})) \quad \equiv \quad \mathit{Correct}.(S \mathit{\ par\ } \{\sqcap W\}) \ .$$

Consequently, *Correct* is also $\sqsubseteq$-monotonic, that is, for system S and processes P and Q with $P \sqsubseteq Q$ we have

$$\mathit{Correct}.(S \mathit{\ par\ } \{P\}) \quad \Rightarrow \quad \mathit{Correct}.(S \mathit{\ par\ } \{Q\}) \ .$$

■

4.5.2 Theorem For processes P and Q we have

$$P \sqsupseteq Q \quad \equiv \quad \text{'system } \{P, \backsim Q\} \text{ is well-defined, closed, and free of interference'} \ .$$

Note that interference, here, involves *isochronic* operation of the system, that is, the implicit internal wires are delayless. ■

4.5.3 Note In the light of Note 4.4.3, the preceding theorem can be rephrased as

$$\mathit{Correct}^{iso}.\{P, Q\} \quad \cong \quad P \sqsupseteq \backsim Q \ . \tag{4.22}$$

Further analysis of isochronic operation would reveal that for processes P and Q we also have

$$P \mathit{\ sat}^{iso}\ Q \quad \equiv \quad P \sqsupseteq Q \ ,$$
$$P \mathit{\ equ}^{iso}\ Q \quad \equiv \quad P = Q \ .$$

■

4.6 Composites and Canonical Representatives

This section deals again with systems. We show how to pick a canonical representative in each equivalence class of systems and how to express *par* and *sat* in terms of these representatives. This forms the basis of the fully abstract model presented in the next section.

For each equivalence class this representative has the form of a special singleton system (consisting of just one process). The class of systems S for which

$$(\forall\, U : U \in \mathcal{SYS} : \neg \mathit{Correct}.(S \mathit{\ par\ } U)) \tag{4.23}$$

is a somewhat special case. No matter what environment these systems are placed in, they do not give rise to a correct system. Examples are $\{\mathtt{I(a;b)}, \mathtt{I(b;c)}\}$ and $\{\mathtt{I(a;b)}, \mathtt{I(b;a)}\}$; the latter is closed, the former not.

That all systems satisfying (4.23) are indeed equivalent becomes even more obvious when we introduce the notion of a system's pass set. Placing system S in an environment U to yield system $S \mathit{\ par\ } U$ can be viewed as a form of **observation** or **testing**. System S passes test U whenever $\mathit{Correct}.(S \mathit{\ par\ } U)$ holds. For system S its **pass set** $\mathit{pass}.S$ is defined by

$$\mathit{pass}.S \;=\; \{\, U : U \in \mathcal{SYS} \wedge \mathit{Correct}.(S \mathit{\ par\ } U) : U \,\}\ , \tag{4.24}$$

that is, $\mathit{pass}.S$ consists of all tests that S passes. Obviously we have

$$\begin{array}{lcl} S \mathit{\ sat\ } T & \equiv & \mathit{pass}.S \supseteq \mathit{pass}.T \quad \text{and} \\ S \mathit{\ equ\ } T & = & \mathit{pass}.S = \mathit{pass}.T\ . \end{array}$$

The exceptional systems mentioned above are characterized by $\mathit{pass}.S = \varnothing$ and, hence, they form an equivalence class on their own. These systems are not very interesting (nevertheless they are present in the model).

We now concentrate on the singleton tests in the pass sets. For system S define the process set $\mathit{Friends}.S$, called the **friends** of S, by

$$\mathit{Friends}.S \;=\; \{P : P \in \mathcal{PROC} \wedge \mathit{Correct}.(S \mathit{\ par\ } \{P\}) : P\}\ . \tag{4.25}$$

Note that $\top \in \mathit{Friends}.S$ for any S. If $\top \in S$ then $\mathit{Friends}.S = \mathcal{PROC}$. If $\top \notin S$ then all friends of S have the same input and output alphabets, in the sense that

$$\mathit{Friends}.S \;\subseteq\; \mathcal{PROC}(I, O)$$

for some disjoint alphabets I and O. On account of the $\sqsubseteq$-monotonicity of correctness (Theorem 4.5.1), $\mathit{Friends}.S$ is $\sqsubseteq$-upward closed: for processes P and Q with $P \sqsubseteq Q$ we have

$$P \in \mathit{Friends}.S \;\Rightarrow\; Q \in \mathit{Friends}.S\ . \tag{4.26}$$

On account of the $\sqcap$-continuity of correctness, $\mathit{Friends}.S$ has a least element (since its greatest lower bound is a friend of S as well). Consequently, $\mathit{Friends}.S$ is completely determined by its least element.

Define $[\![S]\!]$, called the **composite** of S, by

$$[\![S]\!] \;=\; \backsim\!\sqcap \mathit{Friends}.S \;, \tag{4.27}$$

that is, as the reflection of its least friend. Note that if $\mathit{Friends}.S = \{\top\}$ then $[\![S]\!] = \bot$. Thus, $[\![\_]\!]$ is a mapping from $\mathcal{SYS}$ to $\mathcal{PROC}$ (with bottom and top). For distinct processes P and Q we will often write $P \parallel Q$ for $[\![\{P, Q\}]\!]$. Composition $\parallel$ is a partial[2] binary operator on $\mathcal{PROC}$.

The **canonical representative** of the *equ*-class containing system S is defined as singleton system $\{[\![S]\!]\}$. We give three examples to illustrate the concepts.

4.6.1 Example Consider systems S_1 through S_4 defined below. All four have no external inputs and two external outputs $\{\mathtt{a}, \mathtt{b}\}$. System S_1 consists of two processes, while the others are singleton systems.

$$\begin{aligned}
S_1 &= \{\ (\varnothing, \{\mathtt{a}\}, \{\varepsilon, \mathtt{a}\}),\ (\varnothing, \{\mathtt{b}\}, \{\varepsilon, \mathtt{b}\})\ \}\ ,\\
S_2 &= \{\ (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{ab}\})\ \}\ ,\\
S_3 &= \{\ (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{b}, \mathtt{ba}\})\ \}\ ,\\
S_4 &= \{\ (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{ab}, \mathtt{ba}\})\ \}\ .
\end{aligned}$$

We claim that these four non-isomorphic systems are in the same *equ*-class. Here are some friends of all four:

$$\begin{aligned}
P_1 &= \top\ ,\\
P_2 &= (\{\mathtt{a}, \mathtt{b}\}, \varnothing, \{\mathtt{a}, \mathtt{b}\}^*)\ ,\\
P_3 &= (\{\mathtt{a}, \mathtt{b}\}, \varnothing, \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{ab}, \mathtt{ba}\})\ .
\end{aligned}$$

Note that process $P_4 = (\{\mathtt{a}, \mathtt{b}\}, \varnothing, \{\varepsilon, \mathtt{a}, \mathtt{ab}\})$, which is the reflection of the process in S_2, is *not* a friend of either of the four systems, because the (implicit) wires between the system and its environment $\{P_4\}$ may interchange the order of the a- and b-signals, thus giving rise to interference at the test environment when a arrives after b.

The following two statements follow immediately from the definitions:

$$\begin{aligned}
S \mathit{\ equ\ } T &\;\Rightarrow\; \mathit{Friends}.S = \mathit{Friends}.T\ ,\\
S \mathit{\ sat\ } T &\;\Rightarrow\; \mathit{Friends}.S \supseteq \mathit{Friends}.T\ .
\end{aligned}$$

The reverse implications also hold but are not trivial; for all we know, tests with more than one process might play a crucial role in *equ* and *sat*. Theorem 4.6.4 below, however, resolves this issue: only singleton tests are important. Accordingly, to establish the equivalence of the four systems, it suffices to prove that their $\sqsubseteq$-least friends are the same.

[2]Composition $\parallel$ could be extended to a total operator by defining $P \parallel Q = \bot$ whenever systems $\{P\}$ and $\{Q\}$ are not connectable.

We claim that P_3 is this common least friend. Observe that all friends of the systems involved have empty output alphabets. We will use the following, straightforward, property of $\sqsubseteq$ that applies to this case. For alphabet I and processes P and Q in $\mathcal{PROC}(I, \varnothing)$ we have

$$P \sqsubseteq Q \quad \equiv \quad \mathbf{t}P \subseteq \mathbf{t}Q \ . \tag{4.28}$$

All that is left to do is verifying whether any traces can be eliminated from $\mathbf{t}P_3$ while maintaining friendship with the systems. This gives rise to a finite case analysis. It turns out that no traces can be removed without introducing interference. Thus, the canonical representative of the *equ*-class containing the four systems happens to be $\{\backsim P_3\} - S_4$. ◂

4.6.2 Example Now consider systems S_5 through S_8, obtained from S_1 through S_4 above by reflecting all the processes involved:

$$\begin{aligned}
S_5 &= \{\ (\{\mathtt{a}\}, \varnothing, \{\varepsilon, \mathtt{a}\}),\ (\{\mathtt{b}\}, \varnothing, \{\varepsilon, \mathtt{b}\})\ \}\ , \\
S_6 &= \{\ (\{\mathtt{a}, \mathtt{b}\}, \varnothing, \{\varepsilon, \mathtt{a}, \mathtt{ab}\})\ \}\ , \\
S_7 &= \{\ (\{\mathtt{a}, \mathtt{b}\}, \varnothing, \{\varepsilon, \mathtt{b}, \mathtt{ba}\})\ \}\ , \\
S_8 &= \{\ (\{\mathtt{a}, \mathtt{b}\}, \varnothing, \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{ab}, \mathtt{ba}\})\ \}\ .
\end{aligned}$$

These four systems are again non-isomorphic. In this case, however, it will turn out that only S_5 and S_8 belong to the same equivalence class. Here are six *candidates* for friends:

$$\begin{aligned}
P_5 &= (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon\})\ , \\
P_6 &= (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{a}\})\ , \\
P_7 &= (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{b}\})\ , \\
P_8 &= (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{ab}\})\ , \\
P_9 &= (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{b}, \mathtt{ba}\})\ , \\
P_{10} &= (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{ab}, \mathtt{ba}\})\ .
\end{aligned}$$

A little investigation reveals the following friendships:

	P_5	P_6	P_7	P_8	P_9	P_{10}
S_5	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*
S_6	*yes*	*yes*	*no*	*no*	*no*	*no*
S_7	*yes*	*no*	*yes*	*no*	*no*	*no*
S_8	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*

Looking at processes P_6 and P_7, we see that neither S_6 *sat* S_7 nor S_7 *sat* S_6 holds. Furthermore, from the preceding example we already know that P_8, P_9, and P_{10} are equivalent (as singleton systems: $S_2 = \{P_8\}$, $S_3 = \{P_9\}$, and $S_4 = \{P_{10}\}$) and, hence, their membership in some set *Friends*.S comes and goes as a block.

All friends of these systems have empty input alphabets. This situation is covered by a similar property of $\sqsubseteq$ as (4.28) above. For alphabet O and processes P and Q in $\mathcal{PROC}(\varnothing, O)$ we have

$$P \sqsubseteq Q \quad \equiv \quad \mathbf{t}P \supseteq \mathbf{t}Q \ . \tag{4.29}$$

Note that the direction of the trace set inclusion is now reversed. A little more work, attempting to *add* traces while maintaining friendship, yields the following results:

$$\begin{aligned} \sqcap \mathit{Friends}.S_5 &= P_{10} \ , \\ \sqcap \mathit{Friends}.S_6 &= P_6 \ , \\ \sqcap \mathit{Friends}.S_7 &= P_7 \ , \\ \sqcap \mathit{Friends}.S_8 &= P_{10} \ . \end{aligned}$$

Consequently, S_8 is the canonical representative of its *equ*-class, but S_6 and S_7 are not the representatives of their respective classes. Since $\backsim P_{10} \sqsupseteq \backsim P_6$ and $\backsim P_{10} \sqsupseteq \backsim P_7$, we have, according to Theorem 4.6.4, also S_8 *sat* S_6 and S_8 *sat* S_7. ∎

Things become rapidly more complicated when more symbols are involved. Our third example is still manageable and considers systems with both external input and output.

4.6.3 Example Here are four systems, similar to the ones we have seen in the preceding two examples, but with external input a and external output b:

$$\begin{aligned} S_9 &= \{\ (\{\mathtt{a}\}, \varnothing, \{\varepsilon, \mathtt{a}\}),\ (\varnothing, \{\mathtt{b}\}, \{\varepsilon, \mathtt{b}\})\ \}\ , \\ S_{10} &= \{\ (\{\mathtt{a}\}, \{\mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{ab}\})\ \}\ , \\ S_{11} &= \{\ (\{\mathtt{a}\}, \{\mathtt{b}\}, \{\varepsilon, \mathtt{b}, \mathtt{ba}\})\ \}\ , \\ S_{12} &= \{\ (\{\mathtt{a}\}, \{\mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{ab}, \mathtt{ba}\})\ \}\ . \end{aligned}$$

Again, these systems are non-isomorphic and they do not fall into the same *equ*-class. The following six processes are candidates for friends:

$$\begin{aligned} P_{11} &= (\{\mathtt{b}\}, \{\mathtt{a}\}, \{\varepsilon\})\ , \\ P_{12} &= (\{\mathtt{b}\}, \{\mathtt{a}\}, \{\varepsilon, \mathtt{a}\})\ , \\ P_{13} &= (\{\mathtt{b}\}, \{\mathtt{a}\}, \{\varepsilon, \mathtt{b}\})\ , \\ P_{14} &= (\{\mathtt{b}\}, \{\mathtt{a}\}, \{\varepsilon, \mathtt{a}, \mathtt{ab}\})\ , \\ P_{15} &= (\{\mathtt{b}\}, \{\mathtt{a}\}, \{\varepsilon, \mathtt{b}, \mathtt{ba}\})\ , \\ P_{16} &= (\{\mathtt{b}\}, \{\mathtt{a}\}, \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{ab}, \mathtt{ba}\})\ . \end{aligned}$$

The actual friendships are:

	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}
S_9	*no*	*no*	*yes*	*no*	*yes*	*yes*
S_{10}	*yes*	*no*	*yes*	*yes*	*yes*	*yes*
S_{11}	*no*	*no*	*yes*	*no*	*yes*	*no*
S_{12}	*no*	*no*	*yes*	*no*	*yes*	*yes*

From that table above we infer that S_{10}, S_{11}, and S_{12} belong to three different *equ*-classes, while S_9 and S_{12} look equivalent. Because both input and output are involved, properties (4.28) and (4.29) are not applicable and it is more difficult to find the least friends.

Section 4.8 addresses the general case of computing the least friend. The idea is to decrease input capability and increase output capability while maintaining friendship. One then finds as least friends:

$$\begin{aligned}
\sqcap \mathit{Friends}.S_9 &= P_{16} , \\
\sqcap \mathit{Friends}.S_{10} &= P_{14} , \\
\sqcap \mathit{Friends}.S_{11} &= P_{15} , \\
\sqcap \mathit{Friends}.S_{12} &= P_{16} .
\end{aligned}$$

Thus, S_9 and S_{12} are indeed equivalent. Furthermore, we have $S_{10} = \{\backsim P_{14}\}$, $S_{11} = \{\backsim P_{15}\}$, and $S_{12} = \{\backsim P_{16}\}$; these are the canonical representatives of their *equ*-classes. On account of Theorem 4.6.4 and $\backsim P_{14} \sqsupseteq \backsim P_{16} \sqsupseteq \backsim P_{15}$ we have S_{10} *sat* S_{12} and S_{12} *sat* S_{11}. ∎

The following theorem motivates our choice of canonical representatives.

4.6.4 Theorem For systems S and T we have

$$\begin{aligned}
S &\quad \mathit{equ} \quad \{[\![S]\!]\} , \\
[\![S]\!] &= \sqcap \{P : P \in \mathcal{PROC} \wedge S \ \mathit{equ}\ \{P\} : P\} , \\
S \ \mathit{equ}\ T &\equiv [\![S]\!] = [\![T]\!] , \\
S \ \mathit{sat}\ T &= [\![S]\!] \sqsupseteq [\![T]\!] , \\
[\![S \ \mathit{par}\ T]\!] &= [\![S]\!] \parallel [\![T]\!] .
\end{aligned}$$

∎

The first statement in this theorem expresses that the canonical representative of S is indeed in the same *equ*-class as S. The second statement roughly says that the composite equals the $\sqsubseteq$-minimum of all singleton systems equivalent to S. The third statement expresses that the canonical representative is unique for each *equ*-class. The fourth statement, proved in Chapter 7 as Theorem 7.3.10, says that satisfaction corresponds to the $\sqsupseteq$-order on composites. Similarly, the fifth statement expresses that *par* corresponds to $\parallel$ on composites.

4.6.5 Note The implications from left to right in the third and fourth statement are elementary. Concerning the fourth statement, for instance, we derive

$$\begin{aligned}
& S \ \mathit{sat}\ T \\
\equiv \quad & \{ \text{ definition of } \mathit{sat} \ \} \\
& (\forall\, U : U \in \mathcal{SYS} : \mathit{Correct}.(S \ \mathit{par}\ U) \Leftarrow \mathit{Correct}.(T \ \mathit{par}\ U)) \\
\Rightarrow \quad & \{ \text{ definition of } \mathit{Friends} \ \} \\
& \mathit{Friends}.S \supseteq \mathit{Friends}.T \\
\Rightarrow \quad & \{ \text{ property of greatest lower bound } \} \\
& \sqcap \mathit{Friends}.S \sqsubseteq \sqcap \mathit{Friends}.T
\end{aligned}$$

$$\equiv \quad \{ \text{ reflection turns } \sqsubseteq \text{ around, definition of } [\![\_]\!] \ \}$$
$$[\![S]\!] \sqsupseteq [\![T]\!]$$

Antisymmetry of $\sqsubseteq$ then takes care of the third statement. ∎

Composites simplify the verification of satisfaction and equivalence. Consider systems S and T. When T is interpreted as a specification, then 'S satisfies T' and 'S implements T' are expressed by 'S *sat* T'. To ascertain this according to the definition of *sat*, involves a quantification over all systems (acting as testing environments for S and T). For each such environment U, one needs to compare $Correct.(S\ par\ U)$ and $Correct.(T\ par\ U)$, which involves another quantification over all reachable traces of the system. In case the composites of S and T are known, the problem simply boils down to ascertaining $[\![S]\!] \sqsupseteq [\![T]\!]$, which involves just a quantification over all process traces of the composites.

4.7 DI Processes and the JTU-Rules

As we have argued in Chapter 2, it does not make much sense to speak about a delay-insensitive system as such; delay-insensitivity is with respect to some specification. That is exactly what 'S *sat* T' expresses. Delay-insensitivity is implicit in the definition of correctness: the definition of 'free of interference' involves anisochronic operation under all possible delay conditions. Thus, if we say that some (closed) system S is free of interference, then that really includes the qualification 'independent of values for delays in connecting wires and other processes', that is, delay-insensitively.

The canonical processes defined in the preceding section suffice to describe processes, since any other process is equivalent to a canonical process. These particular canonical processes have certain nice properties. Since these are the the only processes needed (everything can be done in terms of them), and since they serve to describe and design "delay-insensitive" systems, let us call them delay-insensitive, or DI, processes.

By definition, processes that occur as composites of systems are called **DI processes**. The set $\mathcal{DI}$ of DI processes is therefore defined by

$$\mathcal{DI} \quad = \quad \{S : S \in \mathcal{SYS} : [\![S]\!]\} \ . \tag{4.30}$$

A fundamental result concerning $\mathcal{DI}$ is given in the following theorem.

4.7.1 Theorem We have

$$\langle \mathcal{SYS}; par, sat\rangle / equ \quad \text{is isomorphic to} \quad \langle \mathcal{DI}; \|, \sqsupseteq\rangle \ .$$

Proof Mapping $[\![\_]\!]$ is an surjective homomorphism from $\langle \mathcal{SYS}; par, sat\rangle$ to $\langle \mathcal{DI}; \|, \sqsupseteq\rangle$. ∎

That is, expressions over the algebra $\langle \mathcal{SYS}; par, sat\rangle$ modulo *equ*-equivalence can be transformed into logically equivalent expressions over the algebra $\langle \mathcal{DI}; \|, \sqsupseteq\rangle$. The latter algebra is called a fully-abstract model, because equivalence now boils down to equality.

4.7.2 Example The statement

$$(\forall\, S, U :: (\exists\, T :: S \; par \; T \; sat \; U))$$

where S, T, and U range over $\mathcal{SYS}$, corresponds to the logically equivalent statement

$$(\forall\, P, R :: (\exists\, Q :: P \parallel Q \sqsupseteq R))$$

where P, Q, and R range over $\mathcal{DI}$. ∎

The next theorem gives some alternative characterizations of $\mathcal{DI}$.

4.7.3 Theorem (*Characterization of DI processes*)
The following statements concerning process P are equivalent.

1. $P \in \mathcal{DI}$, that is, $(\exists\, S : S \in \mathcal{SYS} : [\![S]\!] = P)$,
2. $P = \sqcap \{Q : Q \in \mathcal{PROC} \wedge \{P\}\ equ\ \{Q\} : Q\}$,
3. $[\![\{P\}]\!] = P$,
4. $Correct.\{P, \rightsquigarrow P\}$,
5. P satisfies the JTU-Rules $\mathcal{W}$, $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ given below.

The first statement is just the definition of 'P is DI', namely that P is the composite of some system. The second statement expresses that P is the $\sqsubseteq$-minimum of "its" *equ*-class, that is, the class containing $\{P\}$. The third statement says that P is its "own" composite. The fourth statement expresses that P is free of interference with its reflection. Finally, the fifth statement is a closure property of P's trace set that is easy to verify for P's state graph (see below). ∎

The JTU-Rules are named after Jan Tijmen Udding, who first stated them in [Udd84]. The equivalence of the last two statements in Theorem 4.7.3 is non-trivial and proven in [Ver89].

We now define the four **JTU-Rules**. Let P be a process. The equivalence relation induced by the partition $\{\mathbf{i}P, \mathbf{o}P\}$ in $\mathbf{a}P$ is denoted by $\eqsim_P$. We leave out the subscript when it is obvious from the context. That is, $a \eqsim b$ expresses that a and b have the same direction with respect to P (either both a and b are inputs of P, or both are outputs) and $a \not\eqsim b$ expresses that a and b have opposite direction (one is an input and the other an output).

- P satisfies **Rule** $\mathcal{W}$ when for all traces s and symbols a we have

$$saa \notin \mathbf{t}P \ .$$

- P satisfies **Rule** $\mathcal{X}$ when for all traces s and t, and symbols a and b with $a \eqsim b$ we have

$$sabt \in \mathbf{t}P \equiv sbat \in \mathbf{t}P \ .$$

- P satisfies **Rule** $\mathcal{Y}$ when for all traces s and t, and symbols a, b, and c with $a \not\asymp c$ and $b \asymp a$ we have

$$sactb \in \mathbf{t}P \wedge scat \in \mathbf{t}P \;\Rightarrow\; scatb \in \mathbf{t}P \;.$$

- P satisfies **Rule** $\mathcal{Z}$ when for all traces s, and symbols a and c with $a \not\asymp c$ we have

$$sa \in \mathbf{t}P \wedge sc \in \mathbf{t}P \;\Rightarrow\; sac \in \mathbf{t}P \wedge sca \in \mathbf{t}P \;.$$

Rule $\mathcal{W}$ expresses that no signal may occur twice in immediate succession (because this would cause interference at the connecting wires). Rule $\mathcal{X}$ expresses that the order of signals in the *same* direction is irrelevant for future possibilities. Rule $\mathcal{Y}$ expresses that the order of signals in *opposite* direction is only to a limited extent relevant for future possibilities (some, but not all, possibilities after one order are also possible after the other order; this will be clarified below in Theorem 4.7.4). Rule $\mathcal{Z}$ expresses that signals of *opposite* direction cannot "disable" each other. Often a simpler version of Rule $\mathcal{Y}$ holds:

- P satisfies **Rule** $\mathcal{Y}'$ when for all traces s and t, and symbols a and c with $a \not\asymp c$, $sa \in \mathbf{t}P$, and $sc \in \mathbf{t}P$ we have

$$sact \in \mathbf{t}P \;\equiv\; scat \in \mathbf{t}P \;.$$

Note that $\mathcal{Y}'$ implies $\mathcal{Y}$. Rule $\mathcal{Y}'$ expresses that if two signals of *opposite* direction can both occur, then their order is irrelevant for future possibilities.

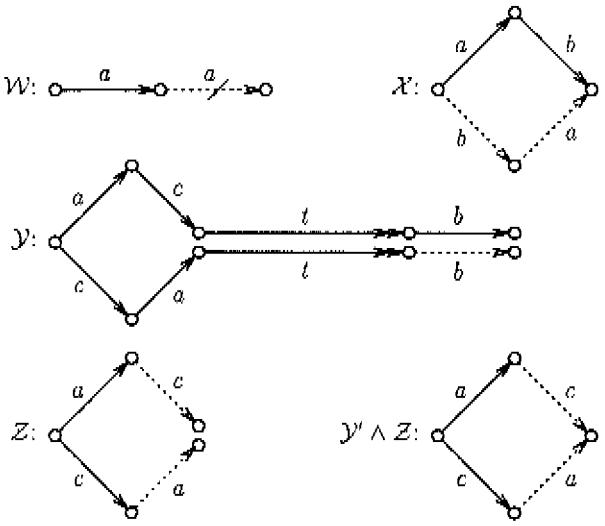


Figure 4.2: JTU-Rules in terms of state graphs: $a \asymp b$ and $a \not\asymp c$

Figure 4.2 illustrates the JTU-Rules in terms of state graphs. In this figure, symbols a and b have the same direction, and symbols a and c have opposite direction. If the solid edges are present in a DI state graph, then the dotted edges are also present (or not present when crossed, in case of Rule $\mathcal{W}$) in the given relationship to the other edges. Beware figures: they are often misleading. For instance, the double-arrowed edges labeled t represent a —possibly empty— path of edges, and these two paths could coincide in the state graph. Rule $\mathcal{Y}'$ has been illustrated in conjunction with $\mathcal{Z}$ because that is more convenient.

On account of the JTU-Rules, DI state graphs often contain rhombuses with opposite edges bearing the same label. To avoid clutter, we omit at times some edge labels in DI state graphs; these labels can then be restored by giving opposite edges in each rhombus the same label.

In terms of the after-operator, Rules $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Y}'$ can be formulated as follows.

4.7.4 Theorem Process P satisfies Rule $\mathcal{X}$ if and only if for all traces s and symbols a and b with $a \not\equiv b$ we have

$$sab \in \mathbf{t}P \;\Rightarrow\; sba \in \mathbf{t}P \wedge P/sab = P/sba \;.$$

Process P satisfies Rule $\mathcal{Y}$ if and only if for all traces s and symbols $a \in \mathbf{i}P$ and $c \in \mathbf{o}P$ we have

$$sac \in \mathbf{t}P \wedge sca \in \mathbf{t}P \;\Rightarrow\; P/sac \sqsubseteq P/sca \;.$$

Process P satisfies Rule $\mathcal{Y}'$ if and only if for all traces s and symbols a and c with $a \not\equiv c$ we have

$$sac \in \mathbf{t}P \wedge sca \in \mathbf{t}P \;\Rightarrow\; P/sac = P/sca \;.$$

Proof We will only do Rule $\mathcal{Y}$. Let P be a process satisfying Rule $\mathcal{Y}$. Assume $sac \in \mathbf{t}P$ and $sca \in \mathbf{t}P$ for trace s, input a, and output c. We prove $P/sac \sqsubseteq P/sca$. The input alphabets of these processes are equal to $\mathbf{i}P$ and the output alphabets to $\mathbf{o}P$. For trace t and output b we derive

$$\begin{array}{ll} & t \in \mathbf{t}P/sac \wedge tb \in \mathbf{t}P/sca \\ = & \quad \{ \text{ definition of after-operator } \} \\ & sact \in \mathbf{t}P \wedge scatb \in \mathbf{t}P \\ \Rightarrow & \quad \{ \text{ Rule } \mathcal{Y} \text{ with } a, c := c, a \} \\ & sactb \in \mathbf{t}P \\ \equiv & \quad \{ \text{ definition of after-operator } \} \\ & tb \in \mathbf{t}P/sac \end{array}$$

Similarly, one may derive for trace t and input b:

$$tb \in \mathbf{t}P/sac \wedge t \in \mathbf{t}P/sca \;\Rightarrow\; tb \in \mathbf{t}P/sca \;.$$

On account of the definition of $\sqsubseteq$, we thus have $P/sac \sqsubseteq P/sca$. ∎

The formulation of Rule $\mathcal{Y}$ in terms of the after-operator expresses that the environment has "more control" over a process when it waits for output c before sending input a rather than the other way round, because P/sca is "at least as good as" P/sac. The JTU-Rules can be further "condensed", but we postpone that until Section 7.2.

The elementary processes introduced in Chapter 3 are all in $\mathcal{DI}$, since they satisfy the JTU-Rules as is readily verified from their state graphs. In fact, they all satisfy Rule $\mathcal{Y}'$ as well.

4.7.5 Theorem $\mathcal{DI}$ is closed under composition and reflection.

Proof Of course, $\mathcal{DI}$ is closed under composition, since by definition $P \parallel Q = [\![\{P, Q\}]\!] \in \mathcal{DI}$ for any processes P and Q. On account of the equivalence of statements 0 and 3 (or 4) in Theorem 4.7.3, $\mathcal{DI}$ is also closed under reflection. ∎

The set of DI processes satisfying Rule $\mathcal{Y}'$ is also closed under reflection. However, it is *not* closed under composition. This came as a surprise at the time Udding formulated his rules. Here follows a simple example.

4.7.6 Example Consider DI processes P and Q given by their state graphs in Figure 4.3. System $\{P, Q\}$ has external input a, external outputs b and c, and internal

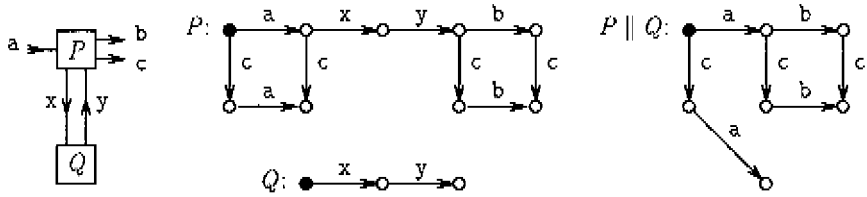


Figure 4.3: Two DI processes (left) and their composite (right)

connections x and y. Thus, at least five additional wires are involved in the operation of this system and its environment. Both P and Q satisfy Rule $\mathcal{Y}'$. When their composite $P \parallel Q$ is determined (shown in Figure 4.3; also see Section 4.8) it turns out not to satisfy Rule $\mathcal{Y}'$. Verify that Rule $\mathcal{Y}$ is satisfied by $P \parallel Q$. We would also like to point out that P and Q satisfy the requirements for composition in [Udd84] and that, after a suitable translation, $P \parallel Q = P \mathbf{b} Q$, where **b** is the blending operator of [Udd84].

Note that if the environment waits for output c from system $\{P, Q\}$ before offering it input a, then the system is guaranteed *not* to produce output b. On the other hand, if the environment immediately offers a, then the system will produce c eventually (it may already have done so, but c can still be "on its way") and output b is possible but not guaranteed. The DI Model lacks some features to argue about such progress properties of systems. We will pay more attention to this limitation in Section 5.5. ∎

Many tasks simplify considerably when DI processes are involved. The next two theorems illustrate this. The first theorem gives a condition under which isochronic and anisochronic operation are equivalent. It motivates the following definition. **A DI system** is a system such that of each pair of connected processes at least one is in $\mathcal{DI}$.

4.7.7 Theorem (*Fundamental property of DI processes*)
For closed DI system S we have

$$\text{'}\hat{S}\text{ is free of interference'} \quad \equiv \quad \text{'}S\text{ is free of interference'} .$$

Proof idea: The effects of additional wires are already incorporated in a DI process. ∎

Compare this to Theorem 4.3.8, which says that the implication from left to right holds in general. On account of Theorem 4.4.2, system S is equivalent to its wired version $\vec{S}$. System $\hat{S}$ is DI, because the additional wires are in $\mathcal{DI}$. Therefore, every system is equivalent to some DI system, by the suitable introduction of explicit DI wires.

4.7.8 Example Note that processes P and $\backsim P$ of system S in Examples 4.3.1 and 4.3.3 are not DI, since they do not satisfy Rule $\mathcal{X}$. Indeed, system S is free of interference, and $\vec{S}$ is *not* free of interference.

Processes P and $\backsim P$ are equivalent to DI processes Q and R respectively, given by

$$\begin{aligned} Q &= (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{ab}, \mathtt{ba}\}) , \\ R &= (\{\mathtt{a}, \mathtt{b}\}, \varnothing, \{\varepsilon, \mathtt{a}\}) . \end{aligned}$$

That is, $\{P\}$ *equ* $\{Q\}$ and $\{\backsim P\}$ *equ* $\{R\}$. Note, however, that $Q \neq \backsim R$. This provides a counterexample for the validity of $[\![\{\backsim P\}]\!] = \backsim[\![\{P\}]\!]$. ∎

4.7.9 Theorem For system S and DI process P we have

$$\begin{aligned} S \textit{ equ } \{P\} &\equiv [\![S]\!] = P , \\ S \textit{ sat } \{P\} &\equiv [\![S]\!] \sqsupseteq P , \\ S \textit{ sat } \{P\} &\equiv \textit{Correct}.(S \textit{ par } \{\backsim P\}) . \end{aligned}$$

∎

It is instructive to find a counterexample for each of the statements in Theorem 4.7.9 with $P \notin \mathcal{DI}$. The implications from right to left hold in general.

The first two statements of the theorem above are a direct consequence of Theorems 4.6.4 and 4.7.3. They show the advantage of specifying a system by means of a DI process rather than an arbitrary process. The third statement expresses that S *sat* $\{P\}$ holds if and only if $\backsim P$ is a friend of S. This is more remarkable than it may at first seem. The definition of S *sat* T involves *Correct*.$(S$ *par* $U)$ and *Correct*.$(T$ *par* $U)$ where U ranges over $\mathcal{SYS}$. In the particular case of DI singleton system $\{P\}$ for T, it suffices to compute just *Correct*.$(S$ *par* $\{\backsim P\})$ (note that P is reflected).

4.8 Computing the Composite

Given system S one can compute its composite $[\![S]\!]$ by (i) starting with some friend of S and (ii) repeatedly reducing it (that is, making it smaller with respect to $\sqsubseteq$) while maintaining friendship, until (iii) no further reduction is possible (this yields the least friend), and finally (iv) reflecting the result.

For step (i) first try process $(\mathbf{xo}S, \mathbf{xi}S, (\mathbf{xo}S)^*)$, the largest (easiest) candidate below $\top$. If that does not work then $Friends.S = \{\top\}$ and, hence, $[\![S]\!] = \bot$. Step (ii) is often best done by making small reductions at a time.

A process can be reduced with respect to $\sqsubseteq$ in two ways. One way is to restrict its willingness to receive inputs (by removing input edges in its state graph). The other way is to increase its capability to send outputs (by adding output edges in its state graph; but after this output all inputs should be accepted in order not to have reduced the process too much).

Observe that the least friend is in $\mathcal{DI}$. Using the JTU-Rules, it is easy to verify whether the "current" friend in step (ii) of the computation above is in $\mathcal{DI}$. If the "current" friend does not satisfy the JTU-Rules, then it should be possible to reduce it further.

Theorem 4.7.7 is helpful when system S consists of DI processes only. When looking for friends of S it is easy to choose the appropriate input and output alphabets, the main problem being interference. If S consists of DI processes only, then the condition of Theorem 4.7.7 is met by the closed system $S\ par\ \{P\}$. Therefore, in order to determine whether $(S\ par\ \{P\})^\neg$ is free of interference, it suffices to restrict oneself to *isochronic* operation.

4.8.1 Example Reconsider processes P and Q of Example 4.7.6. We show in some detail how to compute $P \parallel Q$.

Friends of $\{P, Q\}$, if any, have output alphabet $O = \{\mathtt{a}\}$ and input alphabet $I = \{\mathtt{b}, \mathtt{c}\}$. Note that x and y are internal to $\{P, Q\}$. Since both P and Q are DI processes we can restrict ourselves to isochronic operation when checking for interference, as pointed out above. The first candidate to try for friendship is (I, O, I^*) (see Figure 4.4, state graph R_1). Since $\{P, Q, R_1\}$ has no interference, R_1 is a friend. It is not the least friend, since it can be reduced (in many ways). Removing as many input edges as possible, while preserving friendship, yields R_2. Input c cannot be removed since P would then cause interference. A further reduction is still possible by adding outputs. Adding just one a at the initial state (yielding R_3), however, reduces it by too much, since P causes interference at R_3. Adding output a and after that accepting all inputs again, yields R_4. R_4 is a friend of $\{P, Q\}$, but it is not DI and, hence, not the least friend. Some of the inputs after a can be removed, yielding R_5, which is also a friend but still not DI. Addition of an output a after the initial input c and after that a all inputs again, yields R_6. This is again a smaller friend, but again not DI. It turns out that all inputs after the output a that was just added can be removed. The result is DI and cannot be reduced further. Its reflection is the composite shown on the right in Figure 4.3.

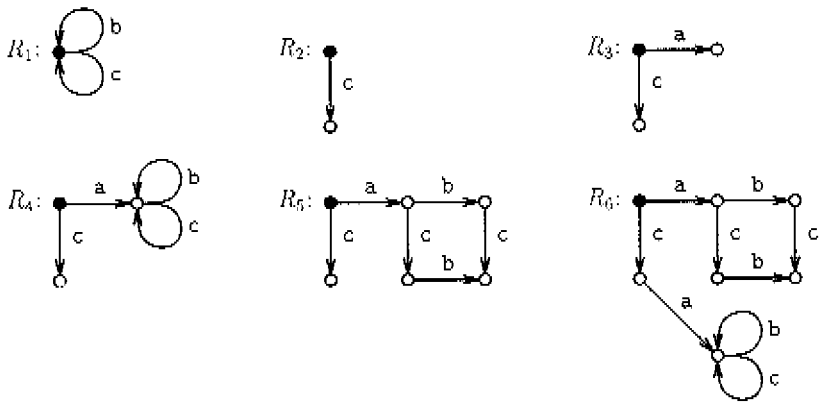


Figure 4.4: Candidates for friends of $\{P, Q\}$

Observe that the following ordering relationships hold:

$$R_1 \sqsupset R_2 \sqsupset R_4 \sqsupset R_5 \sqsupset R_6 \sqsupset \backsim(P \parallel Q) \ .$$

Furthermore $R_5 \sqsupset R_3$, and R_3 is incomparable to both R_6 and the least friend. With a little bit of experience bigger reduction steps are possible. ◢

4.8.2 Example Reconsider Example 3.2.5, concerning a C-element with forked output. It is now easy to verify according to the definitions that both $\backsim P_5$ and $\backsim Q_5$ are DI friends of S_5, and that $\backsim P_5 \sqsupset \backsim Q_5$. Although a bit tedious, exhaustive trial confirms that $\backsim Q_5$ cannot be reduced further while maintaining friendship. Hence, $\backsim Q_5$ is the least friend and Q_5 is the composite of S_5. ◢

Theorem 7.3.7 provides an alternative approach to the computation of composites (also see Note 7.3.9).

4.9 Design Equation

A designer is often confronted with the following problem. Given is a specification in the form of some process R. The designer conjectures that a particular process Q is part of an implementation, that is, the designer attempts to find a solution of the form $P \parallel Q$ for some still unknown process P. What is a specification for P? Obviously, P should satisfy

$$P \parallel Q \sqsupseteq R \ . \tag{4.31}$$

We call this the **design equation**, since designers often encounter it. It expresses that P composed with Q satisfies specification R. The next theorem characterizes all solutions of this design equation.

4.9.1 Theorem (*Factorization Theorem*)
For processes P, Q, and R in $\mathcal{DI}$ we have

$$P \parallel Q \sqsupseteq R \quad \equiv \quad P \sqsupseteq \backsim(Q \parallel \backsim R) \ .$$

Appendix B gives a proof that makes this theorem easier to memorize. ∎

It is called a factorization theorem because "factor" Q is "divided" out of R to obtain an explicit specification for P. In general, composition has no inverse, but the inequality expressed by the design equation (4.31) can be solved within $\mathcal{DI}$. In [Fan86], Fang introduces the notion of 'decomposition by factoring' and gives a (very operational) definition without proofs. The form of the Factorization Theorem reveals a **Galois connection** (see [Bir84, DP90]) between functions $\_ \parallel Q$ and $\backsim(Q \parallel \backsim\_)$. Factorization is similar to the **weakest prespecification**, which solves a design equation involving sequential composition (see [HJ86, HJ87]).

It is possible that the designer makes a wrong choice for Q, in the sense that there is no solution with this Q. In that case one finds $Q \parallel \backsim R = \bot$ and, hence, the specification for P boils down to $P \sqsupseteq \top$. This means that the only "solution" is $P = \top$, which is not a feasible solution, since $\top$ is an imaginary process. The introduction of the imaginary processes $\bot$ and $\top$ makes these sorts of case distinctions unnecessary.

Also note that taking the least solution for P, namely $P = \backsim(Q \parallel \backsim R)$, and plugging it into the design equation need not yield an equality. That is, in general we do *not* have

$$(\backsim(Q \parallel \backsim R)) \parallel Q \quad = \quad R \ . \tag{4.32}$$

The reason is that Q may already be "too good" to implement R minimally. No choice of P may be able to "annihilate" the excess goodness present in Q. A trivial example is obtained for $Q = \top \neq R$. In this case, we find that the least solution of (4.31) equals $\backsim(Q \parallel \backsim R) = \backsim\top = \bot$, that is, every process P is a solution. Consequently, taking $\bot$ for P, we find $P \parallel Q = \top \neq R$.

Examples 5.1.5 and 5.2.2 illustrate how the Factorization Theorem can be used for designing. Example 5.5.4 shows what may happen when an inappropriate factor is chosen. We finish this section with a theoretical application of the Factorization Theorem, which holds more generally for Galois connections.

4.9.2 Theorem Composition $\parallel$ on $\mathcal{DI}$ is $\sqcap$-continuous (distributes over arbitrary $\sqcap$ and, hence, is $\sqsubseteq$-monotonic), that is, for $P \in \mathcal{DI}$ and $W \subseteq \mathcal{DI}$ we have

$$P \parallel \sqcap W \quad = \quad \sqcap\{Q : Q \in W : P \parallel Q\} \ .$$

Proof Let P be a DI process and W a subset of $\mathcal{DI}$. It suffices to prove for all DI processes R

$$P \parallel \sqcap W \sqsupseteq R \quad \equiv \quad \sqcap\{Q : Q \in W : P \parallel Q\} \sqsupseteq R \ .$$

For DI process R we derive

$$
\begin{array}{ll}
& P \parallel \sqcap W \sqsupseteq R \\
= & \quad \{ \text{ Factorization Theorem } \} \\
& \sqcap W \sqsupseteq \backsim (P \parallel \backsim R) \\
\equiv & \quad \{ \text{ property of } \sqcap \} \\
& (\forall\, Q : Q \in W : Q \sqsupseteq \backsim (P \parallel \backsim R)) \\
\equiv & \quad \{ \text{ Factorization Theorem } \} \\
& (\forall\, Q : Q \in W : Q \parallel P \sqsupseteq R) \\
\equiv & \quad \{ \text{ property of } \sqcap \} \\
& \sqcap \{ Q : Q \in W : P \parallel Q \}
\end{array}
$$

■

Chapter 5

Applications

In the preceding chapter we have presented and analyzed the DI Model for the specification, composition, and refinement of delay-insensitive systems. In this chapter we will discuss some applications of the DI Model. In particular we look at composition and design problems. Along the way we introduce additional building blocks and study the phenomenon of output choice. Finally, we point out the limitations of the DI Model.

Most of the results in this chapter are not new, though everything is presented in a new and consistent framework. Also many of the (counter)examples are new. A good source for additional examples is [Ebe89].

5.1 Composition and Design Examples

So far we have introduced only a few kinds of building blocks, namely wires, I-wires, forks, merges, and C-elements. Not much can be accomplished with systems constructed of these building blocks alone. Before presenting additional building blocks we will look at some simple composition and design examples.

5.1.1 Example Let us consider the processes with a single port, say a. These can be used as **terminators** to avoid dangling inputs and outputs. Four kinds of terminators may be distinguished, depending on whether a is an input or an output, and whether the trace

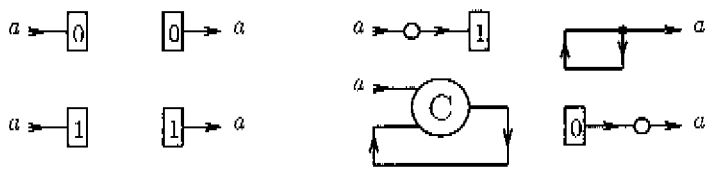


Figure 5.1: Diagrams (left) and designs (right) for the four terminators

set is $\{\varepsilon\}$ or $\{\varepsilon, a\}$[1]. A terminator with input is called a **sink** and with output a **source**. We use the prefix 0- for terminators with trace set $\{\varepsilon\}$ and 1- for trace set $\{\varepsilon, a\}$.

Diagrams for the terminators are shown on the left in Figure 5.1. Possible designs in terms of building blocks are given on the right, though in hardware realizations one would prefer other designs. ∎

5.1.2 Example Figure 5.2 shows two systems consisting of two merges each. It is easy to prove that the composites of both are equal to the **three-input merge** with inputs $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, output $\mathtt{d}$, and a trace set generated by the **regular expression** $((\mathtt{a} + \mathtt{b} + \mathtt{c})\mathtt{d})^*$, where union is denoted by + (with the weakest binding power), catenation by juxtaposition, and Kleene closure by * (with the strongest binding power). The trace set generated by regular expression RE consists of all symbol sequences matching RE, and all of their prefixes (initial segments).

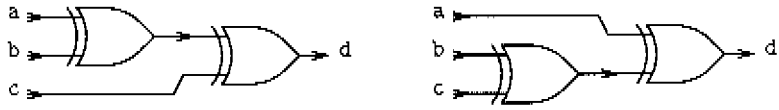


Figure 5.2: Two systems of two merges

Even though the two systems are equivalent within the DI Model, one may be preferred over the other because of performance differences, for example when one of the inputs occurs much more often or is more time critical than the other inputs. ∎

Similarly, one can obtain a three-output fork and three-input C-element, with trace sets generated by the regular expressions $(\mathtt{a}(\mathtt{b}, \mathtt{c}, \mathtt{d}))^*$ and $((\mathtt{a}, \mathtt{b}, \mathtt{c})\mathtt{d})^*$ respectively, where the **comma** operator (with a binding power stronger than union and weaker than catenation) denotes arbitrary interleaving. Larger multiple-input merges and C-elements, and multiple-output forks can be constructed by further cascading the binary versions into larger trees.

In spite of the simplicity of these five building blocks, it is often non-trivial to compute the composite of systems built from them.

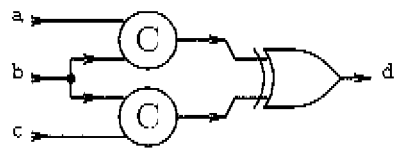


Figure 5.3: System of fork, 2 C-elements, and merge

[1] A trace set with aa is not interesting because, in the case of input, it is equivalent to $\{\varepsilon, a\}$ and, in the case of output, to $\perp$.

5.1.3 Example Consider the system of Figure 5.3 consisting of a fork, two C-elements, and a merge. You are challenged to compute the composite.

The state graph of the composite is depicted in Figure 5.4. Note that the state labeled 3 has no outgoing edges, because supplying a b-input might result in interference at the merge. Also note that the states labeled 4 and 5 are distinct, because different inputs are acceptable. ∎

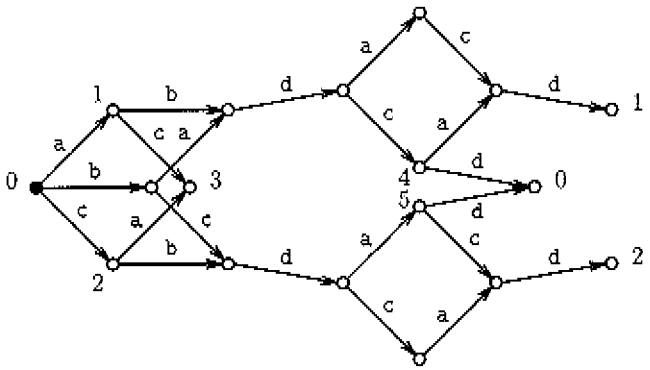


Figure 5.4: State graph of composite

5.1.4 Example A **rendez-vous** is a process P with two inputs $\{\mathtt{a}, \mathtt{b}\}$ and two outputs $\{\mathtt{d}, \mathtt{e}\}$. Its diagram is shown on the left Figure 5.5 and its state graph on the right (the initial state is at the center). Observe that the state graph satisfies the JTU-Rules and, hence, $P \in \mathcal{DI}$.

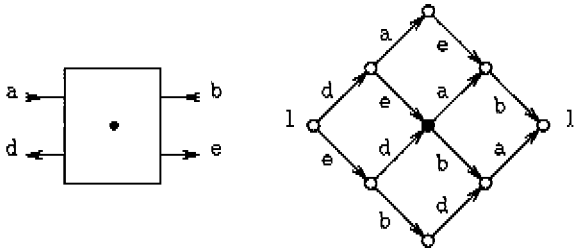


Figure 5.5: Diagram and DI state graph of rendez-vous

The communication behavior of P restricted to $\{\mathtt{a}, \mathtt{d}\}$ is generated by the regular expression $(\mathtt{ad})^*$; similarly, restricted to $\{\mathtt{b}, \mathtt{e}\}$ by $(\mathtt{be})^*$. Consequently, the rendez-vous can operate in a "split environment" where each half alternates output and input. The rendez-vous synchronizes the cycles of the two halves. It is also known as **passivator** in the handshake circuits to which Tangram is compiled (see [vB93]).

System S_5 in Example 3.2.5, consisting of a C-element with forked output (see Figure 3.3), refines the rendez-vous: S_5 *sat* $\{P\}$ because $[\![S_5]\!] \sqsupseteq P$ (see Figure 3.5 for the composite Q_5 of S_5). In fact, S_5 is strictly better than required by specification P, that is, $\neg(S_5$ *equ* $\{P\})$. An informal argument for this is that S_5 is capable of processing input b after output d and also input a after output e. More formally, consider process R with inputs $\{\mathtt{d},\mathtt{e}\}$, outputs $\{\mathtt{a},\mathtt{b}\}$, and a trace set generated by the regular expression $(\mathtt{a},\mathtt{b})(\mathtt{db},\mathtt{ea})$. For this R we have

$$\neg \mathit{Correct}.\{P,R\} \quad \wedge \quad \mathit{Correct}.(S_5 \mathit{\ par\ } \{R\}) \tag{5.1}$$

and, hence, $\neg(\{P\}$ *sat* $S_5)$ holds.

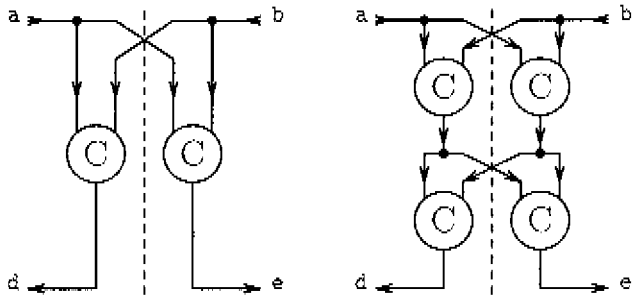


Figure 5.6: Diagrams of systems T (left) and U (right)

Although the C-element with forked output is a compact implementation of the rendez-vous, it cannot be distributed symmetrically over the two parties it synchronizes. System T presented on the left in Figure 5.6 is an attempt at a distributed implementation (the dashed line indicating the distribution). However, if one computes $[\![T]\!]$ then it turns out to yield process P_5 of Figure 3.4, which is not a refinement of the rendez-vous. Informally speaking, if the environment offers input a immediately after T has output d, then this may cause interference.

By suitably combining two copies of T one obtains system U on the right in Figure 5.6. The composite of system U turns out to be Q_5 and, thus, U is a proper implementation of the rendez-vous. System U can be distributed over the two parties it synchronizes as indicated by the dashed line. Note that the resulting two halves are connected by *four* wires. ∎

This example was inspired by [vdSU86] and shows that even the simple systems using only C-elements and forks involve subtle behavioral complications. Another case is provided in Example 5.2.1.

The fork is the only building block so far with more than one output. We now consider another such process, having input a, outputs $\{\mathtt{b},\mathtt{c}\}$, and a trace set generated by the regular expression $(\mathtt{abac})^*$. It is called a **toggle**, because each odd occurrence of input a

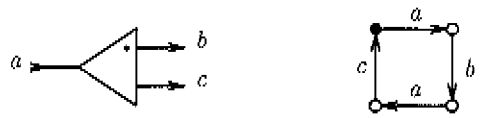


Figure 5.7: Diagram and DI state graph of toggle $\mathtt{T}(a; b, c)$

triggers output b, each even occurrence triggers c. This toggle cannot be constructed from the building blocks so far.

From now on we consider the toggle a building block. The toggle $\mathtt{T}(a; b, c)$ with input a and two outputs b and c is defined by the DI state graph in Figure 5.7. Its diagram is shown on the left. Note that we have

$$\mathtt{T}(a; b, c)/ab \;=\; \mathtt{T}(a; c, b)\ . \tag{5.2}$$

A four-output toggle with trace set $(\mathtt{abacadae})^*$ is easily built from three basic toggles. A three-output toggle with trace set $(\mathtt{abacad})^*$ can be obtained from a four-output toggle and a merge by feeding back one output:

$$\{\ \mathtt{M(a,e;x)},\ \ \mathtt{T(x;y,z)},\ \ \mathtt{T(y;b,d)},\ \ \mathtt{T(z;c,e)}\ \}\ . \tag{5.3}$$

5.1.5 Example Design a **2-phase-to-4-phase converter** R with inputs $\{\mathtt{a},\mathtt{d}\}$, outputs $\{\mathtt{b},\mathtt{c}\}$, and trace set $(\mathtt{acdcdb})^*$. Its environment can be split into two parties, one with communication behavior $(\mathtt{ab})^*$, the other with $(\mathtt{cd})*$. The converter sees to it that each ab-cycle (consisting of two phases, one a and one b) encloses two cd-cycles (having four phases).

A rather naive design technique that sometimes works is based on **output analysis**. In this approach, one considers each output of the circuit to be designed and analyses the conditions under which it is to be produced. The phase converter has outputs b and c. Inspecting the regular expression specifying this circuit, one sees that output b is produced by the second d-input, and that output c is produced by either the a-input or the first d-input. Consequently, we need a toggle to split the d-input into odd and even occurrences, and a merge to combine the two "causes" for the c-output. This results in the design shown on the left in Figure 5.8.

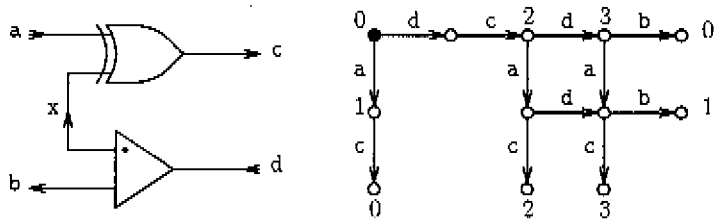


Figure 5.8: Diagram and state graph of design for phase converter

We emphasize that a design obtained by output analysis must always be verified afterwards, for instance, by computing the composite and comparing it to the specification. Example 5.2.1 illustrates what may go wrong.

It is a nice exercise to compute $M \parallel T$, where $M = \mathtt{M(a,x;c)}$ and $T = \mathtt{T(d;x,b)}$. The state graph of the composite is shown on the right in Figure 5.8. Because specification R is in $\mathcal{DI}$, we can immediately infer from Theorem 4.7.9 that R and $M \parallel T$ are not equivalent. In fact, also according to Theorem 4.7.9, $\{M, T\}$ *sat* R holds since $M \parallel T \sqsupseteq R$. The refinement can also be deduced from *Correct.*$\{M, T, \backsim R\}$. This illustrates the advantage of having DI specifications.

We can also tackle this design problem by **factorization**. Suppose we somehow guess the need for toggle $T = \mathtt{T(d;x,b)}$, where x is some internal symbol. We are now interested in the specification P of the remainder that composed with toggle T refines converter R. According to the Factorization Theorem (Theorem 4.9.1), specification P is obtained by computing

$$P \;=\; \backsim(T \parallel \backsim R)\ .$$

Note that by definition, $P = \sqcap$ *Friends.*$\{T, \backsim R\}$. Process P has inputs $\{\mathtt{a}, \mathtt{x}\}$ and output $\mathtt{c}$, and its trace set turns out to be generated by the regular expression $(\mathtt{acxc})^*$, which is the reflection of a toggle and can be refined by merge $M = \mathtt{M(a,x;c)}$. This is the same design as before, but now it is correct by construction. ∎

5.2 More Building Blocks

There are still many of systems that cannot be constructed from our building blocks so far. For instance, is it possible to make a **first-rest discriminator**, with input $\mathtt{a}$, outputs $\{\mathtt{b}, \mathtt{c}\}$, and trace set $\mathtt{ab(ac)}^*$? (Why not?) The following building blocks extend the range of possibilities.

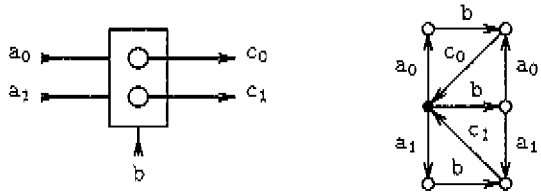


Figure 5.9: Diagram and DI state graph of latch $\mathtt{L}(a_0, a_1, b; c_0, c_1)$

Figure 5.9 shows the diagram and DI state-graph of **latch** $\mathtt{L}(a_0, a_1, b; c_0, c_1)$. The diagram and DI state graph of **decision-wait** $\mathtt{D}(a_0, a_1, b_0, b_1; c_{00}, c_{01}, c_{10}, c_{11})$ is given in Figure 5.10. These processes can be viewed as generalizations of the C-element. The latch temporarily stores a binary decision, and is sometimes also referred to as switch. The decision-wait waits for two binary decisions and reports which of the four combinations occurred.

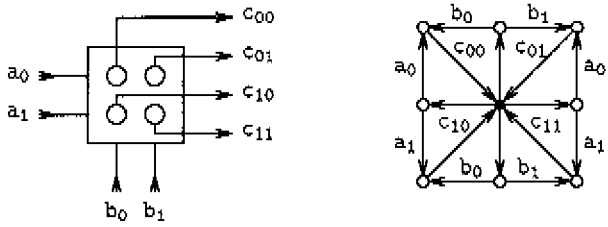


Figure 5.10: Diagram and DI state graph of decision-wait $\mathrm{D}(a_0, a_1, b_0, b_1; c_{00}, c_{01}, c_{10}, c_{11})$

5.2.1 Example Let us try to design latch $L = \mathrm{L}(\mathtt{a}_0, \mathtt{a}_1, \mathtt{b}; \mathtt{c}_0, \mathtt{c}_1)$ by the technique of output analysis. Output $\mathtt{c}_i$ is produced by the combined occurrence of inputs $\mathtt{a}_i$ and $\mathtt{b}$. Thus, we need a fork on input $\mathtt{b}$ and two C-elements to combine each $\mathtt{a}_i$ with a copy of $\mathtt{b}$. This gives rise to system T depicted in Figure 5.11.

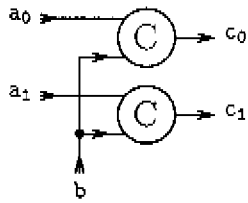


Figure 5.11: System T that does *not* implement a latch

Unfortunately, T does not implement latch L, that is, $\neg(T \; sat \; \{L\})$, because for process R with inputs $\{\mathtt{c}_0, \mathtt{c}_1\}$, outputs $\{\mathtt{a}_0, \mathtt{a}_1, \mathtt{b}\}$, and a trace set generated by the regular expression $\mathtt{a}_0\mathtt{b}\mathtt{c}_0\mathtt{a}_1$, we have

$$Correct.\{L, R\} \quad \wedge \quad \neg Correct.(T \; par \; \{R\}) \; . \tag{5.4}$$

The second conjunct holds because of interference at R. Less formally one might phrase this by saying that T can produce an "incorrect" output by doing $\mathtt{a}_0\mathtt{b}\mathtt{c}_0\mathtt{a}_1\mathtt{c}_1$, because output from the lower C-element remains enabled after the first $\mathtt{b}$-input.

One could also take R with trace set generated by $\mathtt{a}_0\mathtt{b}\mathtt{c}_0\mathtt{a}_0\mathtt{b}$. This R also satisfies (5.4), though now there is interference at T. It reveals that T cannot process all inputs required by specification L.

Yet another way to ascertain $\neg(T \; sat \; \{L\})$ is to recall Theorem 4.7.9 and to observe that $L \in \mathcal{DI}$ and $\neg Correct.(T \; par \; \{\backsim L\})$. One could also compute $[\![T]\!]$ and compare it to L. ∎

This example shows that designing by output analysis does not always work, for it failed to give us a decomposition of the latch. In fact, the latch cannot be implemented by the earlier building blocks at all (a nice proof of this folk theorem is still lacking). The latch can, however, be obtained from a decision-wait together with some terminators to hide

unneeded ports (use a C-element, or fork, with feedback for this purpose). The latch may, thus, be viewed as a 2×1-decision-wait. The decision-wait cannot be made from latches and the earlier building blocks (this is another folk theorem).

It is possible to construct larger latches (latching n inputs) from basic latches, for instance, through factorization.

5.2.2 Example We briefly show how to derive a design for the ternary latch L_3 with inputs $\{\mathtt{a}_0, \mathtt{a}_1, \mathtt{a}_2, \mathtt{b}\}$, outputs $\{\mathtt{c}_0, \mathtt{c}_1, \mathtt{c}_2\}$, and a trace set generated by

$$((\mathtt{a}_0, \mathtt{b})\mathtt{c}_0 + (\mathtt{a}_1, \mathtt{b})\mathtt{c}_1 + (\mathtt{a}_2, \mathtt{b})\mathtt{c}_2)^* \ . \tag{5.5}$$

We guess that latch $L = \mathrm{L}(\mathtt{x}, \mathtt{a}_2, \mathtt{b}; \mathtt{y}, \mathtt{c}_2)$, where symbols $\mathtt{x}$ and $\mathtt{y}$ are internal, might be useful. Factorization yields specification P for the remainder; P has inputs $\{\mathtt{a}_0, \mathtt{a}_1, \mathtt{y}\}$ and outputs $\{\mathtt{c}_0, \mathtt{c}_1, \mathtt{x}\}$. Computing $\backsim(L \parallel \backsim L_3)$ yields a trace set generated by

$$(\mathtt{a}_0\mathtt{x}\mathtt{y}\mathtt{c}_0 + \mathtt{a}_1\mathtt{x}\mathtt{y}\mathtt{c}_1)^* \ . \tag{5.6}$$

Doing an output analysis of the expression suggests that output $\mathtt{x}$ may be produced by a merge of inputs $\mathtt{a}_0$ and $\mathtt{a}_1$. However, these inputs are no doubt still needed to generate the corresponding c-outputs. Therefore we also introduce two forks $F_0 = \mathrm{F}(\mathtt{a}_0; \mathtt{d}_0, \mathtt{e}_0)$ and $F_1 = \mathrm{F}(\mathtt{a}_1; \mathtt{d}_1, \mathtt{e}_1)$, and merge $M = \mathrm{M}(\mathtt{d}_0, \mathtt{d}_1; \mathtt{x})$, where symbols $\{\mathtt{d}_0, \mathtt{d}_1, \mathtt{e}_0, \mathtt{e}_1\}$ are internal. The composite $Q = F_0 \parallel F_1 \parallel M$ has a trace set generated by

$$(\mathtt{a}_0(\mathtt{d}_0, \mathtt{x}) + \mathtt{a}_1(\mathtt{d}_1, \mathtt{x}))^* \ . \tag{5.7}$$

Factorizing P with respect to the further guess Q, yields a process with inputs $\{\mathtt{e}_0, \mathtt{e}_1, \mathtt{y}\}$, outputs $\{\mathtt{c}_0, \mathtt{c}_1\}$, and a trace set generated by

$$((\mathtt{e}_0, \mathtt{y})\mathtt{c}_0 + (\mathtt{e}_1, \mathtt{y})\mathtt{c}_1)^* \ . \tag{5.8}$$

This is easily recognized as latch $\mathrm{L}(\mathtt{e}_0, \mathtt{e}_1, \mathtt{y}; \mathtt{c}_0, \mathtt{c}_1)$. Thus we have derived the design in Figure 5.12. ∎

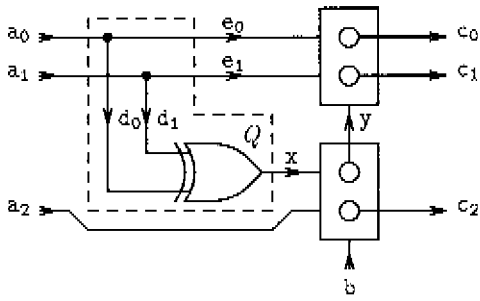


Figure 5.12: Design for a ternary latch

Similarly, larger decision-waits can be constructed, not only of type $m \times n$ but also with more than two dimensions, such as $2 \times 2 \times 2$ (see Figure 5.13).

The next example explains a design technique based on the idea of a **state machine**.

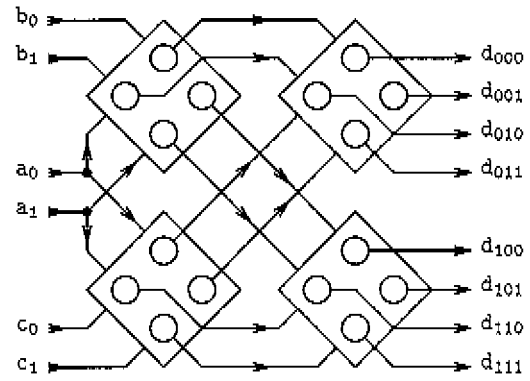


Figure 5.13: Design for $2 \times 2 \times 2$-decision-wait

5.2.3 Example Reconsider the problem of designing the phase converter of Example 5.1.5. The phase converter can be considered to have three states. In the first state, it only expects input a and reacts to it with output c and a change to the second state. In the second state, it responds to input d with output c again, going to the third state. Finally, in the third state, it produces output b on input d and returns to the first state. This suggests a design based on a state machine.

There are many ways to encode the state of such a state machine. The so-called **one-hot code** introduces a wire for each state. Only the wire corresponding to the current state is "active". A 2×3-decision-wait is used to determine the combination of input and state. Some additional circuitry translates this "combination" signal into the appropriate outputs and selects the next state. An I-wire takes care of selecting the initial state.

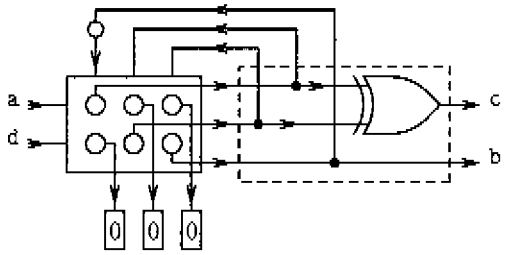


Figure 5.14: State-machine design for phase converter

The resulting design is presented in Figure 5.14. The dashed box encloses the output and next-state circuitry. Assuming the environment adheres to its obligations, some combinations of input and state do not occur. The corresponding outputs of the decision-wait are connected to 0-sinks. In contrast to the design in Example 5.1.5, this design is equivalent to the specification. ∎

In spite of their generality, state machines cannot be used for all design problems. In particular, it is difficult to deal with concurrent inputs. Section 5.4 describes a (partial) solution. Also, state machine designs are often inefficient. Choosing a better state space may improve the design, but may complicate the correctness argument. For the phase converter in the preceding example, one could do with two states, because the environment's choice of input also provides state information.

The state-machine approach can be applied successfully to the design of a toggle and a first-rest discriminator. In both cases there is only one input and there are just two states, so a latch suffices. For the first-rest discriminator a 1-source is needed.

5.3 Output Choice

None of the building blocks so far involves a choice between outputs: if in *some* state either of two outputs can be produced then both outputs can be produced "together", that is, in either order. Put differently, the occurrence of one output does not disable another output.

More formally, we say that symbols a and c of process P are mutually **non-disabling** when for all traces s we have

$$sa \in \mathbf{t}P \wedge sc \in \mathbf{t}P \;\Rightarrow\; sac \in \mathbf{t}P \wedge sca \in \mathbf{t}P \;. \tag{5.9}$$

Furthermore, we introduce two additions to Rule $\mathcal{Z}$ and two additions to Rule $\mathcal{Y}$:

- P satisfies **Rule** $\mathcal{Z}^{out}$ when all pairs of *distinct output* symbols are non-disabling;
- P satisfies **Rule** $\mathcal{Z}^{in}$ when all pairs of *distinct input* symbols are non-disabling;
- P satisfies **Rule** $\mathcal{Y}^{out}$ when for all traces s and t, input a, and outputs b and c we have

  $$sactb \in \mathbf{t}P \wedge scat \in \mathbf{t}P \;\Rightarrow\; scatb \in \mathbf{t}P \;.$$

- P satisfies **Rule** $\mathcal{Y}^{in}$ when for all traces s and t, output a, and inputs b and c we have

  $$sactb \in \mathbf{t}P \wedge scat \in \mathbf{t}P \;\Rightarrow\; scatb \in \mathbf{t}P \;.$$

Rule $\mathcal{Z}^{out}$ expresses that process P has no output choice, and Rule $\mathcal{Z}^{in}$ expresses the absence of input choice. Recall that Rule $\mathcal{Z}$ requires that all symbol pairs of *opposite* direction are non-disabling. The conjunction of Rules $\mathcal{Z}$, $\mathcal{Z}^{out}$, and $\mathcal{Z}^{in}$ will be called **Rule** $\mathcal{Z}'$. Thus, we have

$$\mathcal{Z}' \;\equiv\; \mathcal{Z} \wedge \mathcal{Z}^{out} \wedge \mathcal{Z}^{in} \;. \tag{5.10}$$

Rule $\mathcal{Z}$ expresses that all pairs of *distinct* symbols are non-disabling. A similar relationship holds for the four forms of Rule $\mathcal{Y}$:

$$\mathcal{Y}' \;\equiv\; \mathcal{Y} \wedge \mathcal{Y}^{out} \wedge \mathcal{Y}^{in} \;. \tag{5.11}$$

Rules $\mathcal{Z}^{in}$ and $\mathcal{Y}^{in}$ are introduced only for completeness' sake and do not play an important role.

All building blocks so far satisfy Rules $\mathcal{Z}^{out}$, $\mathcal{Y}^{out}$, and $\mathcal{Y}^{in}$. All but the merge even satisfy Rule $\mathcal{Z}^{in}$. The merge process requires of the environment a choice between inputs.

That Rule $\mathcal{Y}'$ is not preserved under composition was already shown in Example 4.7.6. Process P of that example, however, does not satisfy Rule $\mathcal{Z}^{out}$, since it involves a choice between outputs x and c. Here is different example illustrating that output choice is not crucial. For yet another example, see Example 5.4.2.

5.3.1 Example Figure 5.15 shows the state graphs of DI processes P, Q, and their composite $P \| Q$. Process P is willing to accept input b only after it has output x. Process Q produces output a upon receiving input x or c.

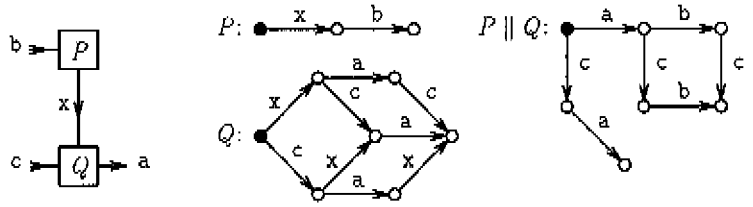


Figure 5.15: DI state graphs of processes P and Q, and their composite

Both P and Q satisfy Rule $\mathcal{Y}'$ and also Rule $\mathcal{Z}'$, so there is no choice between outputs. The composite of P and Q, which happens to be the reflection of the composite in Example 4.7.6, satisfies Rule $\mathcal{Z}'$ but not Rule $\mathcal{Y}^{in}$. Of course, it does satisfy Rule $\mathcal{Y}$ (and also $\mathcal{Y}^{out}$, see Theorem 5.3.3 below). ◢

The set of DI processes satisfying Rule $\mathcal{Z}'$ is not closed under composition either. Examples 3.2.5 and 4.8.2, featuring the C-element with forked output, show that input choice can arise through composition of processes that do not involve any choice. The C-element and fork satisfy $\mathcal{Z}'$ but the composite does not, since it requires the environment to choose between inputs a and b in states 2 and 3 (see state graph of Q_5 in Figure 3.5).

Also not closed under composition is the set of DI processes satisfying Rule $\mathcal{Z}^{out}$. This is illustrated by the following example.

5.3.2 Example Consider DI processes P and Q defined in Figure 5.16. Both processes satisfy Rule $\mathcal{Z}^{out}$, though neither $\mathcal{Y}'$ nor $\mathcal{Y}^{out}$ is satisfied. Their composite, however, does not satisfy Rule $\mathcal{Z}^{out}$: there is a choice between outputs c and d. It is impossible for P to be in the lower part of its state graph while Q is in the higher part, since that would require both processes to have received a signal before having sent any.

Note that, in a sense, there is a possibility of deadlock in system $\{P, Q\}$. Deadlock occurs when both processes start doing output, with P ending up in state 2 and Q in state 3, after which both processes are waiting for input. This scenario is possible because of the delaying nature of connecting wires. ◢

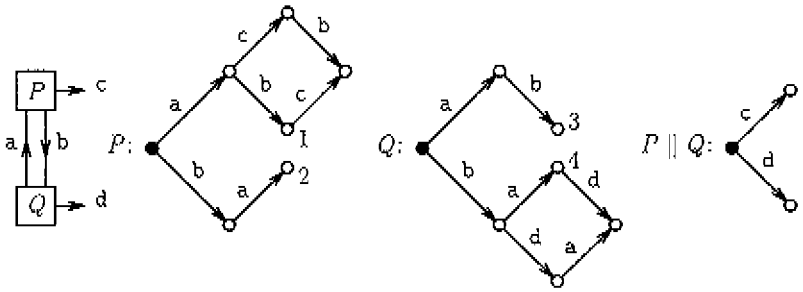


Figure 5.16: DI state graphs of processes P and Q, and their composite

The next theorem, finally, provides a closure result for processes without output choice.

5.3.3 Theorem The set of DI processes satisfying Rules $\mathcal{Y}^{out}$ and $\mathcal{Z}^{out}$ is closed under composition.

Proof See [Waa89]. ∎

The Extended DI Model of Chapter 6 sheds more light on these issues.

5.4 Still More Building Blocks

On account of Theorem 5.3.3 and the fact that all building blocks so far satisfy Rules $\mathcal{Y}^{out}$ and $\mathcal{Z}^{out}$, additional building blocks are needed to construct (the equivalent of) systems with output choice. Below we present three additional building blocks involving output choice.

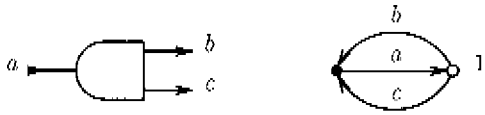


Figure 5.17: Diagram and DI state graph of undetermined selector $\mathtt{U}(a; b, c)$

Figure 5.17 shows the diagram and DI state-graph specification of **undetermined selector** $\mathtt{U}(a; b, c)$ with input a and two outputs b and c. The undetermined selector responds to each input a with a single output on either b or c. In state 1, the specification only prescribes that a choice be made between outputs b and c, not which one to choose. The user of such a process cannot know from the specification alone what choice will be made. For all one knows, the choice may depend on the flip of a coin or the same output may be produced every time. The specification is said to exhibit (output) nondeterminism. Observe that the undetermined selector is related to a merge process:

$$\mathtt{U}(a; b, c)/a \;=\; \backsim\mathtt{M}(b, c; a) \;.$$

Without ill effect, the undetermined selector can be replaced by, for instance, a toggle, because $\mathtt{T}(a; b, c) \sqsupseteq \mathtt{U}(a; b, c)$. Hence, a designer can always eliminate the nondeterminism introduced by undetermined selectors. An undetermined selector might be used because, at the current stage, it is not clear which deterministic refinement of the undetermined selector is most suitable.

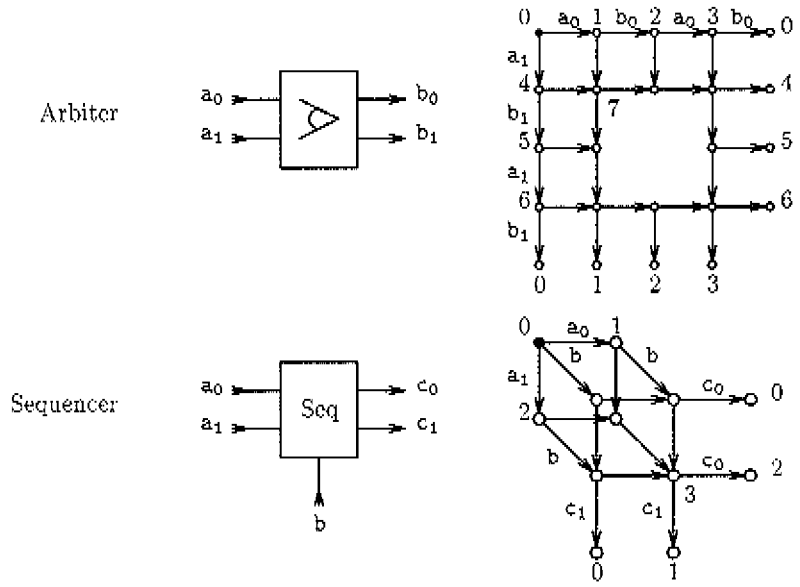


Figure 5.18: Diagrams and DI state graphs of arbiter and sequencer

The kind of output choice present in the undetermined selector is less useful than that of the following processes. Figure 5.18 shows the diagrams and DI state-graphs of **arbiter** $\mathtt{A}(a_0, a_1; b_0, b_1)$ and **sequencer** $\mathtt{S}(a_0, a_1, b; c_0, c_1)$. These processes also involve output choice: for the arbiter in state 7 and for the sequencer in state 3. Thus, they also exhibit nondeterminism. In this case, however, the nondeterminism cannot be eliminated by the designer without violating the (intuitive) progress conditions. This will be explained in more detail in Chapter 6.

The arbiter communicates with two parties, say P_0 and P_1. The parties negotiate with the arbiter to obtain a privilege, which the arbiter grants to only one of them at a time. P_i is hooked up through a_i and b_i. The first input a_i **requests** the privilege. The first output b_i **grants** the privilege. The second input a_i **releases** the privilege. Finally, the second output b_i **acknowledges** the release, allowing the cycle to start anew. The "hole" in the center of the state graph expresses that the privilege is to be granted to at most one party at a time. The specification does not express anything about fairness: whenever there are two outstanding requests the arbiter is free to grant either of them, no matter who got the privilege last time.

The sequencer is a refinement of the latch. Since both are defined by a DI process, this statement boils down to

$$\mathsf{S}(a_0, a_1, b; c_0, c_1) \quad \sqsupseteq \quad \mathsf{L}(a_0, a_1, b; c_0, c_1) \ .$$

In fact, the sequencer improves the latch, in that the environment is not required to choose between inputs a_0 and a_1. Whenever the sequencer is offered all three inputs "simultaneously", it is free to choose which a_i to "pass" to the corresponding c_i-output and which one to "hold" till the next b-input. Therefore, input b is called the 'next' input. The sequencer, thus, sequences inputs a_i to outputs c_i "clocked" by input b. The raison d'être of the switch is that its implementation in terms of transistors is cheaper than that of the sequencer.

The undetermined selector can be made from a sequencer. The arbiter can be designed in terms of the sequencer, and the other way round. Also larger arbiters and sequencers can be constructed (see [Ebe90]). A sufficiently large sequencer can be used at the input end of a state-machine design to take care of concurrent inputs. Its 'next' input is derived from the state wires by a merge. See [JNH93] for a more detailed exposition.

Let us now consider how much we can make with the building blocks that have been presented so far. Can we construct all **finite-state** processes, that is, processes with finite minimal state graphs? It is conjectured that the following five kinds of building blocks, namely

I-wire, merge, fork, decision-wait, and arbiter,

suffice to implement all finite-state processes (see [Ebe89][2]), taking into account "obvious progress requirements" to rule out "bogus" implementations (using, for instance, the do-nothing-wrong process mentioned in the next section). In fact, it is still an open problem whether there exists a "small" set of building blocks to implement all finite state processes, let alone a set of building blocks with efficient hardware realizations.

Below we motivate each of the five kinds of building blocks in the conjecture. We call set V of DI processes **refinement closed** when

$$(\forall\, P, Q : P \in V \ \wedge\ P \sqsupseteq Q : Q \in V) \ , \tag{5.12}$$

that is, when every process that can be implemented with some process from V itself also belongs to V. Observe that the set of processes with no more inputs than outputs is closed under composition and refinement. Among the five kinds, all but the **merge** have no more inputs than outputs and, hence, the merge is indispensable. Similarly, the set of processes with no more outputs than inputs is closed under composition and refinement. All but the **fork** belong to this set, so the fork is also indispensable. The set of **passive** processes that do not start with output is closed under composition and refinement too. The **I-wire** is the only process among the five kinds that is not passive and, thus, it is indispensable. The **arbiter** is also indispensable, but the argument must be postponed till Chapter 8 where

[2] Ebergen uses *RCEL* components instead of decision-waits.

output nondeterminism is analyzed in greater detail. The arbiter is the only process with output choice; the other four are in the set of processes satisfying Rules $\mathcal{Y}^{out}$ and $\mathcal{Z}^{out}$. This set is closed under composition by Theorem 5.3.3. Unfortunately, it is not closed under refinement: the toggle, for instance, belongs to this set and it implements an undetermined selector, which does not belong to the set. A conclusive argument for the **decision-wait** is still lacking. All we can say is that serious attempts to construct it from the other four have failed (give it a try on a rainy day).

The arbiter and sequencer are "expensive" building blocks, which should be avoided whenever possible. Unfortunately, some finite-state specifications require an arbiter for their realization with the set of building blocks mentioned above, even though no arbitration seems to be involved. Here is an example.

5.4.1 Example The **one-all**, nicknamed O'Neall, has inputs $\{a, b\}$, outputs $\{c, d\}$, and a trace set given by the DI state graph of Figure 5.19. The c-output is produced after one input, and the d-output after all inputs have been received. Compare this to process Q of Example 5.3.1.

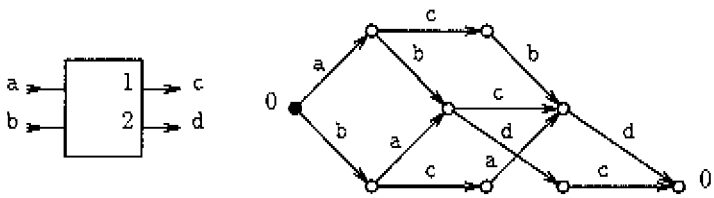


Figure 5.19: Diagram and DI state graph of one-all

Note that it satisfies Rules $\mathcal{Y}'$ and $\mathcal{Z}'$. No realization (that also takes "obvious progress requirements" into account) is known for the one-all in terms of the building-blocks presented so far avoiding arbiter and sequencer. The one-all is easy to implement with a sequencer (but that seems overkill). Maybe it should be added to the set of building blocks. ∎

The three building blocks introduced in this section—namely the undetermined selector, arbiter, and sequencer—all satisfy Rule $\mathcal{Y}'$ (but not Rule $\mathcal{Z}^{out}$, of course). The next example again proves that $\mathcal{Y}'$ is not preserved under composition but now by using only building blocks, and no ad hoc processes as in Examples 4.7.6 and 5.3.1.

5.4.2 Example Figure 5.20 shows on the left the diagram of a sequencer whose outputs are merged into a single output. The state graph of the composite of this system is shown on the right. The reader is urged to verify this.

The composite does not satisfy Rule $\mathcal{Y}'$ since states 1, 2, and 3 are distinct. More particularly, consider the state reached by trace an. In this state both input b is acceptable and output c is enabled. According to Rule $\mathcal{Z}$, they can then occur in either order. However, the order cb leads, via state 0, to state 2, and the order bc leads to state 3. In

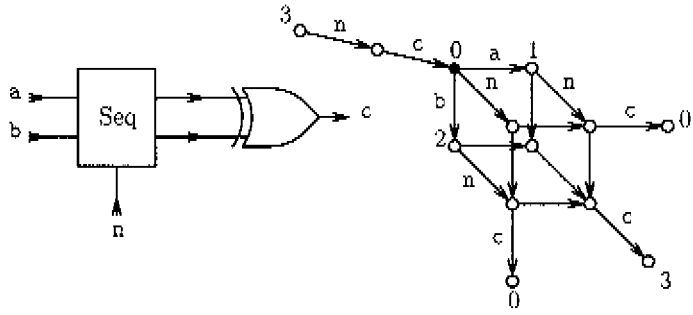


Figure 5.20: Diagram and state graph for sequencer with merged outputs

state 2, input a is acceptable, which is not the case in state 3. Thus, Rule $\mathcal{Y}'$ is violated. On account of symmetry, a similar situation occurs at the state reached by trace bn. ◢

5.5 Limitations

The DI Model gives a precise meaning to such notions as process specification, composition, and satisfaction. However, in many ways the model is not suited for solving realistic design problems. For one thing, it may be inconvenient to specify systems by means of trace sets (though state graphs and regular expressions alleviate the inconvenience to some extent). For another thing, it is often cumbersome to do composition and to verify satisfaction (though one's ability does improve with practice). But there are worse shortcomings, having to do with expressiveness.

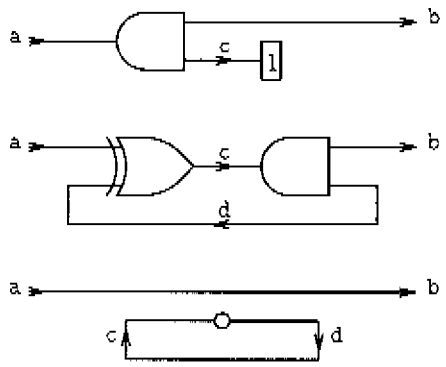


Figure 5.21: Systems D (at the top), L (in the middle), and M (at the bottom)

5.5.1 Example Consider systems D, L, and M shown in Figure 5.21. Each has an external input a and an external output b. According to the DI Model the composites of

these systems equal wire W(a; b). However, systems D and L are "unreliable" wires, in the sense that they may "break down" at any moment, that is, fail to make progress, though the specification does not say when. Both may also behave correctly. System D can stop altogether (deadlock) and system L may get into an infinite loop (livelock).

System M is a special case, for which one can argue both ways. The looped-back I-wire behaves independently of the a-b-wire and, thus, one would say there is no livelock. On the other hand, the looped-back I-wire consumes energy at an unknown rate and it may drain the batteries in no time. The DI Model does not take these considerations into account. ◢

5.5.2 Example Consider process $P = (\{\mathtt{a}\}, \{\mathtt{b}\}, \{\varepsilon, \mathtt{a}\})$. It has input a, output b, and consumes a single input without producing output. In the DI Model, P implements a wire, that is, we have

$$\{P\} \quad sat \quad \{\ \mathtt{W(a;b)}\ \}\ ,$$

because $\mathtt{W(a;b)} \in \mathcal{DI}$ and $Correct.\{P, \sim\mathtt{W(a;b)}\}$ (see Theorem 4.7.9). Process P is even worse as a wire implementation than systems D and L of the preceding example, since it cannot even behave correctly. However, as far as interference is concerned it is no worse than the wire. In fact, it is better in that respect, just because it produces no output. ◢

In general, each process P has a (best) implementation, namely $(\mathbf{i}P, \mathbf{o}P, (\mathbf{i}P)^*)$, which is also known as the **do-nothing-wrong process**. This is obviously not acceptable. Sometimes it is not so clear that an implementation is unacceptable.

5.5.3 Example Tangram (see [vB93]) has a sequential and a parallel operator. The handshake processes used for the translation of these operators are named *SEQ* and *PAR* respectively. Both have inputs $\{\mathtt{a}_0, \mathtt{b}_1, \mathtt{c}_1\}$ and outputs $\{\mathtt{a}_1, \mathtt{b}_0, \mathtt{c}_0\}$. The trace set of *SEQ* is generated by the regular expression $(\mathtt{a}_0\mathtt{b}_0\mathtt{b}_1\mathtt{c}_0\mathtt{c}_1\mathtt{a}_1)^*$, and that of *PAR* by $(\mathtt{a}_0(\mathtt{b}_0\mathtt{b}_1, \mathtt{c}_0\mathtt{c}_1)\mathtt{a}_1)^*$. The a-ports signal initiation and completion of the operator and the b- and c-ports signal initiation and completion of its left and right operands.

SEQ and *PAR* are easily implemented with building blocks:

$$\{\ \mathtt{W(a_0;b_0)},\ \mathtt{W(b_1;c_0)},\ \mathtt{W(c_1;a_1)}\ \} \quad sat \quad \{SEQ\}\ ,$$
$$\{\ \mathtt{F(a_0;b_0,c_0)},\ \mathtt{C(b_1,c_1;a_1)}\ \} \quad sat \quad \{PAR\}\ .$$

However, in the DI Model we also have $\{SEQ\}$ *sat* $\{PAR\}$. This is not acceptable, unless the specification for *PAR* is indeed intended to allow the implementer the freedom to choose an order for the b- and c-cycles. The reason that *SEQ* is not acceptable as an implementation of *PAR* is that the operands of the parallel operator might communicate on a channel and thus they would deadlock if the operator's implementation would insist that the left operand terminates successfully before the right operand is started. ◢

Finally, we give a an example that shows an anomaly of the Factorization Theorem, which is again attributable to the lack of progress as a correctness concern.

5.5.4 Example Consider process R with inputs $\{\mathtt{a}, \mathtt{b}\}$ and outputs $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, and a trace set generated by the regular expression $(\mathtt{axbz} + \mathtt{by})^*$. We wish to design R in terms of the building blocks. Let us attempt a design involving fork $F = \mathtt{F(a; x, c)}$, where $\mathtt{c}$ is some internal symbol.

According to the Factorization Theorem (Theorem 4.9.1), the remainder, say P, is now specified by $\backsim(F \parallel \backsim R)$. Process P has inputs $\{\mathtt{b}, \mathtt{c}\}$ and outputs $\{\mathtt{y}, \mathtt{z}\}$. Its trace set turns out to be generated by the regular expression $((\mathtt{b}, \mathtt{c})\mathtt{z})^*$. Note that it contains no $\mathtt{y}$. In particular, once P has received input $\mathtt{b}$ but not $\mathtt{c}$, it must not produce any output. If input $\mathtt{c}$ does occur, then R should output just $\mathtt{z}$ and, otherwise, just $\mathtt{y}$. Since P cannot "know" whether $\mathtt{c}$ will ever arrive, it cannot safely produce any output at all after receiving just input b.

Process P can, for instance, be implemented by the following system of building blocks:

$$\{\ \mathtt{C(b, c; z)},\ \ \mathtt{F(d; e, y)},\ \ \mathtt{W(e; d)}\ \}\ .$$

The C-element takes care of output $\mathtt{z}$ and the fork with feedback wire keep output $\mathtt{y}$ quiet. Together with fork F this results in a four-building-block design for R. As far as interference is concerned there is nothing wrong with this design. If, however, we look at progress, then this design is obviously not acceptable, since it fails to produce output $\mathtt{y}$ when called for. Example 6.3.5 shows that when progress is taken into account the attempted design with fork F is bound to fail, since it yields $P = \top$.

Can it be done "right"? Yes. How R reacts to input $\mathtt{b}$ depends on whether or not input a has occurred. Therefore, R is easily designed as a state machine with two states, involving a 2×2-decision-wait. This is left as an exercise. ∎

These examples show that the DI Model presented in Chapter 4 does not deal with progress concerns. The process space is not rich enough to distinguish all relevant differences between systems. Consequently, the satisfaction relation is too weak to trust in blind faith. The DI Model will be extended in Chapter 6 to overcome most of these shortcomings.

A limitation of a different nature is that so-called **isochronic forks** cannot be modeled in the DI Model. Under isochronic operation a single signal can be transferred without delay from one process to another. However, when signals need to be duplicated, an explicit fork process is required. Such a fork has two outgoing branches whose delays are independent. Some circuit designs rely for their correctness on the assumption that the two branches of the fork have (almost) equal delays (see [vB92]). The DI Model has no counterparts for such isochronic forks. It is not hard to extend the model to incorporate isochronic forks and their kin, but we will not do so. Such an extension could be based on *sets of symbols* as atomic events instead of isolated symbols.

Chapter 6

Extended DI Model

The DI Model of Chapter 4 has some limitations as pointed out in Section 5.5. In this chapter we introduce the **Extended DI Model**, which incorporates a progress concern—besides the concern for interference.

We first extend the notion of a process and adapt the notions of system operation and correctness accordingly. From that point on, satisfaction and equivalence just follow in the familiar way. Canonical representatives are again defined in terms of a partial order on processes, giving rise to the notion of DI processes. Then we treat the characterization, composition, and factorization of these DI processes. The JTU-Rules have to be modified slightly. In Chapter 7, we introduce enhanced characteristic functions as an alternative way of describing process behavior. These functions shed new light on the partial order and JTU-Rules. Finally, Chapter 8 looks into the classification of DI processes, in particular with respect to output nondeterminism.

We will use the same notation as in Chapter 4 for related entities. Whenever it is necessary to distinguish "old" and "new" entities, we subscript entities from Chapter 4 with α and entities defined in this chapter with β.

6.1 Processes

Let Σ again be an infinite set of symbols, serving as a source for alphabets and traces.

The trace set of a process in the DI Model of Chapter 4 partitions the universe of traces into two parts: the "allowed" traces (inside the trace set) and the "disallowed" traces (those outside). To capture a progress concern we subdivide the class of allowed traces into three subclasses: "transient" traces (those corresponding to states that the process is obliged to leave by producing some output), "input-demanding" traces (those which the process is not obliged to leave, but where it demands input; that is, for which the obligation to leave the corresponding state lies with the environment, namely by supplying some input) and "indifferent" traces (those for which neither process nor environment have an obligation to proceed).

Thus, the processes in the Extended DI Model will put each trace into one of *four*

categories. In keeping with the DI Model we leave the disallowed traces implicit. Formally, a **process** P now is a quintuple $(\mathbf{i}P, \mathbf{o}P, \nabla P, \Box P, \Delta P)$ such that the following six requirements are met:

1. $\mathbf{i}P$ and $\mathbf{o}P$ are disjoint alphabets,
2. ∇P, $\Box P$, and ΔP are pairwise disjoint trace sets,
3. $\mathbf{t}P \subseteq (\mathbf{a}P)^*$ (see below for the definitions of $\mathbf{a}P$ and $\mathbf{t}P$),
4. $\mathbf{t}P$ is non-empty and prefix-closed,
5. $(\forall\, t : t \in \nabla P : (\exists\, a : a \in \mathbf{o}P : ta \in \mathbf{t}P))$,
6. $(\forall\, t : t \in \Delta P : (\exists\, a : a \in \mathbf{i}P : ta \in \mathbf{t}P))$,

where $\mathbf{a}P = \mathbf{i}P \cup \mathbf{o}P$ is the **alphabet** of P as before, and $\mathbf{t}P = \nabla P \cup \Box P \cup \Delta P$ is the **trace set** of P. Traces in ∇P are called **transient** traces, those in ΔP (input-)**demanding** traces, and those in $\Box P$ **indifferent** traces. Requirement 5 expresses that in a transient trace some output is enabled. Similarly, requirement 6 expresses that in a demanding trace some input is acceptable.

The distinction between transient, indifferent, and demanding traces will be formalized when system operation is defined. The symbolism behind ∇, $\Box$, and Δ can be memorized as follows. The transient triangle ∇ will eventually topple. The demanding delta Δ does not. The indifferent box $\Box$ has a flat base like Δ but is upside-down symmetric.

The set of all processes is denoted again by $\mathcal{PROC}$. **Reflection** is the binary operator $\backsim$ on $\mathcal{PROC}$ defined by

$$\backsim P \;=\; (\mathbf{o}P, \mathbf{i}P, \Delta P, \Box P, \nabla P)\ . \tag{6.1}$$

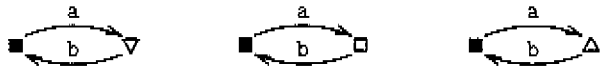
It interchanges inputs and outputs, and also transient and demanding traces, whereas the indifferent traces are invariant under reflection. $\mathcal{PROC}$ is closed under reflection.

The **after**-operator is defined as follows. For process P and trace $t \in \mathbf{t}P$, process P/t is given by

$$P/t \;=\; (\ \mathbf{i}P,\ \mathbf{o}P,\ \nabla P/t,\ \Box P/t,\ \Delta P/t\ )\ . \tag{6.2}$$

Recall that for trace set V and trace t, trace set V/t equals $\{u : tu \in V : u\}$. Observe that P/t is indeed a process because of $t \in \mathbf{t}P$; in particular, it satisfies requirements 5 and 6. Reflection and 'aftering' enjoy the properties one would expect.

In state graphs, the transient states will be labeled with ∇, the demanding states with Δ, and the indifferent states with $\Box$. Of course, a state may be labeled ∇ only if it has an outgoing output edge; similarly, Δ requires an outgoing input edge. We hold on to the convention to render initial states solidly filled. The definition of the minimal state graph of a process is directly taken from the DI Model.



Reliable wire P Unreliable wire Q Desiring I-wire R

Figure 6.1: Labeled state graphs of three wires

6.1.1 Example Wire processes P and Q have input a and output b. The traces of P are generated by the left-hand labeled state graph of Figure 6.1. P does not care about receiving input (indicated by $\Box$ in the initial state), but once input has arrived it is obliged to respond with output (indicated by ∇ in the other state). It is a reliable wire.

The traces of process Q are generated by the middle labeled state graph of Figure 6.1. Q does not care about input either, and after each input, it may respond with output but it might also not respond (indicated by $\Box$ in the state after receiving input). It is an unreliable wire. The composite of system D in Example 5.5.1 is better described by Q.

The state graph on the right-hand side of Figure 6.1 belongs to input-demanding unreliable I-wire R with output a and input b. It may, but need not, produce output (indicated by $\Box$ in the initial state), and if it does produce output then it insists on input (indicated by Δ in the other state). R is the reflection of the reliable wire P on the left, and it will turn out to be the severest test that P passes. ◢

6.1.2 Note The main reason for distinguishing input-demanding traces from indifferent traces, is that testing environments will be taken from the same process space. By being input-demanding, a test can distinguish between a process that is *guaranteed* to send output and a process that just *may* send output or not. This will be clarified in the next section where system operation is defined. A related argument is that the Factorization Theorem requires a process space that is closed under reflection. Because the expression $\backsim(Q \,\|\, \backsim R)$ appears in the Factorization Theorem, we also see that there will be a need for combining transient and input-demanding traces in a single process.

Input-demanding states are also useful for another purpose. Recall the two-input-two-output arbiter $\mathtt{A}(\mathtt{a},\mathtt{b};\mathtt{c},\mathtt{d})$, which takes alternating request–release signals on its inputs $\{\mathtt{a},\mathtt{b}\}$ and produces alternating grant–acknowledge signals on its outputs $\{\mathtt{c},\mathtt{d}\}$. If such an arbiter is used to provide mutually exclusive access to a resource, then it might be a good idea to specify the arbiter with a *"desire"* for release signals, in order to express that the resource should be released eventually.

Furthermore, one might wonder whether it makes sense to have states labeled both ∇ and Δ. Such a state, of course, should have both outgoing input and output edges. It would express that the process is guaranteed to produce output and that the environment is obliged to supply input. However, this is equivalent to labeling the state with ∇ only. The input "desire" never gives rise to a deadlock, since the state will be left by the process anyway. Again, this will become clearer once system operation has been defined. ◢

There is a "natural" mapping, say ψ, from $\mathcal{PROC}_\beta$ (the process space of this chapter) to $\mathcal{PROC}_\alpha$ (the process space of Chapter 4) that abstracts from progress aspects. It is

defined by

$$\psi.P \;=\; (\mathbf{i}_\beta P, \mathbf{o}_\beta P, \mathbf{t}_\beta P) \;. \tag{6.3}$$

A mapping from $\mathcal{PROC}_\alpha$ to $\mathcal{PROC}_\beta$ is called an **embedding**. Embedding φ is said to be **trace-set preserving** when

$$(\forall\, P : P \in \mathcal{PROC}_\alpha : \psi.(\varphi.P) = P) \;. \tag{6.4}$$

There exist many trace-set-preserving embeddings. We discuss two of them here. The first such embedding $\varphi_\sqcap$ is defined for $P \in \mathcal{PROC}_\alpha$ by

$$\varphi_\sqcap.P \;=\; (\mathbf{i}_\alpha P, \mathbf{o}_\alpha P, \varnothing, \mathbf{t}_\alpha P, \varnothing) \;, \tag{6.5}$$

where the right-hand side is trivially in $\mathcal{PROC}_\beta$. Process $Q \in \mathcal{PROC}_\beta$ with $\nabla Q = \varnothing$ is called **minimally transient**. Process Q with no demanding traces, that is with $\Delta Q = \varnothing$, is said to be **minimally demanding**. A process Q is both minimally transient and minimally demanding if and only if $\Box Q = \mathbf{t}Q$. Such a process is called **maximally indifferent** or simply **indifferent**. Embedding $\varphi_\Box$ is uniquely determined by the requirements that it be trace-set preserving and that its images be maximally indifferent processes. It is briefly referred to as the **indifferent embedding**.

The indifferent embedding is mainly interesting for theoretical purposes, since it preserves most aspects. For instance, it preserves reflection, in the sense that

$$\varphi_\Box.(\backsim P) \;=\; \backsim \varphi_\Box.P \;. \tag{6.6}$$

Since $\varphi_\Box$ is trace-set preserving, ψ is a left inverse of $\varphi_\Box$, that is $(\psi \circ \varphi_\Box).P = P$. Therefore, the pair $(\varphi_\Box, \psi)$ is a **retraction** (see [HS86]).

The second embedding φ_∇ is defined by

$$\varphi_\nabla.P \;=\; (\mathbf{i}_\alpha P, \mathbf{o}_\alpha P, V, W, \varnothing) \;, \tag{6.7}$$

where $V = \{t, a : a \in \mathbf{o}_\alpha P \wedge ta \in \mathbf{t}_\alpha P : t\}$ and $W = \mathbf{t}_\alpha P \setminus V$. V consists of all traces for which some output is enabled and, hence, V cannot be extended without violating process requirement 5. It is easily verified that indeed $\varphi_\nabla.P \in \mathcal{PROC}_\beta$. Process $Q \in \mathcal{PROC}_\beta$ with

$$\nabla Q \;=\; \{t, a : a \in \mathbf{o}Q \wedge ta \in \mathbf{t}Q : t\} \tag{6.8}$$

is called **maximally transient**. A process that is both minimally demanding and maximally transient is said to be **progressive**. This second embedding is the unique trace-set-preserving embedding that maps onto progressive processes. It is briefly referred to as the **progressive embedding**. The pair (φ_∇, ψ) is also a retraction.

We now carry over all building blocks introduced so far, by the *progressive embedding*. Note that this embedding does not preserve reflection. For instance, the (progressive embeddings of) $\mathtt{W}(a;b)$ and $\mathtt{I}(a;b)$ are no longer each other's reflection.

Here is another example to illustrate the increase in expressive power of the Extended DI Model. This example also explains how processes that are neither minimally nor maximally transient may arise

6.1.3 Example Figure 6.2 depicts the labeled state graphs of processes P_0 through P_3, each with input a, output b, and a trace set generated by the regular expression a, b. The four processes are minimally demanding, but they differ with respect to their transient and indifferent traces.

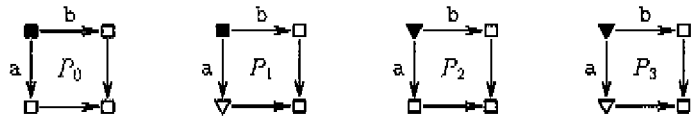


Figure 6.2: Labeled state graphs of P_0 through P_3 with input a and output b

Each of the four processes can accept input a and produce output b independently (the occurrence of a and b is not "coupled" as far as interference is concerned). P_0 is minimally transient (output b might occur but then again it might not occur) and P_3 is maximally transient (output b is guaranteed to occur, no matter what). P_1 and P_2 are neither minimally nor maximally transient. In P_1, occurrence of input a will guarantee the—otherwise unreliable—output b. In P_2, however, occurrence of input a may jeopardize the otherwise reliable—output b.

It is not difficult to envisage how P_0 and P_3 arise as composites of systems built from (progressive) building blocks, but this is much less straightforward for P_1 and P_2. Why would such in-between processes as P_1 and P_2 be needed?

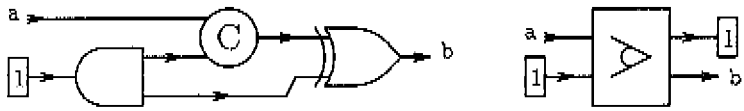


Figure 6.3: Systems S_1 (left) and S_2 (right)

Figure 6.3 shows systems S_1 and S_2, whose composites are claimed to be P_1 and P_2 respectively. These systems consist of progressive building blocks (the two kinds of boxes labeled 1 are processes that either accept up their input once, or are guaranteed to produce their output once). That the composites of these systems are indeed P_1 and P_2 cannot yet be verified, but with the help of some intuition one should at least get the idea.

For S_1, it is clear that input a will guarantee output b, whereas b is not guaranteed without a because a signal may get "stuck" at the C-element. In S_2, when no a-input is offered, the arbiter will grant the internal request, thus guaranteeing output b. However, when the environment supplies input a, there is a "race" between two requests at the arbiter; which request gets granted is now undetermined. ∎

In the light of the preceding example, it appears that the set of all progressive processes is not closed under composition. In contrast to this, the set of all maximally indifferent processes is closed. We will come back to this in Chapter 8 (in particular, in Theorem 8.1.8).

6.2 Operation and Correctness of Systems

The structure of systems is the same as before, though now systems are constructed from new processes. The operation of systems is refined as follows.

First we deal with isochronic operation. Let S be a closed system. The reachable traces $reach.S$ and interfering traces $intf.S$ of S are defined as before, and also the statement 'S is free of interference' retains its earlier definition.

Each non-interfering trace of S is put into one of three, pairwise disjoint, sets ∇S, $\Box S$, or ΔS, according to the following definition. For trace $t \in reach.S \setminus intf.S$ we postulate

$$\begin{aligned} t \in \nabla S &\equiv (\exists P : P \in S : t\lceil \mathtt{a} P \in \nabla P) , \\ t \in \Box S &\equiv (\forall P : P \in S : t\lceil \mathtt{a} P \in \Box P) , \\ t \in \Delta S &\equiv (\forall P : P \in S : t\lceil \mathtt{a} P \notin \nabla P) \wedge (\exists P : P \in S : t\lceil \mathtt{a} P \in \Delta P) . \end{aligned}$$

To paraphrase, if there exists at least one process $P \in S$ such that $t\lceil \mathtt{a} P \in \nabla P$, then trace t is transient in S. Otherwise, if there is no such P and there exists at least one process $Q \in S$ such that $t\lceil \mathtt{a} Q \in \Delta Q$, then t is demanding in S. Otherwise, if there are no such P or Q, that is, if $t\lceil \mathtt{a} R \in \Box R$ for each process $R \in S$, then t is indifferent in S.

System S is called **free of deadlock** when

$$\Delta S = \varnothing , \tag{6.9}$$

that is, when S has no demanding traces. Since S is closed, the presence of a demanding trace constitutes a potential deadlock, because no process will see to it that this state does not persist and there is at least one process that insists on progress. Keep in mind that there is no (implicit) environment that might keep the system going.

6.2.1 Example Consider closed system $S = \{P, Q, R\}$ where

$$\begin{aligned} P &= (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\varepsilon\}, \{\mathtt{a}, \mathtt{b}\}, \varnothing) , \\ Q &= (\{\mathtt{a}\}, \varnothing, \varnothing, \{\mathtt{a}\}, \{\varepsilon\}) , \\ R &= (\{\mathtt{b}\}, \varnothing, \varnothing, \{\mathtt{b}\}, \{\varepsilon\}) . \end{aligned}$$

Process P will produce either output $\mathtt{a}$ or output $\mathtt{b}$, after which it is indifferent. Process Q insists on receiving input $\mathtt{a}$, after which it also is indifferent. Similarly, R insists on receiving $\mathtt{b}$, after which it becomes indifferent. Compare this to system S_7 *par* $\{P_7\}$ of Example 3.2.7.

As one may readily verify, we have for S:

$$\begin{aligned} reach.S &= weave.S = \{\varepsilon, \mathtt{a}, \mathtt{b}\} , \\ \nabla S &= \{\varepsilon\} , \\ \Box S &= \varnothing , \\ \Delta S &= \{\mathtt{a}, \mathtt{b}\} . \end{aligned}$$

Consequently, system S is free of interference but not free of deadlock. Both Q and R desire input, but P satisfies only one such desire. ∎

Anisochronic operation of S is again defined as isochronic operation of $\vec{S}$ (the wired version of S). The extra wires introduced in $\vec{S}$ are taken to be progressive, that is, maximally transient and minimally demanding.

Freedom of deadlock is imposed as an additional correctness concern. That is, we define $Correct.S$ by

$$Correct.S \quad \equiv \quad \text{'}\vec{S} \text{ is closed, and free of interference and deadlock'} \ .$$

Note that correctness is (again) based on *anisochronic* operation. Satisfaction and equivalence are defined as before, involving the modified notions of process and correctness.

6.2.2 Example Wires P and Q of Example 6.1.1 are not equivalent in the Extended DI Model, because P passes tests that Q fails. One such test is input-demanding I-wire R defined in the same example. We have

$$Correct.\{P,R\} \quad \wedge \quad \neg Correct.\{Q,R\} \ .$$

The latter holds because unreliable wire Q deadlocks with R: R insists on input after a ($\mathtt{a}! \in \Delta\rho_R.R$), which Q need not provide ($\mathtt{a}? \in \Box\rho_Q.Q$) and, therefore, we have $\mathtt{a}!\mathtt{a}? \in \Delta\{Q,R\}^{\neg}$. In fact, this shows $\neg(Q \ sat \ P)$. We do have $P \ sat \ Q$. ∎

6.2.3 Example For processes SEQ and PAR from Example 5.5.3 we have $\{SEQ\} \ sat_\alpha \ \{PAR\}$ in the DI Model. For the progressive embeddings of SEQ and PAR we no longer have $\{SEQ\} \ sat_\beta \ \{PAR\}$ in the Extended DI Model. This is corroborated by testing environment U with

$$U \quad = \quad \{ \ \backsim\mathtt{W}(\mathtt{a}_0;\mathtt{a}_1), \ \ \mathtt{C}(\mathtt{b}_0,\mathtt{c}_0;\mathtt{x}), \ \ \mathtt{F}(\mathtt{x};\mathtt{b}_1,\mathtt{c}_1) \ \} \ .$$

Note that the reflected wire is a *demanding* I-wire. We have for this test

$$\neg Correct.(\{SEQ\} \parallel U) \quad \wedge \quad Correct.(\{PAR\} \parallel U) \ ,$$

because SEQ deadlocks with U, whereas PAR does not.

If one wishes to express that the implementer of PAR has the freedom to order the $\mathtt{b}$- and $\mathtt{c}$-cycles, then the four states of PAR reached by $\mathtt{a}_0\mathtt{b}_0$, $\mathtt{a}_0\mathtt{b}_0\mathtt{b}_1$, $\mathtt{a}_0\mathtt{c}_0$, and $\mathtt{a}_0\mathtt{c}_0\mathtt{c}_1$ should be made indifferent instead of transient. ∎

Let us have a quick look at the relationship between the concepts of this section and their relatives in the DI Model of Chapter 4. An abstraction or embedding of one process space in another is lifted to system spaces by elementwise application. It is easy to verify the following statements for appropriate systems S:

$$\begin{array}{lcl} weave_\beta.S & = & weave_\alpha.(\psi.S) \ , \\ Correct_\beta.S & \Rightarrow & Correct_\alpha.(\psi.S) \ , \\ \Delta(\varphi.S) & = & \varnothing \ , \\ Correct_\alpha.S & \equiv & Correct_\beta.(\varphi.S) \ , \end{array}$$

where φ is either the progressive or the indifferent embedding. The latter two equalities hold because for all processes $P \in \mathcal{PROC}_\beta$ involved, we have $\Delta P = \varnothing$. Thus, it seems as if we have not gained much by extending the model.

When considering satisfaction, the picture changes drastically. For systems S and T in $\mathcal{SYS}_\alpha$ and the indifferent embedding $\varphi_\Box$, we have

$$S \; sat_\alpha \; T \quad \equiv \quad \varphi_\Box.S \; sat_\beta \; \varphi_\Box.T \; . \tag{6.10}$$

Note that the tests involved in sat_β range over $\mathcal{SYS}_\beta$ and not just over $\varphi_\Box.\mathcal{SYS}_\alpha$. Since only indifferent processes are compared, tests R with $\Delta R \neq \varnothing$ do not affect the outcome.

Consequently, we have that

$$\langle \mathcal{SYS}_\alpha; par_\alpha, sat_\alpha \rangle \;\; \text{is isomorphic to} \;\; \langle \varphi_\Box.\mathcal{SYS}_\alpha; par_\beta, sat_\beta \rangle \; . \tag{6.11}$$

This shows that we can find the old model as a submodel of the extended model, namely through the indifferent embedding. The Extended DI Model is indeed an extension of the DI Model, although the indifferent view is not what we had in mind with, for instance, the building blocks.

Equation 6.10, however, does not hold when we take the progressive embedding $\varphi_\triangledown$ instead of $\varphi_\Box$. We do have

$$S \; sat_\alpha \; T \quad \Leftarrow \quad \varphi_\triangledown.S \; sat_\beta \; \varphi_\triangledown.T \; , \tag{6.12}$$

but the implication from left to right is, in general, not valid. Here, tests R with $\Delta R \neq \varnothing$ play a crucial role. A counterexample to the reverse implication of (6.12) is provided by Example 5.5.3 above.

Through the progressive embedding, the Extended DI Model can be viewed as a *"non-conservative"* extension of the DI Model: the progressive embedding provides a more appropriate interpretation for the building blocks, which – by necessity– does not preserve such aspects as the *sat* relation.

6.2.4 Note System D in Example 5.5.1 and process P in Example 5.5.2 also fail test $R = \backsim\varphi_\triangledown.\mathtt{W(a;b)}$ and, hence, they are not good wire implementations. In the Extended DI Model the anomaly has disappeared. Systems L and M in Example 5.5.1, however, pass test R, because they never fail to do something (internally). In fact, they are still equivalent to $\mathtt{W(a;b)}$. This may be considered an anomaly, which falls outside the scope of our work.

In order to deal with livelock one might treat internal symbols more explicitly and introduce infinite traces. Again three kinds can be distinguished: those traces that the system is obliged to avoid, those the environment is obliged to avoid, and those that need not be avoided. Livelock is characterized by a reachable infinite trace that should have been avoided. For example, a "fair" undetermined selector $\mathtt{U(a;b,c)}$ maps infinite traces of the form $t(ab)^\omega$ and $t(ac)^\omega$, where $ta \in reach.\mathtt{U(a;b,c)}$, to $\top$.

Not every such mapping is an acceptable process description. For instance, the progressive wire $\mathtt{W(a;b)}$ may not map $(ab)^\omega$ to $\top$ because that constitutes a contradictory requirement, which cannot be met. See [Ros88] for some of the subtleties that may emerge in such an approach to liveness. ∎

6.3 Canonical Representatives

The Extended DI Model gives rise to a new notion of canonical representatives and DI processes. We begin again with partial order $\sqsubseteq$ on $\mathcal{PROC}$. At this point, the definition of $\sqsubseteq$ may seem to fall from the sky, but in Chapter 7 we show how it can be derived from the definition of system operation.

For processes P and Q we define $P \sqsubseteq Q$ by

$$\begin{aligned} P \sqsubseteq Q \quad \equiv \quad & \mathbf{i}P = \mathbf{i}Q \wedge \mathbf{o}P = \mathbf{o}Q \wedge \\ & (\forall\, t, a : a \in \mathbf{i}P \wedge ta \in \mathbf{t}P \wedge t \in \mathbf{t}Q : ta \in \mathbf{t}Q) \wedge \\ & (\forall\, t, a : a \in \mathbf{o}P \wedge t \in \mathbf{t}P \wedge ta \in \mathbf{t}Q : ta \in \mathbf{t}P) \wedge \\ & (\forall\, t : t \in \nabla P \wedge t \in \mathbf{t}Q : t \in \nabla Q) \wedge \\ & (\forall\, t : t \in \mathbf{t}P \wedge t \in \Delta Q : t \in \Delta P) \end{aligned} \tag{6.13}$$

The first four conjuncts are taken from $\sqsubseteq_a$. New are the last two conjuncts, which can be rephrased as follows.

- For all states $t \in \mathbf{t}P \cap \mathbf{t}Q$, if P is transient in t, then so is Q ("P is no more transient than Q").
- For all states $t \in \mathbf{t}P \cap \mathbf{t}Q$, if Q is demanding in t, then so is P ("P is at least as demanding as Q").

Relation $\sqsubseteq$ is a partial order on $\mathcal{PROC}$. In Chapter 7, we look at some technical aspects of this order.

6.3.1 Example For processes P_0 through P_3 of Example 6.1.3, we have $P_0 \sqsubseteq P_1 \sqsubseteq P_3$ and $P_0 \sqsubseteq P_2 \sqsubseteq P_3$. Processes P_1 and P_2 are incomparable. ∎

To make process P smaller with respect to $\sqsubseteq$, one should not change its input and output alphabets, but one may (i) increase its output capability, (ii) decrease its input willingness, (iii) make it less transient (transfer a trace from ∇P to $\Box p$), (iv) make it more demanding (transfer a trace from $\Box P$ to ΔP), or any combination of the preceding. Methods (ii) and (iv) are conflicting, since a process must be willing to receive at least one input in a demanding state. Hence, there exists no least process in the set of processes with fixed input and output alphabet. This contrasts with the DI Model, where such processes do exist. Because reflection turns the order around, a similar situation holds for greatest processes.

6.3.2 Example Consider processes P, Q, and R defined by

$$\begin{aligned} P &= (\{\mathtt{a}\}, \varnothing, \varnothing, \{\varepsilon, \mathtt{a}\}, \varnothing)\ , \\ Q &= (\{\mathtt{a}\}, \varnothing, \varnothing, \{\mathtt{a}\}, \{\varepsilon\})\ , \\ R &= (\{\mathtt{a}\}, \varnothing, \varnothing, \{\varepsilon\}, \varnothing)\ . \end{aligned}$$

These processes have only one port, namely input a. P accepts one a but does not desire it, Q insists on one a, and R does not accept input at all. R is obtained from P by method (ii) above, and Q by method (iv). Thus we have $Q \sqsubset P$ and $R \sqsubset P$. Both Q and R are $\sqsubseteq$-minimal processes, and their set of common lower bounds is empty. ∎

The preceding example also shows that $\mathcal{PROC}(I, O)$, the set of processes with inputs I and outputs O, is not a complete lattice (only in the trivial case $I = O = \varnothing$). As before, we add the two "virtual" processes $\bot$ (as $\sqsubseteq$-least process) and $\top$ (as $\sqsubseteq$-greatest process) to force completeness. For instance, the greatest lower bound of Q and R in Example 6.3.2 above is $\bot$.

This time it is more complicated to verify that the thus expanded set of processes, which we denote by $\mathcal{PROC}$ again, is indeed a complete lattice. We will come back to this in Chapter 7. Nevertheless, everything proceeds as before. We will not repeat all definitions and theorems. The definitions of *pass*, *Friends*, $[\![\text{-}]\!]$, $\|$, and $\mathcal{DI}$ carry over in a straightforward way. The JTU-Rules, however, have to be adapted. We do so in the next section. First we look at three composition examples.

6.3.3 Example We reconsider the composition appearing in Example 4.7.6. The progressive embeddings of P and Q are presented on the left in Figure 6.4. On the right is

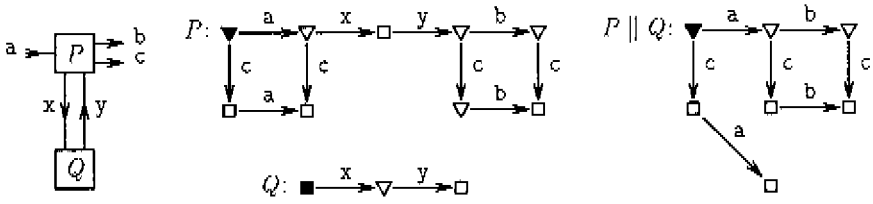


Figure 6.4: Labeled state graphs of P, Q, and their composite $P \| Q$

depicted the composite $P \| Q$. It can be obtained in a manner similar to that explained in Section 4.8 for the DI Model. In the Extended DI Model, however, there are two additional ways for reducing a process with respect to $\sqsubseteq$. The first way is to make it less transient (by changing a ∇-label to a $\square$-label in its state graph). The second way is to make it more demanding (by changing a $\square$-label to a Δ-label; but this requires that the affected state has an input edge).

It should be noted that the composite is not maximally transient, since the state reached by ac is labeled $\square$ and not ∇. When it is changed to ∇, the reflection is no longer a friend, because, for instance, trace a!c!a?c? is then demanding. ◢

The preceding example shows that the progressive embedding does not preserve $\mathcal{DI}$, that is, in general, we do *not* have $P \in \mathcal{DI}_\alpha \Rightarrow \varphi_\nabla.P \in \mathcal{DI}_\beta$. Theorem 8.1.10 characterizes the conditions under which the implication *is* valid.

6.3.4 Example Let us study system S whose diagram appears on the left in Figure 6.5. The composite of S is shown on the right. If the environment waits for output c (which will come) and after receiving c sends input a, then output b is guaranteed. However, if the environment starts with sending input a, then there is a race between two requests and output b is still guaranteed but output c is not, though c may still appear (recall that the arbiter does not grant a second request until the first is released, which is not done

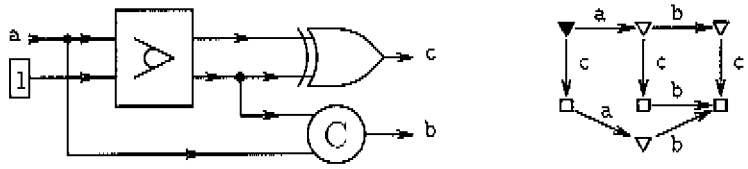


Figure 6.5: System S (left) and labeled state graph of its composite (right)

here). Thus the states reached by ca and ac are not the same (their labels differ). As far as "future" traces are concerned, these states are equivalent, that is, in the DI Model they would be the same state. Also note that $[\![S]\!]_\beta$ does not satisfy Rule $\mathcal{Y}'_\beta$, whereas the abstraction $P = \psi.[\![S]\!]_\beta$ does satisfy Rule $\mathcal{Y}'_\alpha$ because $P/\mathtt{ac} = P/\mathtt{ca}$. ◢

6.3.5 Example As promised, we consider again the design problem put forward in Example 5.5.4, but now in the context of the Extended DI Model. We attach the progressive interpretation to specification R.

Let us again attempt the design with (progressive) fork $F = \mathsf{F}(\mathtt{a}; \mathtt{x}, \mathtt{c})$. On account of the Factorization Theorem, which is again valid, the remainder is specified by $P = \backsim(F \,\|\, \backsim R)$. Figure 6.6 shows the labeled (DI) state graphs of $\backsim R$ and F. We are interested in the least friend of system $\{\backsim R, F\}$. It is a process with inputs $\{\mathtt{b}, \mathtt{c}\}$ and outputs $\{\mathtt{y}, \mathtt{z}\}$.

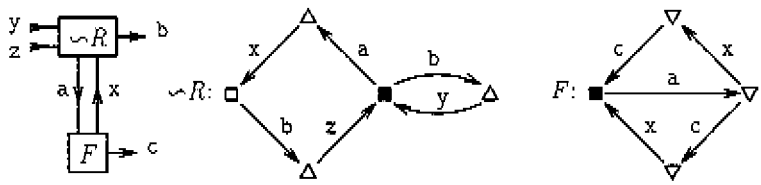


Figure 6.6: State graphs of $\backsim R$ and F

In Example 5.5.4 we figured out that—within the DI Model—the trace set of P is generated by $((\mathtt{b}, \mathtt{c})\mathtt{z})^*$. In the Extended DI Model, however, even the progressive version of this process is not a friend of $\{\backsim R, F\}$, because after output b there is still a demand for input in $\backsim R$. A friend of $\{\backsim R, F\}$ must be willing to receive input b, and after that it must produce some output to meet the input desire of $\backsim R$. However, it is impossible to know at this point which of the two outputs y or z may safely be produced. Therefore, only process $\top$ is a friend, and the design equation $P: P \,\|\, F \sqsupseteq R$ has $\top$ as only solution, indicating that there are no "satisfactory" solutions (for a design of R using fork F). ◢

The last example shows that for $S \in \mathcal{SYS}_\beta$, in general, we do *not* have $\psi.[\![S]\!]_\beta = [\![\psi.S]\!]_\alpha$. However, the indifferent embedding $\varphi_\square$ preserves composites, that is, for $S \in \mathcal{SYS}_\alpha$ we have $\varphi_\square.[\![S]\!]_\alpha = [\![\varphi_\square.S]\!]_\beta$.

6.4 Extended JTU-Rules

Characterization Theorem 4.7.3 also holds in the Extended DI Model provided that the JTU-Rules are properly extended as well. In particular, we have $P \in \mathcal{DI}$ if and only if P satisfies the extended JTU-Rules. Here they are.

- P satisfies **Rule** $\mathcal{W}$ when for all traces s and symbols a we have

$$saa \notin \mathbf{t}P \ .$$

- P satisfies **Rule** $\mathcal{X}$ when for all traces s and t, and symbols a and b with $a \eqsim b$ we have

$$\begin{aligned} sabt \in \mathbf{t}P &\equiv sbat \in \mathbf{t}P \ , \\ sabt \in \nabla P &\equiv sbat \in \nabla P \ , \\ sabt \in \Delta P &\equiv sbat \in \Delta P \ . \end{aligned}$$

- P satisfies **Rule** $\mathcal{Y}$ when for all traces s and t, and symbols a, b, and c with $a \not\eqsim c$ and $b \eqsim a$ we have

$$\begin{aligned} &sactb \in \mathbf{t}P \wedge scat \in \mathbf{t}P \ \Rightarrow \ scatb \in \mathbf{t}P \ , \\ &scat \in \nabla P \wedge sact \in \mathbf{t}P \ \Rightarrow \ sact \in \nabla P \quad \text{if } a \subset \mathbf{o}P \wedge c \in \mathbf{i}P \ , \\ &scat \in \Delta P \wedge sact \in \mathbf{t}P \ \Rightarrow \ sact \in \Delta P \quad \text{if } a \in \mathbf{i}P \wedge c \subset \mathbf{o}P \ . \end{aligned}$$

- P satisfies **Rule** $\mathcal{Z}$ when for all traces s, and symbols a and c with $a \not\eqsim c$ we have

$$sa \in \mathbf{t}P \wedge sc \in \mathbf{t}P \ \Rightarrow \ sac \in \mathbf{t}P \wedge sca \in \mathbf{t}P \ .$$

The formulation of Rules $\mathcal{W}$ and $\mathcal{Z}$ has not changed. Rules $\mathcal{X}$ and $\mathcal{Y}$ have both been extended with two conditions concerning transient and demanding traces. Since Rule $\mathcal{X}$ implies

$$sabt \in \Box P \equiv sbat \in \Box P \ ,$$

it can again be interpreted as expressing that the order of signals in the same direction is irrelevant for future possibilities (where 'future possibilities' not only concern trace-set membership, but also whether it is transient, indifferent, or demanding). Under this modified interpretation of 'future possibilities', also the interpretation of Rule $\mathcal{Y}$ is maintained.

Again, often a simpler version of Rule $\mathcal{Y}$ holds:

- P satisfies **Rule** $\mathcal{Y}'$ when for all traces s and t, and symbols a and c with $a \not\eqsim c$, $sa \in \mathbf{t}P$, and $sc \in \mathbf{t}P$ we have

$$\begin{aligned} sact \in \mathbf{t}P &\equiv scat \in \mathbf{t}P \ , \\ sact \in \nabla P &\equiv scat \in \nabla P \ , \\ sact \in \Delta P &\equiv scat \in \Delta P \ . \end{aligned}$$

Note that $\mathcal{Y}'$ implies $\mathcal{Y}$. Rule $\mathcal{Y}'$ expresses that if two signals of *opposite* direction can both occur, then their order is irrelevant for future possibilities.

The composites in Examples 6.3.3 and 6.3.4 above satisfy Rule $\mathcal{Y}$ but not $\mathcal{Y}'$. In terms of the after-operator, Rules $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Y}'$ can again be rephrased in accordance with Theorem 4.7.4. We will elaborate on this in the next section.

6.4.1 Theorem (*Fundamental property of DI processes*)
For closed system S such that of each pair of connected processes at least one is in $\mathcal{DI}$, we have

'$\tilde{S}$ is free of interference and deadlock'

$\equiv$

'S is free of interference and deadlock'

Proof idea: The effects of additional wires are already incorporated in a DI process. Especially see the proof of Theorem 4.7.3 in Section 7.2. ∎

6.4.2 Example Reconsider the processes of Example 5.3.2. The initial states of P and Q can be labeled ∇ to indicate that these processes will eventually produce output when no input is supplied. Also the states reached by a? (in P) and b? (in Q) can be labeled with ∇. However, states 2 and 3 cannot be labeled ∇ (nor Δ) because they have no outgoing edges, so they must be labeled $\Box$.

In order to make the processes as transient as possible, one might be tempted to label states 1 and 4 with ∇. This, however, would not yield a DI process, that is, the result is not a canonical representative, since it fails Rule $\mathcal{Y}$. The reason is that states 2 and 3 are not labeled ∇. States 1 and 4 must be labeled $\Box$ (again Δ is impossible). As a consequence, the initial state of composite $P \parallel Q$ will be labeled $\Box$ as well and not ∇, revealing that $P \parallel Q$ may fail to do any output.

This is true even under isochronic operation, since P and Q are DI processes (see Theorem 6.4.1). When states 1 and 4 are labeled ∇, the processes are no longer DI and deadlock can only be "detected" under anisochronic operation. ∎

Chapter 7

Enhanced Characteristic Functions

The partial order $\sqsubseteq$ and also the JTU-Rules can be understood better when processes are described in a different way. This chapter introduces an alternative representation for processes. We work in the context of the Extended DI Model, though everything can be translated to the DI Model as well. In contrast to other chapters, it looks more at the details of the underlying mathematics.

As pointed out in the introduction of Chapter 6, a process partitions the universe of traces into four parts. A slightly more convenient partition—one that takes into account how a process operates in a system—splits the trace set's complement (the disallowed traces) into traces that correspond either to unreachable or to interfering states (also see [Ver89]). A trace is unreachable if it "steps" outside the trace set via an output. The *process* is disallowed (in fact, unable) to produce these traces. A trace is interfering if it "steps" outside via an input. The obligation to avoid these traces lies with the *environment*. These two classes exchange roles under reflection. A process can, thus, be viewed as attaching one of *five* labels to each trace: unreachable, transient, indifferent, demanding, or interfering. Let us develop this idea more formally.

An **enhanced characteristic function** (**ECF**) is a mapping from the set Σ^* of all traces to $\Lambda = \{\top, \nabla, \Box, \Delta, \bot\}$, where Λ is just a set of suggestive squiggles representing the five trace labels. The set of all ECFs is denoted by $\mathcal{ECF}$. Typically, variables f, g, and h range over $\mathcal{ECF}$. Observe that for every trace $t \in \mathbf{t}P$, where P is a process different from $\top$ and $\bot$, we have

$$t \notin \mathbf{t}P \;\equiv\; (\exists\, t_0, a, t_1 : t = t_0 a t_1 \wedge t_0 \in \mathbf{t}P \wedge t_0 a \notin \mathbf{t}P : a \in \mathbf{a}P)\,, \tag{7.1}$$

because $\mathbf{t}P$ is non-empty and prefix-closed. In fact, t_0, a, and t_1 on the right-hand side are uniquely determined by t when the left-hand side holds. For process $P \in \mathcal{PROC} \setminus \{\top, \bot\}$ we now define its ECF $\mathbf{f}P$ by

$$\mathbf{f}P.t \;=\; \begin{cases} \top & \text{if} \quad (\exists\, t_0, a, t_1 : t \lceil \mathbf{a}P = t_0 a t_1 \wedge t_0 \in \mathbf{t}P \wedge t_0 a \notin \mathbf{t}P : a \in \mathbf{o}P) \\ \nabla & \text{if} \quad t \lceil \mathbf{a}P \in \nabla P \\ \Box & \text{if} \quad t \lceil \mathbf{a}P \in \Box P \\ \Delta & \text{if} \quad t \lceil \mathbf{a}P \in \Delta P \\ \bot & \text{if} \quad (\exists\, t_0, a, t_1 : t \lceil \mathbf{a}P = t_0 a t_1 \wedge t_0 \in \mathbf{t}P \wedge t_0 a \notin \mathbf{t}P : a \in \mathbf{i}P) \end{cases}$$

Note the projection of t on $\mathbf{a}P$. The images ∇, $\Box$, and Δ are directly associated with the three components ∇P, $\Box P$, and ΔP respectively. The images $\top$ and $\bot$ are associated with traces that stepped outside the trace set via an output or an input respectively. The reason for defining ECFs with domain Σ^* instead of $(\mathbf{a}P)^*$ is that this simplifies the treatment of composition.

The ECFs of processes $\top$ and $\bot$ are simply defined by $\mathbf{f}\top.t = \top$ and $\mathbf{f}\bot.t = \bot$.

7.1 Composition and Correctness for Trace Labels

We first study the set Λ of trace labels a little more. Variables λ, μ, and ν range over Λ. **Composition**, denoted by $\|$, is the binary operator on Λ defined in Table 7.1. This operator forms the basis for characterizing correct system operation. The intuition is as follows. For a trace to be unreachable under the composition of two processes, it must

$\|$	$\top$	∇	$\Box$	Δ	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
∇	$\top$	∇	∇	∇	$\bot$
$\Box$	$\top$	∇	$\Box$	Δ	$\bot$
Δ	$\top$	∇	Δ	Δ	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$

Table 7.1: Composition operator $\|$ on Λ

be unreachable by at least one; thus, $\top$ prevails. For a trace to be interfered under the composition, it must be reachable for both and interfered for at least one; thus, $\bot$ prevails for non-$\top$ arguments. For a trace to be transient under composition, it must be reachable and not interfered for both, and transient for at least one. For a trace to be deadlocked, it must be reachable, not interfered, and not transient for both, and demanding for at least one. For a trace to be indifferent, it must be indifferent for both.

7.1.1 Theorem Composition operator $\|$ on Λ is commutative, associative, idempotent, and has $\Box$ as unit. Furthermore, it has $\top$ as zero and there are no zero divisors under $\|$, that is,

$$\lambda \| \mu = \top \quad \equiv \quad \lambda = \top \vee \mu = \top \,.$$

Proof Consider the order $\preceq$ on Λ defined by

$$\top \prec \bot \prec \nabla \prec \Delta \prec \Box$$

and observe that composition $\|$ corresponds to taking the minimum under this order. The binary minimum operator is commutative, associative, idempotent and has $\Box$, being the $\preceq$-greatest element, as unit, and $\top$, being the $\preceq$-least element, as zero. Finally, a minimum operator has no zero divisors. ∎

In view of its associativity, commutativity, idempotence, and unit element, we can extend $\|$ to a unary operator on sets over Λ. The ECF $\mathbf{f}S$ of system S is now defined by tracewise composition:

$$\mathbf{f}S.t \quad = \quad \|\{P : P \in S : \mathbf{f}P.t\} \ . \tag{7.2}$$

If, furthermore, we define *Correct* on Λ by

$$Correct.\lambda \quad \equiv \quad \lambda \notin \{\bot, \Delta\} \ , \tag{7.3}$$

then we have

$$Correct.S \quad \equiv \quad \langle \forall\, t : t \in \Sigma^* : Correct.(\mathbf{f}\bar{S}.t) \rangle \ . \tag{7.4}$$

This characterization of correctness abstracts from the operational view. On Λ we also define *pass*-sets and a corresponding satisfaction relation *sat* and equivalence *equ*:

$$\begin{array}{lcl} pass.\lambda & = & \{\mu : Correct.(\lambda \| \mu) : \mu\} \ , \\ \lambda \ sat \ \mu & \equiv & pass.\lambda \supseteq pass.\mu \ , \\ \lambda \ equ \ \mu & \equiv & pass.\lambda = pass.\mu \ . \end{array}$$

The *pass*-sets of Λ are tabulated on the left in Table 7.2. Notice that these *pass*-sets are

λ	$pass.\lambda$
$\top$	$\{\top,\ \nabla,\ \Box,\ \Delta,\ \bot\}$
∇	$\{\top,\ \nabla,\ \Box,\ \Delta\ \ \}$
$\Box$	$\{\top,\ \nabla,\ \Box\ \ \ \ \}$
Δ	$\{\top,\ \nabla\ \ \ \ \ \ \}$
$\bot$	$\{\top\ \ \ \ \ \ \ \ \}$

Table 7.2: The *pass*-sets for Λ and the Hasse diagram for $\sqsubseteq$ on Λ

unique, that is, $pass.\lambda = pass.\mu$ if and only if $\lambda = \mu$. Hence, relation *sat* induced by *pass* is a partial order. We also denote it by $\sqsupseteq$. The Hasse diagram of $\sqsubseteq$ is given on the right in Table 7.2. Obviously, $\langle \Lambda; \sqsubseteq \rangle$ is a complete lattice. Relation *equ* on Λ boils down to equality.

Observe that each *pass*-set of Λ has a $\sqsubseteq$-minimum. **Reflection** $\backsim$ on Λ is defined by

$$\backsim\lambda \quad = \quad \min(pass.\lambda) \tag{7.5}$$

and is tabulated in Table 7.3.

λ	$\top$	∇	$\Box$	Δ	$\bot$
$\backsim\lambda$	$\bot$	Δ	$\Box$	∇	$\top$

Table 7.3: Reflection operator $\backsim$ on Λ

Everything involving Λ has been built up from $\amalg$ and *Correct*. The partial order $\sqsubseteq$ and reflection $\backsim$ are derived concepts. From Tables 7.1, 7.2, and 7.3 we can readily infer a number of properties, such as

$$\lambda \in \mathit{pass}.\mu \;\equiv\; \lambda \sqsupseteq \backsim\mu \;, \tag{7.6}$$

$$\mathit{Correct}.\lambda \;\equiv\; \lambda \sqsupseteq \backsim\Box \;, \tag{7.7}$$

$$\backsim\backsim\lambda \;=\; \lambda \;, \tag{7.8}$$

$$\lambda \sqsubseteq \mu \;\equiv\; \backsim\lambda \sqsupseteq \backsim\mu \;, \tag{7.9}$$

$$\lambda \amalg \mu \sqsupseteq \nu \;=\; \lambda \sqsupseteq \backsim(\mu \amalg \backsim\nu) \;. \tag{7.10}$$

In fact, these properties can be proved without relying on the specific definitions of $\amalg$ and *Correct*. All that is needed are (i) the properties of $\amalg$ mentioned in Theorem 7.1.1, (ii) that each *pass*-set has a minimum, and (iii) the definitions of *pass*, $\sqsubseteq$, and $\backsim$ (see [Ver94]).

7.2 Neighbor-Swap Rule

Partial order $\sqsubseteq$ and reflection $\backsim$ on Λ can be lifted to $\mathcal{ECF}$ by tracewise application. This makes $\langle \mathcal{ECF}; \sqsubseteq \rangle$ a complete lattice as well.

Reflection on $\mathcal{PROC}$ is related to reflection on $\mathcal{ECF}$:

$$\mathbf{f}(\backsim P) \;=\; \backsim \mathbf{f} P \;. \tag{7.11}$$

Less obviously, the partial order $\sqsubseteq$ on $\mathcal{PROC}$ turns out to be related to $\sqsubseteq$ on $\mathcal{ECF}$:

$$P \sqsubseteq Q \;\equiv\; \mathbf{i}P = \mathbf{i}Q \wedge \mathbf{o}P = \mathbf{o}Q \wedge \mathbf{f}P \sqsubseteq \mathbf{f}Q \;. \tag{7.12}$$

From this equivalence it is immediately obvious that $\sqsubseteq$ is a partial order on $\mathcal{PROC}$.

7.2.1 Theorem For connectable systems S and T such that $S \mathbin{\mathit{par}} T$ is closed and DI, and $\mathbf{u}S \cap \mathbf{u}T = \varnothing$ (in which case no renaming is needed for $S \mathbin{\mathit{par}} T$), we have

$$\mathit{Correct}.(S \mathbin{\mathit{par}} T) \;\equiv\; \mathbf{f}S \sqsupseteq \backsim \mathbf{f}T \;, \tag{7.13}$$

Proof Bearing in mind the properties of the preceding section, we derive

$$\begin{array}{ll} & \mathit{Correct}.(S \mathbin{\mathit{par}} T) \\ \equiv & \quad \{ \text{ Equation 7.4 and Theorem 6.4.1, using that } S \mathbin{\mathit{par}} T \text{ is closed and DI } \} \\ & (\forall\, t :: \mathit{Correct}.(\mathbf{f}(S \mathbin{\mathit{par}} T).t)) \\ = & \quad \{ \text{ second property above (correctness on } \Lambda \text{ } \} \end{array}$$

$$
\begin{array}{ll}
& (\forall\, t :: \mathbf{f}(S \mathit{\ par\ } T).t \sqsupseteq \backsim\Box) \\
\equiv & \quad \{ \text{ the assumptions imply } S \mathit{\ par\ } T = S \cup T \ \} \\
& (\forall\, t :: \mathbf{f}(S \cup T).t \sqsupseteq \backsim\Box) \\
\equiv & \quad \{ \text{ definition of } \mathbf{f} \text{ for systems } \} \\
& (\forall\, t :: \mathbf{f}S.t \parallel \mathbf{f}T.t \sqsupseteq \backsim\Box) \\
\equiv & \quad \{ \text{ last property above (factorization on } \Lambda), \text{ using that } \backsim\backsim\Box = \Box \text{ is the unit of } \parallel \ \} \\
& (\forall\, t :: \mathbf{f}S.t \sqsupseteq \backsim(\mathbf{f}T.t)) \\
\equiv & \quad \{ \text{ definition of } \sqsupseteq \text{ for ECFs } \} \\
& \mathbf{f}S \sqsupseteq \backsim \mathbf{f}T
\end{array}
$$

■

Using ECFs, the after-operator can also be defined sensibly for traces not in the trace set. For ECF f and trace t, ECF f/t (pronounced 'f after t') is defined by

$$(f/t).u \;=\; f.tu \;. \tag{7.14}$$

Thus, for process P and trace $t \notin \mathbf{t}P$, we have $(\mathbf{f}P)/t = \mathbf{f}\top$ if $\mathbf{f}P.t = \top$, and $(\mathbf{f}P)/t = \mathbf{f}\bot$ if $\mathbf{f}P.t = \bot$. Therefore, we define P/t for $t \notin \mathbf{t}P$ by

$$P/t \;=\; \mathbf{f}P.t \;. \tag{7.15}$$

Observe that these definitions imply $(\mathbf{f}P)/t = \mathbf{f}(P/t)$.

Theorem 4.7.4 reformulates JTU-Rules $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ in terms of the after-operator. With the extended after-operator, these three rules can be combined into a single rule. It is called the **neighbor-swap rule** and defined as follows:

- Process P satisfies **Rule** $\mathcal{NS}$ when for all traces s and symbols a and b we have

$$a \in \mathbf{o}P \;\vee\; b \in \mathbf{i}P \;\;\Rightarrow\;\; P/sab \sqsupseteq P/sba \;.$$

Note the disjunction on the left. For process P, relation $\gtrsim_P$ on $\mathbf{a}P$ defined by

$$a \gtrsim_P b \;\;\equiv\;\; a \in \mathbf{o}P \;\vee\; b \in \mathbf{i}P \tag{7.16}$$

happens to be a pre-order (we omit the subscript when P is clear from the context). The corresponding equivalence is given by $\eqsim$, expressing that symbols have the same direction. The neighbor-swap rule is equivalent to

$$(\forall\, s, a, b, t : a \gtrsim b : \mathbf{f}P.sabt \sqsupseteq \mathbf{f}P.sbat) \;. \tag{7.17}$$

7.2.2 Theorem The conjunction of Rules $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ is equivalent to Rule $\mathcal{NS}$.

Proof That Rule $\mathcal{NS}$ implies the other three follows immediately from Theorem 4.7.4. Assuming process P satisfies Rules $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$, we show that it satisfies the neighbor-swap rule. On account of symmetry (under reflection) we may also assume that $a \in \mathbf{o}P$. We distinguish the cases $b \in \mathbf{o}P$ and $b \in \mathbf{i}P$.

Case $b \in \mathbf{o}P$: If $sab \in \mathbf{t}P$, then Theorem 4.7.4 applies and together with Rule $\mathcal{X}$ yields $P/sab = P/sba$, so we are done. Now assume $sab \notin \mathbf{t}P$. We infer

$$\begin{array}{lcl} sa \in \mathbf{t}P \wedge sab \notin \mathbf{t}P & \Rightarrow & P/sab = \top \ , \\ s \in \mathbf{t}P \wedge sa \notin \mathbf{t}P & \Rightarrow & P/sab = \top \ , \\ s \notin \mathbf{t}P & \Rightarrow & P/sab = P/sba \ . \end{array}$$

In all three cases we thus find $P/sab \sqsupseteq P/sba$.

Case $b \in \mathbf{i}P$: If $sa \in \mathbf{t}P$ and $sb \in \mathbf{t}P$, then Rule $\mathcal{Z}$ yields $sab \in \mathbf{t}P$ and $sba \in \mathbf{t}P$. In that case, Theorem 4.7.4 applies and together with Rule $\mathcal{Y}$ gives $P/sab \sqsupseteq P/sba$. Now assume $sa \notin \mathbf{t}P$ (the case $sb \notin \mathbf{t}P$ follows by symmetry under reflection). We infer

$$\begin{array}{lcl} s \in \mathbf{t}P \wedge sa \notin \mathbf{t}P & \Rightarrow & P/sab = \top \ , \\ s \notin \mathbf{t}P & \Rightarrow & P/sab = P/sba \ . \end{array}$$

In both cases we again find $P/sab \sqsupseteq P/sba$, which completes the proof. ∎

Finally, we outline a derivation of the equivalence of characterizations 4 and 5 of Theorem 4.7.3 when translated into the Extended DI Model. That is, we prove that process P satisfies the extended JTU-Rules if and only if $Correct.\{P, \backsim P\}$.

Proof We define relation $\precsim$ on $(\mathbf{a}P)^*$ as the smallest transitive relation satisfying

$$a \gtrsim b \quad \equiv \quad sabt \precsim sbat \tag{7.18}$$

for all traces s and t and symbols a and b. It depends on $\mathbf{i}P$ and $\mathbf{o}P$ but not on $\mathbf{t}P$. Relation $\precsim$ is a pre-order. An interpretation of $\precsim$ is given below (it is related to the *composability* relation of [Udd84] and the reordering relations of [CM84, JHJ89]). First we derive

$$\begin{array}{ll} & \text{'}P \text{ satisfies Rules } \mathcal{X}, \mathcal{Y}, \text{ and } \mathcal{Z}\text{'} \\ \equiv & \quad \{ \text{ Theorem 7.2.2, Equation 7.17 } \} \\ & (\forall\, s, a, b, t : a \gtrsim b : \mathbf{f}P.sabt \sqsupseteq \mathbf{f}P.sbat) \\ \equiv & \quad \{ \text{ induction on the definition of } \precsim \text{ in terms of } \gtrsim \ \} \\ & (\forall\, u, v : u \precsim v : \mathbf{f}P.u \sqsupseteq \mathbf{f}P.v) \\ \equiv & \quad \{ \text{ property of } \sqsupseteq \text{ on } \Lambda \ \} \\ & (\forall\, u, v : u \precsim v : \mathbf{f}P.u \between \backsim\mathbf{f}P.v \sqsubseteq \square) \\ \equiv & \quad \{ \text{ property of } \backsim \text{ for ECFs } \} \\ & (\forall\, u, v : u \precsim v : \mathbf{f}P.u \between \mathbf{f}(\backsim P).v \sqsupseteq \square) \end{array}$$

The last expression taken together with Rule $\mathcal{W}$ is equivalent to $Correct.\{P, \backsim P\}$, because $u \precsim v$ characterizes the indifferent states of the wire interface between P and $\backsim P$, and $\mathbf{f}P.u \between \mathbf{f}(\backsim P).v \sqsubseteq \square$ expresses that in 'state' (u, v) there is neither interference at P or $\backsim P$, nor deadlock. Rule $\mathcal{W}$ takes care of interference at the wire interface. ∎

7.3 GLBs and Composites

Every process has an ECF, but not every ECF is obtainable from a process. We will now characterize the ECFs of processes and explain how to compute greatest lower bounds and composites using ECFs.

For ECF f and disjoint alphabets I and O, the seven predicates $\mathcal{E}_i$ are defined by

$$\begin{array}{lll}
\mathcal{E}_0\colon & t\restriction A = u\restriction A & \Rightarrow\ f.t = f.u \quad \text{where } A = I \cup O \\
\mathcal{E}_1\colon & f.t = \top & \Rightarrow\ f.tu = \top \\
\mathcal{E}_2\colon & f.t = \bot & \Rightarrow\ f.tu = \bot \\
\mathcal{E}_3\colon & f.tu = \top \wedge u \in I^* & \Rightarrow\ f.t = \top \\
\mathcal{E}_4\colon & f.tu = \bot \wedge u \in O^* & \Rightarrow\ f.t = \bot \\
\mathcal{E}_5\colon & f.t = \nabla & \Rightarrow\ (\exists\, a : a \in O : f.ta \neq \top) \\
\mathcal{E}_6\colon & f.t = \Delta & \Rightarrow\ (\exists\, a : a \in I : f.ta \neq \bot)
\end{array}$$

where each predicate should be read as universally quantified over traces t and u. Note that $\mathcal{E}_2$, $\mathcal{E}_4$, and $\mathcal{E}_6$ are the "reflections" of $\mathcal{E}_1$, $\mathcal{E}_3$, and $\mathcal{E}_5$ respectively; $\mathcal{E}_0$ is its own "reflection". Predicate $\mathcal{E}_0$ expresses that the f-images depend only on symbols in $I \cup O$. Predicates $\mathcal{E}_1$ and $\mathcal{E}_2$ express that $\top$ and $\bot$ "persist". Predicates $\mathcal{E}_3$ and $\mathcal{E}_4$ together capture (indirectly) that a trace "stepping outside the trace set" via an output is mapped to $\top$, and via an input to $\bot$. Predicates $\mathcal{E}_5$ and $\mathcal{E}_6$ derive from requirements 5 and 6 for processes.

We write $\mathcal{E}_{j,k}$ for the conjunction of $\mathcal{E}_j$ and $\mathcal{E}_k$. The subsets $\mathcal{ECF}_i(I, O)$ of $\mathcal{ECF}$ corresponding to the predicates $\mathcal{E}_i$ are defined by

$$\mathcal{ECF}_i(I, O) \ = \ \{f : \mathcal{E}_i : f\} \ . \tag{7.19}$$

Similarly, we write $\mathcal{ECF}_{j,k}(I, O)$ for the intersection of $\mathcal{ECF}_j(I, O)$ and $\mathcal{ECF}_k(I, O)$.

The next theorem gives a one-one correspondence between processes in $\mathcal{PROC}(I, O)$ and ECFs in $\mathcal{ECF}_{0,\ldots,6}(I, O)$.

7.3.1 Theorem (*Characterization of process ECFs*)
For process P we have

$$\mathbf{f}P \ \in \ \mathcal{ECF}_{0,\ldots,6}(\mathbf{i}P, \mathbf{o}P) \ .$$

Conversely, if $f \in \mathcal{ECF}_{0,\ldots,6}(I, O)$, then quintuple

$$(\ I,\ O,\ f^{\frown}.\nabla,\ f^{\frown}.\Box,\ f^{\frown}.\Delta\ )\ , \tag{7.20}$$

where $f^{\frown}.\lambda = \{t : f.t = \lambda : t\}$, is a process with ECF f. ∎

The **tree** of ECF f is a vertex- and edge-labeled directed graph, where the edge-labeled directed graph is given by

$$(\ \Sigma^*,\ \{t, a :: (t, a, ta)\}\ ) \tag{7.21}$$

and f is the vertex-labeling. Such a graph is a tree with root ε.

7.3.2 Example Let $I = \{a, b\}$ and $O = \{c, d\}$. Figure 7.1 presents corresponding parts of the trees of six ECFs f_1 through f_6 (the bold vertex labels will be explained in a moment). ECFs f_1 and f_2 may be obtained from suitable processes: they are postulated to satisfy $\mathcal{E}_0$ through $\mathcal{E}_6$.

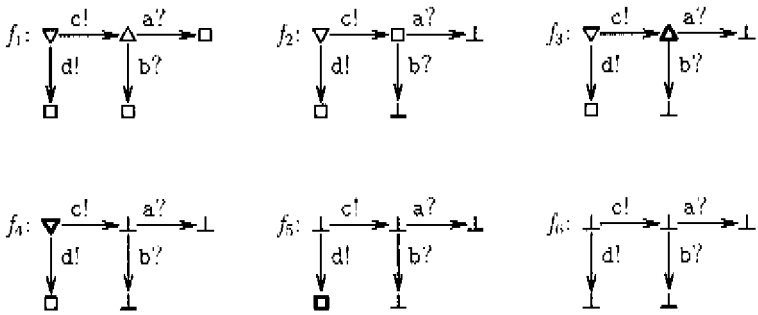


Figure 7.1: Subgraphs of ECFs f_1 through f_6

ECF f_3 is the greatest lower bound in $\mathcal{ECF}$ of f_1 and f_2, that is, taken tracewise. It is does not satisfy $\mathcal{E}_6$; see the state with the bold Δ-label. Every ECF g obtained from a process such that $g \sqsubseteq f_3$, labels that bold state with $\perp$. Therefore, to find the greatest lower bound of f_1 and f_2 in $\mathbf{f}\mathcal{PROC}$, this label should be changed to $\perp$. The result is shown as f_4. However, f_4 fails to satisfy $\mathcal{E}_4$ (at the bold state). The only way to eliminate this violation without increasing the ECF, is to change the bold ∇-label to $\perp$, yielding f_5. Now, f_5 violates $\mathcal{E}_2$, which can be repaired by changing the bold $\Box$-label (and all its successors) to $\perp$ as well. The final result is f_6. This shows how the greatest lower bound in $\mathcal{PROC}$ may be approximated by successive reductions. ∎

The question remains whether the approximations suggested in the previous example actually converge to a process. That this is indeed the case is a little delicate and can be understood as follows.

Let I and O be disjoint alphabets; these are implicit parameters to the definitions in the remainder of this subsection. For $i \in \{0, 2, 4, 6\}$, define transformations Φ_i on $\mathcal{ECF}$ by

$$\Phi_0.f.t \;=\; \sqcap\{u : t \lceil A = u \lceil A : f.u\} \qquad \text{where } A = I \cup O$$

$$\Phi_2.f.t \;=\; \begin{cases} \perp & \text{if } (\exists\, t_0, t_1 : t = t_0 t_1 : f.t_0 = \perp) \\ f.t & \text{otherwise} \end{cases}$$

$$\Phi_4.f.t \;=\; \begin{cases} \perp & \text{if } (\exists\, u : u \in O^* : f.t = \perp) \\ f.t & \text{otherwise} \end{cases}$$

$$\Phi_6.f.t \;=\; \begin{cases} \perp & \text{if } f.t = \Delta \wedge (\forall\, a : a \in I : f.ta = \perp) \\ f.t & \text{otherwise} \end{cases}$$

7.3.3 Example For the ECFs of Example 7.3.2 we have $f_4 = \Phi_6.f_3$, $f_5 = \Phi_4.f_4$, and $f_6 = \Phi_2.f_5$. ∎

The next theorem states some properties of the transformations Φ_i, in particular how they relate to the predicates $\mathcal{E}_j$.

7.3.4 Theorem For $i \in \{0, 2, 4, 6\}$ we have

1. Φ_i is $\sqsubseteq$-monotonic,
2. $\Phi_i.f \sqsubseteq f$,
3. $\Phi_i.f = f$ if and only if $f \in \mathcal{ECF}_i(I, O)$,
4. $g \sqsubseteq f$ with $g \in \mathcal{ECF}_{2,4,6}(I, O)$ implies $g \sqsubseteq \Phi_i.f$,
5. $f \in \mathcal{ECF}_{1,3,5}(I, O)$ implies $\Phi_i.f \in \mathcal{ECF}_{1,3,5}(I, O)$.

∎

Define transformation Φ on $\mathcal{ECF}$ as the composition of the four Φ_i:

$$\Phi \;=\; \Phi_0 \circ \Phi_2 \circ \Phi_4 \circ \Phi_6 \;.$$

Transformation Φ inherits all properties of the four Φ_i, except that property 3 should be restated as

3′. $\Phi.f = f$ if and only if $f \in \mathcal{ECF}_{0,2,4,6}(I, O)$.

By iterating Φ sufficiently "often" a fixpoint is reached (see [CC79]): there exists a least ordinal κ such that $\Phi^\kappa.f$ is a fixpoint of Φ. It turns out that ω iterations, where ω is the least infinite ordinal, suffices. In fact, for ECFs derived from finite-state processes, a finite number of iterations will do. Define transformation $\lfloor\_\rfloor$ on $\mathcal{ECF}$ by

$$\lfloor f \rfloor \;=\; \Phi^\omega.f \;.$$

From the next theorem we may infer that for $f \in \mathcal{ECF}_{1,3,5}(I, O)$, $\lfloor f \rfloor$ is the greatest *process* ECF at most f, that is, we have

$$\lfloor f \rfloor \;=\; \sqcup\{g : g \in \mathcal{ECF}_{0,\ldots,6}(I, O) \wedge g \sqsubseteq f : g\} \;. \tag{7.22}$$

7.3.5 Theorem We have

1. $\lfloor\_\rfloor$ is $\sqsubseteq$-monotonic,
2. $\lfloor f \rfloor \sqsubseteq f$,
3. $\lfloor f \rfloor = f$ if and only if $f \in \mathcal{ECF}_{0,2,4,6}(I, O)$,
4. $g \sqsubseteq f$ with $g \in \mathcal{ECF}_{2,4,6}(I, O)$ implies $g \sqsubseteq \lfloor f \rfloor$,

5. $f \in \mathcal{ECF}_{1,3,5}(I,O)$ implies $\lfloor f \rfloor \in \mathcal{ECF}_{0,\ldots,6}(I,O)$.

■

7.3.6 Example The premise in statement 5 of Theorem 7.3.5 is indispensable. For instance, assume $\mathtt{a} \in I$, $f.\varepsilon = \Box$, and $f.t = \top$ for $t \neq \varepsilon$; in particular $f.\mathtt{a} = \top$. In that case, $f \notin \mathcal{ECF}_{1,3,5}(I,O)$ because f violates $\mathcal{E}_3$. Moreover, $\Phi_1 f = f$ and, hence, also $\lfloor f \rfloor = f$. ■

Let $W \subseteq \mathcal{PROC}$. We are interested in computing the greatest lower bound of W with respect to $\sqsubseteq$ on $\mathcal{PROC}$. Without loss of generality we may assume $W \cap \{\top, \bot\} = \varnothing$, since $\sqcap W = \sqcap(W \setminus \{\top\})$ and $\bot \in W \Rightarrow \sqcap W = \bot$. Moreover we may assume $W \subseteq \mathcal{PROC}(I,O)$, because if W contains processes P and Q for which $(\mathbf{i}P, \mathbf{o}P) \neq (\mathbf{i}Q, \mathbf{o}Q)$, then $\sqcap W = \bot$.

7.3.7 Theorem Let $W \subseteq \mathcal{PROC}(I,O)$. Then ECF f defined by

$$f \;=\; \sqcap\{P : P \in W : \mathbf{f}P\}\,,$$

where the greatest lower bound is taken in $\mathcal{ECF}$, (by tracewise application) is in $\mathcal{ECF}_{1,3,5}$, and $\sqcap W$ is the process in $\mathcal{PROC}(I,O)$ corresponding to ECF $\lfloor f \rfloor$.

Furthermore, for system S we have that its ECF $\mathbf{f}S$ is in $\mathcal{ECF}_{1,3,5}$ and its composite $[\![S]\!]$ is the process in $\mathcal{PROC}(\mathbf{xi}S, \mathbf{xo}S)$ corresponding to ECF $\lfloor \mathbf{f}S \rfloor$, where $\lfloor \_ \rfloor$ is taken with $I = \mathbf{xi}S$ and $O = \mathbf{xo}S$.

Proof For the first statement it suffices to verify that for $V \subseteq \mathcal{ECF}_{1,3,5}(I,O)$ we have $\sqcap V \in \mathcal{ECF}_{1,3,5}(I,O)$, where $\sqcap$ is taken in $\mathcal{ECF}$ (tracewise). This verification is merely tedious, and omitted here.

A similar verification yields that for system S we have $\mathbf{f}S \in \mathcal{ECF}_{1,3,5}(I,O)$, where $I = \mathbf{xi}S$ and $O = \mathbf{xo}S$. For the final proof obligation, we now derive

$$\begin{array}{ll}
& [\![S]\!] \\
= & \quad \{ \text{ definition of } [\![_]\!] \ \} \\
& \backsim \sqcap \mathit{Friends}.S \\
= & \quad \{ \text{ definition of } \mathit{Friends} \ \} \\
& \backsim \sqcap \{R : R \in \mathcal{PROC} \wedge \mathit{Correct}.(S \mathbin{\mathit{par}} \{R\}) : R\} \\
= & \quad \{ \text{ reflection turns } \sqsubseteq \text{ around, } \mathcal{PROC} \text{ is closed under reflection } \} \\
& \sqcup \{R : R \in \mathcal{PROC} \wedge \mathit{Correct}.(S \mathbin{\mathit{par}} \{\backsim R\}) : R\} \\
= & \quad \{ \text{ Theorem 7.2.1 } \} \\
& \sqcup \{R : R \in \mathcal{PROC} \wedge \mathbf{f}S \sqsupseteq \mathbf{f}R : R\} \\
= & \quad \{ \lfloor f \rfloor \text{ is the greatest process ECF at most } f, \text{ using that } \mathbf{f}S \in \mathcal{ECF}_{1,3,5}(I,O) \ \} \\
& \text{`process corresponding to } \lfloor \mathbf{f}S \rfloor\text{'}
\end{array}$$

■

7.3.8 Note In Example 7.3.2 we have seen that the ECF of the greatest lower bound of two processes is not necessarily obtained by taking the tracewise greatest lower bound. The tracewise greatest lower bound f may still be too large and must be lowered to $\lfloor f \rfloor$.

It turns out that in the DI Model of Chapter 4, greatest lower bounds can be taken tracewise without further lowering. In the Extended DI Model this fails because of condition $\mathcal{E}_6$. ∎

7.3.9 Note When considering finite-state specifications, the tracewise composition operator $\|$ on $\mathcal{ECF}$ ($\|$ appears in the definition of $\mathbf{f}S$) involves a *product* construction, whereas the transformation $\lfloor \_ \rfloor$ involves a *power* construction. This says something about the complexity of computing $[\![S]\!]$. ∎

Finally we prove the fourth statement of Theorem 4.6.4.

7.3.10 Theorem For connectable systems S and U such that S *par* U is closed we have

$$\mathit{Correct.}(S \textit{ par } U) \;=\; [\![S]\!] \sqsupseteq \rightharpoonup [\![U]\!] \,.$$

For systems S and T we have

$$S \textit{ sat } T \;\equiv\; [\![S]\!] \sqsupseteq [\![T]\!] \,.$$

Proof On account of Theorem 4.4.2 and by appropriate renaming of the internal symbols in $\bar{S}$ and $\bar{U}$, we can find systems S' and U' such that $\mathbf{n}S' \cap \mathbf{n}U' = \varnothing$, S *equ* S', U *equ* U', and S' *par* U' is **DI**. We derive

$\mathit{Correct.}(S \textit{ par } U)$
$\equiv$ { construction of S' and U' }
$\mathit{Correct.}(S' \textit{ par } U')$
$\equiv$ { Theorem (7.2.1 }
$\mathbf{f}S' \sqsupseteq \rightharpoonup \mathbf{f}U'$
$\equiv$ { Theorem 7.3.5, using that $\rightharpoonup \mathbf{f}U' \in \mathcal{ECF}_{2,4,6}(I,O)$ }
$\lfloor \mathbf{f}S' \rfloor \sqsupseteq \rightharpoonup \mathbf{f}U'$
$\equiv$ { reflection turns $\sqsupseteq$ around }
$\rightharpoonup \lfloor \mathbf{f}S' \rfloor \sqsubseteq \mathbf{f}U'$
$\equiv$ { Theorem 7.3.5, using that $\lfloor \mathbf{f}S' \rfloor \in \mathcal{ECF}_{1,3,5}(I,O)$ }
$\rightharpoonup \lfloor \mathbf{f}S' \rfloor \sqsubseteq \lfloor \mathbf{f}U' \rfloor$
$\equiv$ { reflection turns $\sqsupseteq$ around, $[\![T]\!] = \lfloor \mathbf{f}T \rfloor$ }
$[\![S']\!] \sqsupseteq \rightharpoonup [\![U']\!]$
$\equiv$ { construction of S' and U', Note 4.6.5 }
$[\![S]\!] \sqsupseteq \rightharpoonup [\![U]\!]$

Concerning the second statement we now derive

$$
\begin{array}{ll}
& S \mathit{\ sat\ } T \\
= & \quad \{ \text{ definition of } \mathit{sat} \ \} \\
& (\forall\, U :: \mathit{Correct}.(S \mathit{\ par\ } U) \ \Leftarrow \ \mathit{Correct}.(T \mathit{\ par\ } U)) \\
\equiv & \quad \{ \text{ first statement } \} \\
& (\forall\, U :: [\![S]\!] \sqsupseteq \backsim[\![U]\!] \ \Leftarrow \ [\![T]\!] \sqsupseteq \backsim[\![U]\!]) \\
\Rightarrow & \quad \{ \text{ instantiate with } U := \{\backsim[\![T]\!]\}, \text{ using that } \mathcal{DI} \text{ is closed under } \backsim \ \} \\
\Leftarrow & \quad \{ \text{ transitivity of } \sqsubseteq \ \} \\
& [\![S]\!] \sqsupseteq [\![T]\!]
\end{array}
$$

This concludes the proof. ∎

7.3.11 Note There is some freedom in defining the predicates $\mathcal{E}_i$ that characterize the ECFs of processes. For instance, $\mathcal{E}_3$ could be changed to

$$f.tu = \top \ \wedge \ u\restriction I = \varepsilon \ \ \Rightarrow \ \ f.t = \top \ , \tag{7.23}$$

without affecting the conjunction $\mathcal{E}_0 \wedge \mathcal{E}_3$, because $\mathcal{E}_0$ implies

$$u\restriction(I \cup O) = \varepsilon \ \ \Rightarrow \ \ f.t = f.tu \ . \tag{7.24}$$

It may, however, make a difference in the analysis of the related closure transformations Φ_i. Our choice of $\mathcal{E}_i$'s involved some fine tuning. We do not claim that there is a recipe for fine tuning, nor that it is always possible to determine fixpoints by taking limits based on separate closure properties. ∎

7.3.12 Note The juggling with alphabets I and O, in particular in Theorem 7.3.7, could have been avoided by considering triples (I, O, f). For instance, the set $\mathcal{PROC}'$ defined by

$$\mathcal{PROC}' \ = \ \{ \ I, O, f : f \in \mathcal{ECF}_{1,3,5}(I, O) : (I, O, f) \ \}$$

could be used as a new process space, from which systems can be built in the usual way. In fact, it is even *nicer* to describe process structure not by the two alphabets I and O, but by a mapping e from Σ to

$$\{ \ \restriction, \Diamond, !, ?, \blacklozenge, \bot \} \ ,$$

where $\Diamond$ stands for 'unused', ! for 'external output', ? for 'external input', $\blacklozenge$ for 'internal', $\restriction$ for 'conflicting', and $\top$ for the reflection of $\bot$. Processes are then described by pairs (e, f) satisfying certain predicates. This idea is worked out in more detail in [Ver94].

$\mathcal{PROC}'$ is closed under composition and taking greatest lower bounds, both done trace-wise. However, it is not closed under reflection and, in general, we do *not* have

$$\lfloor \backsim f \rfloor \ = \ \backsim \lfloor f \rfloor \ . \tag{7.25}$$

In spite of the additional processes in $\mathcal{PROC}'$ — these are also used as test environments — the satisfaction relation on the subspace of $\mathcal{PROC}'$ corresponding to $\mathcal{PROC}$ is the same as *sat*. An advantage of the extended space $\mathcal{PROC}'$ is that greatest lower bounds and composites (now corresponding to $\mathcal{PROC}$) may be approximated in smaller steps that stay within the process space. ∎

Chapter 8

Output Nondeterminism

A process specification captures both the obligations of the process (regarding output) and the obligations of its environment (regarding input). The output obligations play a role similar to that of the post-condition in programming, whereas the role of the input obligations resembles that of the pre-condition. In programming, the post-condition often provides better guidance for design than the pre-condition. Similarly, the output obligations are generally more important for design of delay-insensitive systems than the input obligations.

In this chapter, we take a closer look at nondeterminism and its relation to design freedom in the context of the Extended DI Model. Taking the introductory observations at heart, we will focus on nondeterminism related to output, in particular. Input nondeterminism—think, for instance, of the merge process—is less interesting.

8.1 Output Refusal Sets

We will define output (non)determinism in terms of refusal sets, which are familiar from the Failures Model for CSP (see [Hoa85]). For process P, we say that alphabet A, $A \subseteq \mathbf{o}P$, is an **output refusal set**, or briefly a **refusal**, at trace t, $t \in \mathbf{t}P$, when

$$(\exists\, u : u \in (\mathbf{o}P \setminus A)^* : \mathbf{f}P.tu \in \{\Box, \Delta\}) \,. \tag{8.1}$$

The idea is that alphabet $A \subseteq \mathbf{o}P$ is a refusal at $t \in \mathbf{t}P$, if process P, after doing t, can evolve to a state, by doing output, where it has no further output obligations, without having done any of the outputs in A.

8.1.1 **Note** In the Extended DI Model, it does not make much sense to introduce *input* refusal sets. Whenever a process *may* refuse an input, the environment had better not send it at all, acting as if the process *cannot* accept the input. Consider, for example, system $S = \{\ \mathtt{W(a;b)},\ \mathtt{W(b;c)}\ \}$. When analyzing the operation of this system, one will find an execution scenario in which the environment sends two a-inputs in a row before receiving output without causing interference. This happens if the first wire has transferred its signal to the second wire before the second a-input arrives. However, this same behavior of the

environment may also result in interference. Hence, the environment should refrain from sending the second a-input before receiving the c-output.

Of course, one may look at the output refusal sets of the reflection to say something about the nondeterminism in the environment of the process. ◼

The definition of refusals can be extended to traces not in $\mathbf{t}P$ as follows. For process P, alphabet $A \subseteq \mathbf{o}P$ is a refusal at trace t, when

$$(\exists\, u : u \in (\mathbf{o}P \setminus A)^* : \mathbf{f}P.tu \sqsubseteq \sqcap) \;. \tag{8.2}$$

Let us analyze this definition. If for some $u \in (\mathbf{o}P)^*$ we have $\mathbf{f}P.tu = \bot$, then $\mathbf{f}P.t = \bot$ on account of property $\mathcal{E}_4$ of process ECFs. This shows that (8.2) is indeed an extension of (8.1). If $\mathbf{f}P.t = \top$, then for any $u \in (\mathbf{o}P)^*$ we have $\mathbf{f}P.tu = \top$ on account of property $\mathcal{E}_1$. Hence there are no refusals at such t. If $\mathbf{f}P.t \sqsubseteq \sqcup$, in particular if $\mathbf{f}P.t = \bot$, then any $A \subseteq \mathbf{o}P$ is a refusal at t. Apparently the extreme cases with $t \notin \mathbf{t}P$ are not very interesting.

Definition (8.2) is preferred because it simplifies proofs. For instance, the following statement is an immediate consequence of (8.2): If alphabet $A \subseteq \mathbf{o}P$ is a refusal at trace t for process P and $P \sqsupseteq Q$, then A is also a refusal at t for Q.

8.1.2 Example Figure 8.1 shows the state graphs of the fifteen DI processes with outputs $\{\mathtt{a}, \mathtt{b}\}$ and no inputs. In some of the state graphs, dotted edges appear. These transitions step outside the trace set and have been included to simplify comparison under $\sqsubseteq$. The two columns in the middle are mutually symmetric under swapping of a and b.

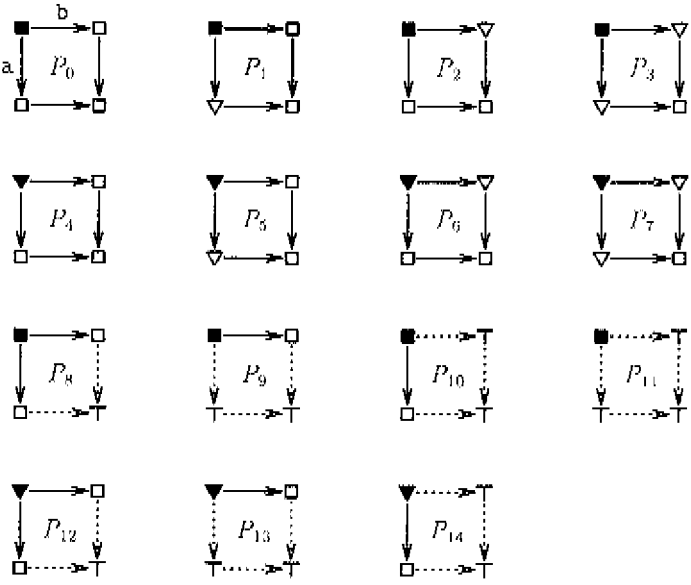


Figure 8.1: State graphs of all DI processes with two outputs

It may be surprising that all these processes can actually be made from building blocks. For instance, in process P_1, neither output is guaranteed initially, but output b is guaranteed after a. However, output a is not guaranteed after b. Figure 8.2 shows a system with composite P_1.

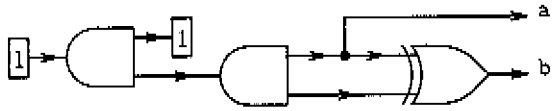


Figure 8.2: System with composite P_1

Table 8.1 lists the refusals at ε of processes P_0 through P_{14}.

process	refusals at trace ε
$P_{0..3}, P_{8..11}$	$\varnothing, \{\mathtt{a}\}, \{\mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}$
P_4, P_{12}	$\varnothing, \{\mathtt{a}\}, \{\mathtt{b}\}$
P_5, P_{13}	$\varnothing, \{\mathtt{a}\}$
P_6, P_{14}	$\varnothing, \{\mathtt{b}\}$
P_7	$\varnothing$

Table 8.1: Refusals of P_0 through P_{14}

Refusals are "downward closed", in the sense that if for process P, alphabet $A \subseteq \mathbf{o}P$ is a refusal at trace t, and alphabet $B \subseteq A$, then also B is a refusal at t. Hence, only $\subseteq$-maximal refusals are interesting.

Refusals "partly propagate backward over outputs", in the sense that if for process P, alphabet $A \subseteq \mathbf{o}P$ is a refusal at trace ta with $a \in \mathbf{o}P$, then $A \setminus \{a\}$ is a refusal at t.

For process P, refusal $A \subseteq \mathbf{o}P$ at $t \in \mathbf{t}P$ is called **trivial** when

$$(\forall\, u : u \in (\mathbf{o}P)^* \wedge tu \in \mathbf{t}P : u \restriction A = \varepsilon)\ , \tag{8.3}$$

that is, when P, after doing t, *cannot* produce any of the outputs in A.

8.1.3 Example Table 8.2 lists the trivial refusals at ε of the processes in Example 8.1.2. Note that there is no process with precisely $\varnothing$, $\{\mathtt{a}\}$, and $\{\mathtt{b}\}$ as trivial refusals. Also note

process	trivial refusals at ε
$P_{0..8}, P_{12}$	$\varnothing$
P_9, P_{13}	$\varnothing, \{\mathtt{a}\}$
P_{10}, P_{14}	$\varnothing, \{\mathtt{b}\}$
P_{11}	$\varnothing, \{\mathtt{a}\}, \{\mathtt{b}\}, \{\mathtt{a}, \mathtt{b}\}$

Table 8.2: Trivial refusals of P_0 through P_{14}

that the four $\sqsubseteq$-maximal processes P_7, P_{11}, P_{13}, and P_{14} are the only ones for which all refusals at ε are trivial.

Here are some general properties of trivial refusals. If $A \subseteq \mathbf{o}P$ is a trivial refusal at t and $B \subseteq A$, then B is also a trivial refusal at t. If $\varnothing$ is a refusal at t, then it is a trivial refusal. If A and B are trivial refusals at t, then so is $A \cup B$. However, if $A \subseteq \mathbf{o}P$ satisfies (8.3), then it is not necessarily a refusal at t as the next example shows.

8.1.4 Example Consider process $P = (\varnothing, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}\}^* \cup \{\mathtt{b}\}^*, \varnothing, \varnothing)$. P has outputs $\mathtt{a}$ and $\mathtt{b}$, and all its traces are transient. It will either constantly output $\mathtt{a}$ or constantly $\mathtt{b}$. Alphabet $A = \{\mathtt{b}\}$ satisfies (8.3) for trace $t = \mathtt{a}$, but A is not a refusal at t (since all traces are transient).

In fact, P has no refusals anywhere. At the initial state ε, one might expect the refusals $\varnothing$, $\{\mathtt{a}\}$, and $\{\mathtt{b}\}$ (but not $\{\mathtt{a}, \mathtt{b}\}$). Observe that P is not a DI process, since it violates Rule $\mathcal{W}$. ∎

The definition of refusal set can be modified to deal more properly with anomalous processes like in the preceding example, but this requires an additional quantification. Anyway, it would not make a difference for DI processes as shown by the following theorem. The theorem characterizes a trivial refusal of a DI process as a set of outputs that are all non-successors. This is also in line with the Failures Model for CSP.

8.1.5 Theorem Let P be a DI process. Alphabet $A \subseteq \mathbf{o}P$ is a trivial refusal at $t \in \mathbf{t}P$ if and only if

$$(\forall\, a : a \in A : ta \notin \mathbf{t}P)\ . \tag{8.4}$$

Proof That (8.4) holds for a trivial refusal at $t \in \mathbf{t}P$ follows immediately from (8.3), the definition of 'trivial'.

Assuming $A \subseteq \mathbf{o}P$, $t \in \mathbf{t}P$, and (8.4), we show that A is a trivial refusal at t. The set V of all traces $u \in (\mathbf{o}P)^*$ such that $tu \in \mathbf{t}P$ is finite because P is a DI process, in particular because $\mathbf{o}P$ is finite and Rule $\mathcal{W}$ is satisfied. Let $u \in V$ be of maximal length, then $tu \notin \nabla P$. Furthermore, $u \in (\mathbf{o}P \setminus A)^*$, because if symbol $a \in A$ occurs in u, then on account of Rule $\mathcal{X}$, we infer $ta \in \mathbf{t}P$ from $tu \in \mathbf{t}P$, contradicting assumption (8.4). Therefore, A is a refusal at t. In fact, for every $u \in V$ we have $u \restriction A = \varepsilon$ on account of Rule $\mathcal{X}$ and (8.4). Consequently, A is a trivial refusal at t. ∎

From now on we will restrict our attention to DI processes. We call DI process P (output) **deterministic** when all its output refusal sets are trivial, that is, when it can only "refuse" to produce output that is disallowed anyway. Put differently, whenever an output-deterministic process is capable of producing an output, it will eventually produce that output. The definition of output-deterministic processes in terms of output refusal sets is the same as that of deterministic processes based on refusal sets in CSP. In the Extended DI Model, however, refusals are a derived concept, whereas in the Failures Model for CSP they are fundamental (this is also reported in [Jos92]). Furthermore, in CSP the deterministic processes are maximal under the refinement order and this is not the case in the Extended DI Model, because an output-deterministic process need not be maximal with respect to input processing.

Below we will give an alternative characterization of output-deterministic DI processes. For that purpose, the definitions of Rules $\mathcal{Z}^{out}$ (no output choice), $\mathcal{Z}^{in}$ (no input choice), and $\mathcal{Z}'$ (choice-free) are carried over to the Extended DI Model as they are. For the sake of convenience, we repeat the definitions of maximally transient and Rule $\mathcal{Z}^{out}$:

- Process P is maximally transient when for all traces s and outputs a we have

$$sa \in \mathbf{t}P \;\Rightarrow\; s \in \nabla P \;,$$

- Process P satisfies Rule $\mathcal{Z}^{out}$ when for all traces s and distinct outputs a and b we have

$$sa \subset \mathbf{t}P \wedge sb \in \mathbf{t}P \;\Rightarrow\; sab \in \mathbf{t}P \wedge sba \in \mathbf{t}P \;.$$

8.1.6 Theorem (*Characterization of Output-Deterministic Processes*)
DI process P is output deterministic if and only if it is maximally transient and satisfies Rule $\mathcal{Z}^{out}$.

Proof We start by proving that if P is maximally transient and satisfies $\mathcal{Z}^{out}$ then all its refusals are trivial. Assume $A \subseteq \mathbf{o}P$ is a refusal at $t \in \mathbf{t}P$. According to Theorem 8.1.5, it suffices to prove (8.4). Assuming $a \in A$, we show $ta \notin \mathbf{t}P$. Since A is a refusal at t, let $u \in (\mathbf{o}P)^*$ such that $\mathbf{f}P.tu \in \{\square, \Delta\}$. We derive

$$\begin{array}{ll} & tu \in \mathbf{t}P \wedge tu \notin \nabla P \\ \Rightarrow & \{\ P \text{ is maximally transient, using } tu \in \mathbf{t}P \text{ and } a \in \mathbf{o}P\ \} \\ & tu \in \mathbf{t}P \wedge tua \notin \mathbf{t}P \\ \Rightarrow & \{\ P \text{ satisfies Rule } \mathcal{Z}^{out},\ u \text{ does not contain } a \in \mathbf{o}P\ \} \\ & ta \notin \mathbf{t}P \end{array}$$

This completes the first part. Now we do the second part: If all of P's refusals are trivial, then P is maximally transient and satisfies $\mathcal{Z}^{out}$.

First we prove that P is maximally transient. Assuming $a \in \mathbf{o}P$ and $ta \in \mathbf{t}P$ we prove $t \in \nabla P$. From the assumptions we infer that $\{a\}$ is not a refusal at t, since it would not be trivial on account of $ta \in \mathbf{t}P$. If $\mathbf{f}P.t \sqsubseteq \square$, then $\{a\}$ would be a refusal set at t. Consequently, we have $t \in \nabla P$.

Finally we prove that P satisfies Rule $\mathcal{Z}^{out}$. Let a and b be distinct outputs, such that $ta \in \mathbf{t}P$ and $tb \in \mathbf{t}P$. As above, $\{a\}$ is not a refusal at t. Hence, $\{a\}$ is not a refusal at tb either. (for, otherwise, $\{a\}$ would be a refusal at t on account of $b \notin \{a\}$ and backward propagation over $b \in \mathbf{o}P$). Therefore, $tba \in \mathbf{t}P$, because, otherwise, $\{a\}$ would be a (trivial) refusal at tb. Similarly, we have $tab \in \mathbf{t}P$, which concludes the proof. ◢

An immediate consequence of this theorem is that the (progressive embeddings of the) undetermined selector, arbiter, and sequencer are not output deterministic, since they fail Rule $\mathcal{Z}^{out}$, whereas all other building blocks are output deterministic. Also the one-all of Example 5.4.1 is output deterministic.

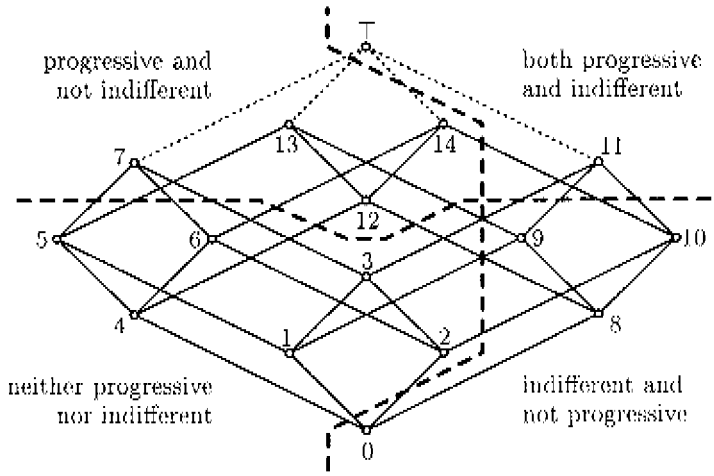


Figure 8.3: Hasse diagram for order on 2-output DI processes

8.1.7 Example Figure 8.3 depicts the Hasse diagram of relation $\sqsubseteq$ on the fifteen processes P_0 through P_{14} in Example 8.1.2. The dotted edges indicate the relationship to $\top$ (including $\top$, the Hasse diagram is a hypercube). The $\sqsubseteq$-maximal processes are P_7, P_{11}, P_{13} and P_{14}. These four are the only output-deterministic processes. P_{12} is also maximally transient but it has output choice (that is, it violates Rule $\mathcal{Z}^{out}$). Note that P_{11} is both maximally and minimally transient. P_0 is the least process. ∎

The following theorem generalizes Theorem 5.3.3, which expresses that the set of DI processes satisfying Rules $\mathcal{Y}_\alpha^{out}$ and $\mathcal{Z}_\alpha^{out}$ is closed under composition. Instead of $\mathcal{Y}_\alpha^{out}$ we now require the processes to be maximally transient, that is, output deterministic.

8.1.8 Theorem The set of output-deterministic DI processes is closed under composition. Proof idea: Use Theorem 8.1.6 and enhanced characteristic functions. ∎

To see the relationship with Theorem 5.3.3 we extend Rules $\mathcal{Y}^{out}$ and $\mathcal{Y}^{in}$ to the Extended DI Model as follows:

- P satisfies **Rule** $\mathcal{Y}^{out}$ when for all traces s and t, outputs a and b, and input c we have

$$\begin{aligned} scatb \in \mathbf{t}P \wedge sact \in \mathbf{t}P &\Rightarrow scatb \in \mathbf{t}P \ , \\ sact \in \nabla P \wedge scat \in \mathbf{t}P &\Rightarrow scat \in \nabla P \ . \end{aligned}$$

- P satisfies **Rule** $\mathcal{Y}^{in}$ when for all traces s and t, inputs a and b, and output c we have

$$\begin{aligned} scatb \in \mathbf{t}P \wedge sact \in \mathbf{t}P &\Rightarrow scatb \in \mathbf{t}P \ , \\ sact \in \Delta P \wedge scat \in \mathbf{t}P &\Rightarrow scat \in \Delta P \ . \end{aligned}$$

Note that again we have

$$\mathcal{Y}' \;\equiv\; \mathcal{Y} \wedge \mathcal{Y}^{out} \wedge \mathcal{Y}^{in} \;. \tag{8.5}$$

The relationship between output determinism and Rule $\mathcal{Y}^{out}$ follows from the next theorem.

8.1.9 Theorem Every output-deterministic DI process satisfies Rule $\mathcal{Y}^{out}$.

Proof According to Theorem 8.1.6, P has only trivial refusals. Consider traces s and t, input a, and outputs b and c. First we derive

$$\begin{array}{ll}
& sactb \in \mathbf{t}P \;\wedge\; scat \in \mathbf{t}P \\
\equiv & \quad \{\; P \text{ has only trivial refusals} \;\} \\
& \text{`}\{b\} \text{ is not a refusal at } sact \in \mathbf{t}P\text{'} \;\wedge\; scat \in \mathbf{t}P \\
\Rightarrow & \quad \{\; P \text{ satisfies the neighbor-swap rule (7.17), using } c \in \mathbf{o}P \text{ and (8.2)} \;\} \\
& \text{`}\{b\} \text{ is not a refusal at } scat \in \mathbf{t}P\text{'} \\
\equiv & \quad \{\; P \text{ has only trivial refusals} \;\} \\
& scatb \in \mathbf{t}P
\end{array}$$

Next we derive

$$\begin{array}{ll}
& scat \in \nabla P \;\wedge\; sact \in \mathbf{t}P \\
\Rightarrow & \quad \{\; P \text{ is a process (requirement 5)} \;\} \\
& (\exists\, d : d \in \mathbf{o}P : scatd \in \mathbf{t}P \;\wedge\; sact \in \mathbf{t}P) \\
\Rightarrow & \quad \{\; P \text{ satisfies Rule } \mathcal{Y}, \text{ using } a \not\approx c \text{ and } c \approx d \;\} \\
& (\exists\, d : d \in \mathbf{o}P : sactd \in \mathbf{t}P) \\
\Rightarrow & \quad \{\; P \text{ is maximally transient} \;\} \\
& sact \in \nabla P
\end{array}$$

This concludes the proof that P satisfies Rule $\mathcal{Y}^{out}$. ∎

In the DI Model, failure to satisfy Rule $\mathcal{Y}^{out}$ is the shadow cast by lack of progress, which is not otherwise noticeable in that setting. The Extended DI Model treats progress more explicitly. The invalid implication mentioned below Example 6.3.3 can now be qualified.

8.1.10 Theorem Process $P \in \mathcal{DI}_\alpha$ satisfies Rule $\mathcal{Y}^{out}_\alpha$ if and only if $\varphi_\nabla.P \in \mathcal{DI}_\beta$. ∎

8.2 Static versus Dynamic Output Nondeterminism

We proceed with a classification of output nondeterminism, something which could not be done properly for output choice in the DI Model. Let P be a DI process that is *not* output deterministic. We say that P has **static output nondeterminism** when there exists an output-deterministic DI process Q with $Q \sqsupseteq P$, that is, when P has an

output-deterministic implementation. Otherwise, we say that P has **dynamic output nondeterminism**, that is, when P has no output-deterministic implementation.

The undetermined selector has static output nondeterminism, since it is refined by a toggle. The arbiter and the sequencer have dynamic output nondeterminism, which will be explained in the next example.

8.2.1 Example Figure 8.4 shows the labeled DI state graphs of processes P and Q. When P has received a single input, it must respond with the corresponding output. This is also the case for Q. Upon receiving both inputs, however, there is no obligation to produce output ($\Box$ in state 2) and P may produce either of the two outputs but not both, while Q may also produce both. Observe that we have $P \sqsupseteq Q$.

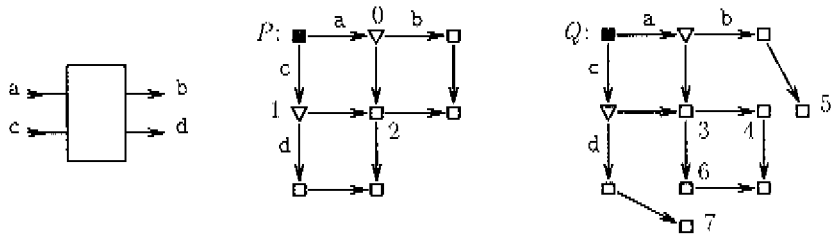


Figure 8.4: State graphs of P (middle) and Q (right)

The state graphs of the arbiter and the sequencer (after input n) contain subgraphs similar to that of P, except that they are labeled ∇ in state 2. Of course, if process P (with $\Box$ in state 2) has no output-deterministic implementations, then neither has the process with ∇ in state 2 (any implementation of the latter would also be an implementation of the former).

We claim that P and Q have *dynamic* output nondeterminism. Obviously, P is not output deterministic, because it is neither maximally transient nor does it satisfy Rule $\mathcal{Z}^{out}$ (check state 2). Q is not output deterministic either, since it is not maximally transient (states 3, 4, and 6), though it does satisfy $\mathcal{Z}^{out}$. Note that P satisfies $\mathcal{Y}^{out}$ but Q does not.

There are essentially two methods of making processes more output deterministic.

1. In a state where output is possible but not required (that is, where progress is not maximal), one may either (a) suppress some outputs, or (b) guarantee output by making the state transient.

2. In a transient state requiring a choice between outputs, one may eliminate all but one of the mutually exclusive outputs.

However, one should also see to it that the result is still a DI process.

The only "output-nondeterministic" states of Q are states 3, 4, and 6, where progress is not maximal. We first focus on states 4 and 6. Only the first method applies here. These states cannot be made transient (method 1b) because that would violate Rule $\mathcal{Y}$

(since states 5 and 7 are indifferent). There remains method 1a: suppressing the outputs in states 4 and 6. Doing so yields process P.

The only "output-nondeterministic" state of P is state 2. Again only the first method applies. Output elimination (1a) does not work, because of Rule $\mathcal{Z}$ and the fact that states 0 and 1 are transient. Output guaranteeing (1b) does apply. Doing so yields a process that is still not output deterministic (it is maximally transient but does not satisfy $\mathcal{Z}^{out}$). Now only the second method might be of help. However, neither of the outputs in state 2 can be eliminated as was already pointed out. Therefore, P and Q have no output-deterministic implementation and, hence, their output nondeterminism is dynamic. ∎

The existence—in the Extended DI Model—of processes with dynamic output nondeterminism contrasts with CSP, where every process has deterministic implementations.

Static output nondeterminism can be eliminated by the designer. The nondeterminism is then resolved at *design-time*. Dynamic output nondeterminism can only be resolved at *run-time*. It is intertwined with the *behavior* of the environment, hence the name. Whether or not such nondeterminism is actually encountered during operation depends on the interaction between the process and its environment. For instance, an arbiter's operation is deterministic as long as no more than one request is made at a time. Nondeterminism only plays a role when two requests are made "simultaneously". It cannot be resolved at design-time precisely because simultaneity is not a well-definable concept in the DI Models.

The **refinement closure** of process set V is defined as the process set

$$\{P, Q : P \in V \wedge P \sqsupseteq Q : Q\}\ , \tag{8.6}$$

obtained from V by adding all processes refined by processes of V. The refinement closure of the set of output deterministic DI processes will be denoted by $\mathcal{RCOD}$. It consists of the DI processes that are output deterministic and the ones that have static output nondeterminism. That we need building blocks outside $\mathcal{RCOD}$ for implementing dynamic output nondeterminism, follows from the fact that $\mathcal{RCOD}$ is closed under refinement and composition. The latter is a consequence of Theorem 8.1.8 and the next theorem.

8.2.2 Theorem The refinement closure of a composition closed set of DI processes is itself composition closed.

Proof Let V be a composition closed set of DI processes and let W be its refinement closure. For processes P and Q we derive

$$\begin{array}{ll}
& P \in W \wedge Q \in W \\
\equiv & \quad \{\ W \text{ is refinement closure of } V\ \} \\
& (\exists\, P', Q' : P' \in V \wedge Q' \in V : P' \sqsupseteq P \wedge Q' \sqsupseteq Q) \\
\Rightarrow & \quad \{\ V \text{ is composition closed and composition is } \sqsupseteq\text{-monotonic (Theorem 4.9.2)}\ \} \\
& (\exists\, P', Q' : P' \parallel Q' \in V : P' \parallel Q' \sqsupseteq P \parallel Q) \\
\equiv & \quad \{\ W \text{ is refinement closure of } V\ \} \\
& P \parallel Q \in W
\end{array}$$

This shows that W is also composition closed. ∎

The following two theorems give sufficient conditions for a process to be in $\mathcal{RCOD}$. The proof of the first theorem relies solely on method 1b mentioned above for making a process more output deterministic, whereas that of the second theorem uses methods 1a and 2 only.

8.2.3 Theorem If DI process P satisfies Rules $\mathcal{Y}^{out}$ and $\mathcal{Z}^{out}$, then it is in $\mathcal{RCOD}$.

Proof Consider process $Q = \varphi_{\nabla}.(\psi.P)$. Q is obtained from P by changing each trace where output is enabled into a transient trace and by changing all remaining traces into indifferent traces. On account of Theorem 8.1.10, using that Q satisfies $\mathcal{Y}^{out}$, we have $Q \in \mathcal{DI}$. By construction, Q implements P ($Q \sqsupseteq P$), Q is maximally progressive, and Q satisfies Rule $\mathcal{Z}^{out}$. Thus, Q is an output-deterministic implementation of P, which shows $P \in \mathcal{RCOD}$. ∎

That the converse does not hold is shown by the undetermined selector, which is in $\mathcal{RCOD}$ but does not satisfy $\mathcal{Z}^{out}$.

For process P, we say that refusal $A \subseteq \mathbf{o}P$ at trace $ta \in \mathbf{t}P$ propagates backward over input $a \in \mathbf{i}P$ when A is also a refusal at t. In general, this need not be the case, as is exemplified by process P_2 of Example 6.1.3, for which $\{\mathtt{b}\}$ is a refusal at $\mathtt{a}$ but not at ε. Note that P_2 has static output nondeterminism on account of the preceding theorem (indeed, P_3 of Example 6.1.3 is an output-deterministic implementation). It is easy to verify that the refusals of an output-deterministic process propagate backward over all inputs.

8.2.4 Theorem If the refusals of DI process P propagate backward over all inputs, then P is in $\mathcal{RCOD}$.

Proof We will construct an output-deterministic implementation of P. The construction is by induction on the number n of outputs $a \in \mathbf{o}P$ such that $\{a\}$ is a non-trivial refusal at some trace $t \in \mathbf{t}P$. Bear in mind that the output alphabet of P is finite. If $n = 0$ then P is output deterministic on account of Theorem 8.1.5. Assuming $n > 0$ we show how P can be refined while reducing n.

Let $\{a\} \subseteq \mathbf{o}P$ be a non-trivial refusal at $t \in \mathbf{t}P$ and, hence, $ta \in \mathbf{t}P$. Consider process Q obtained from P by removing all traces $uav \in \mathbf{t}P$ for which $\{a\}$ is a refusal at u, that is, we have

$$Q \;=\; (\; \mathbf{i}P,\; \mathbf{o}P,\; \nabla P \setminus V,\; \Box P \setminus V,\; \Delta P \setminus V \;)\;, \tag{8.7}$$

where $V = \{u, v : uav \in \mathbf{t}P \wedge \text{`}\{a\} \text{ is a refusal at } u\text{'} : uav\}$. We claim that Q is a DI process and that "its n" has decreased. That Q is indeed a process we infer from the fact that if $u \in \nabla P$ and $\{a\}$ is a non-trivial refusal at u, then there is another output b with $ub \in \mathbf{t}P$ (for otherwise $\{a\}$ would not be refusal at u). Q satisfies Rule $\mathcal{W}$ because no traces have been added. Concerning Rules $\mathcal{X}$ and $\mathcal{Y}$, observe that for traces s and u, and symbols b and c such that $b \in \mathbf{o}P \vee c \in \mathbf{i}P$, we have

'A is refusal at $sbct$'

$\Rightarrow$ { P satisfies neighbor-swap rule 7.17, using $b \in \mathbf{o}P \vee c \in \mathbf{i}P$ and (8.2) }

'A is refusal at $scbt$'

Rule Z is satisfied precisely because refusals propagate backward over inputs: if for input b, traces $sbav \in \mathbf{t}P$ have been removed (because $\{a\}$ is a refusal at sb), then also traces sav will have been removed (because $\{a\}$ is also a refusal at s). Consequently, Q is DI. By construction, if $\{a\}$ is a refusal at $u \in \mathbf{t}Q$, then it is trivial. Trivial refusals in P are also trivial in Q. Therefore, the number of outputs a such that $\{a\}$ is a non-trivial refusal in Q is less than n. (N.B. The number may have gone down by more than one. Undetermined selector $\mathtt{U(a;b,c)}$ has both $\{\mathtt{a}\}$ and $\{\mathtt{b}\}$ as non-trivial refusals at ε. Eliminating output $\mathtt{a}$, removes refusal $\{\mathtt{b}\}$ altogether and makes refusal $\{\mathtt{a}\}$ trivial.) ∎

Again the converse does not hold, as shown by P_2 above. However, we conjecture that the set $\mathcal{RPBI}$, consisting of all DI processes such that their refusals propagate backward over all inputs, is closed under composition. Thus, in order to construct a process *equivalent* to P_2 above, at least one process outside $\mathcal{RPBI}$ is needed. An arbiter would suffice, but also lies outside $\mathcal{RCOD}$ and, hence, seems more than is asked for. Is there a useful building block in $\mathcal{RCOD} \setminus \mathcal{RPBI}$?

It would be interesting to have a characterization---comparable to Theorem 8.1.6 for output determinism---of the dividing line between static and dynamic output nondeterminism.

8.3 Closure Results

What could be more appropriate than to close the theoretical developments with a table of closure results? Given a set of DI processes, one might wonder whether it is closed under refinement, composition, and reflection. Table 8.3 summarizes these closure properties for many of the DI process sets that we have considered.

DI process set	$\sqsupseteq$-closed	$\|$-closed	$\rightsquigarrow$-closed
$\mathcal{DI}$	yes	yes	yes
$\#\mathbf{i}P = \#\mathbf{o}P$	yes	yes	yes
$\#\mathbf{i}P \geqslant \#\mathbf{o}P$	yes	yes	no
$\#\mathbf{i}P \leqslant \#\mathbf{o}P$	yes	yes	no
passive	yes	yes	no
finite state	no	yes	yes
$\mathcal{Y}'$	no	no	yes
$\mathcal{Z}'$	no	no	yes
$\mathcal{Z}^{out}$	no	no	no
$\mathcal{Y}^{out} \wedge \mathcal{Z}^{out}$	no	α: yes (β?)	no
indifferent	no	yes	yes
progressive	no	no	no
deterministic	no	yes	no
$\mathcal{RCOD}$	yes	yes	no
$\mathcal{RPBI}$	no	(yes?)	no

Table 8.3: Closure results for some DI process sets

Chapter 9

Conclusion

We now look back at our work, evaluate it, and relate it to the work of others. We also discuss some practical aspects of delay-insensitive systems that have been ignored in the preceding chapters. Along the way we suggest interesting topics for further investigation.

9.1 Retrospect

In Chapter 2 we pointed out that the timing problem is a fundamental issue in the design of digital circuitry. Delay-insensitivity is a solution to the timing problem. In the ideal case, the correctness of a delay-insensitive circuit is completely independent of delay assumptions. However, this is not practically feasible. More realistic is the two-stage solution, where circuits are designed as networks of building blocks. The timing problem is confined to the building blocks: their correct internal operation may depend on timing, but the correctness of their cooperation is independent of delays. The design of building blocks is considered a separate topic (also see Section 9.4 below). Therefore, we have restricted ourselves to specifications that are free of a time metric, that is, in which time plays a role for sequencing only.

Our starting point for a model of delay-insensitive systems is a set $\mathcal{PROC}$ of processes. These processes act as specifications for building blocks and systems. In the DI Model of Chapter 4, a process is characterized by a triple (I, O, V), where I and O are finite sets of symbols (representing the communication ports: I for inputs, O for outputs) and V is a non-empty prefix-closed set of finite-length symbol sequences (representing the allowed orders for communication events). Finite symbol sets are called alphabets, and finite-length symbol sequences are called traces. A network of processes is modeled as a set of processes satisfying certain structural conditions, capturing that there are only point-to-point connections from output ports to input ports. Such sets of processes are called systems. The set $\mathcal{SYS}$ of systems can be viewed as generalizing $\mathcal{PROC}$. By convention, output a of process P in system S is connected to input a of process Q in S; that is, connections are modeled by common symbols. These common symbols are considered dummies and may be renamed consistently.

The connection of two systems into a larger system is modeled by composition operator *par* on $\mathcal{SYS}$. Operator *par* is simply set union, after appropriate renaming of internal connections to avoid name clashes. The operation of a closed system (without external ports) gives rise to reachable traces and interfering traces. A system is called free of interference when no reachable trace is interfering. Interference arises when a process is sent input that is not acceptable. Predicate *Correct* captures autonomous correctness of systems, and requires that the system be well-defined, closed, and free of interference. For systems S and T, predicate $Correct.(S\ par\ T)$ can be interpreted as expressing that S passes test T. In order to compare systems, the satisfaction relation *sat* is introduced. We say that S is a satisfactory substitute, denoted by $S\ sat\ T$, when every test that T passes is also passed by S, or in a formula:

$$S\ sat\ T \quad \equiv \quad (\forall\, U : U \in \mathcal{SYS} : Correct.(S\ par\ U) \Leftarrow Correct.(T\ par\ U)) \ . \tag{9.1}$$

A system containing T can always be written as $T\ par\ U$ for some suitable U and, hence, $S\ sat\ T$ holds when in every correct system containing T, T can be replaced by S without destroying the correctness. Relation *sat* is a testing pre-order.

Thus we obtained model $\langle \mathcal{SYS}; par, sat \rangle$, in which we can talk about systems and their composition and comparison. A typical design problem might be formulated as follows. Given systems T and U find system S such that

$$S\ par\ T \quad sat \quad U \ . \tag{9.2}$$

Often, U will be a singleton system, consisting of just one process, and T will be taken from a prescribed subset $\mathcal{BB}$ of $\mathcal{PROC}$. Thus, U is the overall specification, T describes part of a design in terms of building blocks, and S is what is needed to complete the design. We refer to (9.2) as the design equation.

Systems S and T are equivalent, denoted by $S\ equ\ T$, when they are satisfactory substitutes for each other. Since equivalence is not the same as equality, $\langle \mathcal{SYS}; par, sat \rangle$ is a pre-abstract model. Because *equ* is compatible with both *par* and *sat*, we can consider the quotient $\langle \mathcal{SYS}; par, sat \rangle / equ$, which is a fully abstract model. The quotient turns out to be isomorphic to $\langle \mathcal{DI}; \|, \sqsupseteq \rangle$, where $\mathcal{DI}$ is an appropriate subset of $\mathcal{PROC}$, $\|$ derives from a binary operator on $\mathcal{PROC}$, and $\sqsupseteq$ derives from a partial order on $\mathcal{PROC}$. The subset $\mathcal{DI}$ can be characterized by the so-called JTU-Rules, and $\sqsupseteq$ is easily expressible in terms of alphabets and trace sets. However, composition $\|$ is more complicated. Within the fully abstract model based on $\mathcal{DI}$, the design equation can be solved, since we have

$$P \,\|\, Q \sqsupseteq R \quad \equiv \quad P \sqsupseteq \backsim(Q \,\|\, \backsim R) \ . \tag{9.3}$$

That is, the least solution of the equation in P on the left is obtained as $\backsim(Q \,\|\, \backsim R)$, where $\backsim$ is a unary operator on $\mathcal{PROC}$ called reflection. Reflection interchanges the roles of input and output, but does not affect the trace set. Note that composition $\|$ has no inverse and, hence, the equation $P \,\|\, Q = R$ in P is not always solvable given Q and R. That the design equation is solvable is, in fact, a fortunate consequence of our choice for $\mathcal{PROC}$. Apparently, $\mathcal{PROC}$ is sufficiently large (or small) to allow a reflection operator.

This is, for instance, not the case for the Failures Model of CSP. It would be interesting to study the extension of process spaces to incorporate the minimal solutions of the design equation (also see [Pra91]).

In search of a suitable set of building blocks, the possibilities of a number of elementary processes are investigated in Chapter 5. In particular, we have identified several subsets of processes that are closed under composition. In order to implement processes outside such a closed set, a building block outside that set is required. It was pointed out that a satisfactory set of building blocks to realize all finite-state processes is still not available. Also the three classes around the C-element, latch, and decision-wait have not been characterized satisfactorily. It should be noted that there are actually two kinds of search problems. The first asks for a (minimal) set of building blocks that suffices to *implement* all processes within a certain set. The second asks for a (minimal) set of building blocks that suffices to construct an *equivalent* of each process within a certain set. The first is more practical, the latter has mainly theoretical importance.

The DI Model has several shortcomings, one of them being that progress is not a correctness concern. The Extended DI Model of Chapter 6 incorporates a progress concern. It is constructed along the same lines as the DI Model of Chapter 4. The trace set of a process P is now divided into three trace sets, comprising the transient (∇P), the indifferent ($\Box P$), and the input-demanding (ΔP) traces. The distinction between the first two is that in a transient trace the process is obliged to produce some output, whereas in an indifferent trace it may fail to do so. Input-demanding traces are similar to indifferent traces as far as output production is concerned, but the environment is obliged to provide some input. These can be viewed as arising from the reflection of transient traces (similar to negative money being money owed). A process space that is closed under reflection is desirable for the existence of the minimal solution of the design equation.

The fully abstract version of the Extended DI Model is very similar to that of the DI Model. The JTU-Rules and the partial order $\sqsubseteq$ are easily extended. The whole structure can be better understood when processes are represented by enhanced characteristic functions (ECFs) mapping traces to the trace labels $\top$, ∇, $\Box$, Δ, and $\bot$. A simple algebra on the trace labels captures composition and correctness, and generates definitions for order $\sqsubseteq$ and reflection $\backsim$. The order and reflection are then lifted to ECFs. ECFs allow a concise formulation of the JTU-Rules, in the form of the neighbor-swap rule. Normalizing transformations on ECFs can be used to compute compositions and greatest lower bounds.

Chapter 8 continues the investigation started in Chapter 5. In particular, it focuses on output (non)determinism. Inspired by the Failures Model for CSP, we define output refusal sets. Alphabet A is an output refusal set at trace t in process P when, after doing t, P can do some outputs not in A and arrive in a state where it has no further output obligations. Output refusal set A at trace t is called trivial, when the process is unable to do any of the outputs in A. A process is said to be output deterministic when it has only trivial output refusal sets. The output-deterministic DI processes are exactly the maximally transient DI processes satisfying Rule $\mathcal{Z}^{out}$ (no output choice). The set of output-deterministic DI processes is closed under composition. However, not every DI process has an output-deterministic DI implementation (in CSP every process has a deter-

ministic implementation). Examples are the arbiter and sequencer. They are said to exhibit *dynamic* output nondeterminism. This contrasts with the *static* output nondeterminism of the undetermined selector, which can be eliminated in an implementation. Dynamic output nondeterminism cannot be eliminated at design time, because it is intertwined with the behavior of the environment. We have not encountered the distinction between static and dynamic nondeterminism elsewhere. There is a vague resemblance with the notion of *confusion* in the theory of Petri nets (see [Rei85]). The complete characterization of dynamic output nondeterminism is still an open problem.

9.2 Evaluation

Our development of a theory for delay-insensitive systems involved several fundamental decisions. In some cases the consequences of our decisions are pleasing; in other cases we still have reservations. First we discuss some items on the positive side of the scales.

We like the line of development that starts with a pre-abstract model and from there continues to a fully abstract model, possibly culminating in an axiomatic model. This approach enables one to set up a formal model without knowing in advance what is needed for a fully abstract model. Furthermore, it may simplify the link to other models, such as continuous physical models.

We are also satisfied with the use of the testing paradigm to set up pre-abstract models. The testing paradigm may be viewed as a primitive but powerful method to define observations on processes. A requirement is that the set of tests is sufficiently rich. Our desire to include the tests in the set of processes led us to distinguish indifferent and input-demanding traces in the Extended DI Model. We could, in fact, have started with maximally transient processes and a separate set of maximally input-demanding tests. There are, however, some technical complications with such a separation. For instance, the proof that *equ* is a congruence relation with respect to *par* would no longer work as shown in Appendix B. Nevertheless, the resulting fully abstract model would then have processes with a mixture of transient and indifferent traces, but it would not be closed under reflection. Later we found out that this means that the minimal solution of the design equation need not exist in the model. Thus it was fortunate to insist on one set for both processes and tests.

We consider the development of the Extended DI Model—in particular, the definitions of deadlock, the partial order $\sqsubseteq$, and the extended JTU-Rules—as rather elegant. The introduction of enhanced characteristic functions (ECFs), which split the complement of the trace set symmetrically, is very useful. ECFs shed more light on the mathematical underpinnings of the DI Models. Furthermore, it is intriguing that so many properties can be lifted from a small algebra on trace labels to ECFs. Also the fact that three JTU-Rules can be combined into the neighbor-swap rule is a nice result. Finally, the treatment of output (non)determinism brought up several interesting points.

Let us now mention some items that we feel are on the negative side of the scales. The treatment of system structure has been quite ad hoc. Composition operator *par* is a partial

operator and involves renaming of dummies (internal connections). It is in general not associative. Moreover, it is not immediately clear how to generalize it when other connection rules apply (such as an output being able to drive at most three inputs). Many definitions and theorems involve conditions that deal with system structure (such as systems being connectable), though we have not always made them completely explicit. One example is the definition of *sat* (see Equation 9.1 above), which involves a quantification over all systems U, including those U for which S *par* U, or T *par* U, is not a system. We have postulated that *Correct* does not hold in those cases, but this is hardly a satisfactory solution. An alternative approach (see [Ver94]) is to define systems as *bags* of processes and to make the names of internal connections visible on the outside (though not the internal communication events). This way, system structure can be dealt with by testing as well. We can introduce mappings, similar to ECFs, from the symbol universe to symbol labels (also see Note 7.3.12).

We would have preferred to analyze isochronic operation first, as suggested in Note 4.4.3. We have not done so because this would further increase the notational complexity and slow down the pace. It is also felt as a drawback that isochronic connectors and forks cannot be handled. A way to overcome this is to take *sets* of symbols as atomic events. Symbols within such an event are required to "happen at the same step". An isochronic fork with input a and two outputs b and c might then be described as an alternation of events $\{a\}$ and $\{b, c\}$.

ECFs are a neat way of describing processes and their cooperation in systems. However, the tediousness and sheer number of all details concerning ECFs was disappointing. That is why we have not presented the theory in terms of ECFs from the very beginning.

The Extended DI Model improves the DI Model, but it also has its shortcomings. Further extensions of the models might include infinite systems (as limits of unbounded sets of finite systems) and infinite traces (to deal with livelock). In both cases nasty technical problems lurk ahead. For instance, if we have $P_i \sqsupseteq Q_i$ for all natural i, do we then also have

$$\{i : 0 \leqslant i : P_i\} \quad sat \quad \{i : 0 \leqslant i : Q_i\} \,, \tag{9.4}$$

that is, how do we deal with infinite substitutions?

We would have liked to give axiomatic characterizations of our models, but the amount of work involved was prohibitive. The groundwork has been laid in the form of properties relating composition, refinement, greatest lower bounds, reflection, and 'aftering' (for instance, the Factorization Theorem). See the discussion of the DI Algebra below for other work in this direction.

9.3 Related Work

The framework of the DI Models is entirely my own. It is based on the testing paradigm, which, in fact, I discovered myself, but I later also traced it to [dNH83] and even to Leibniz (see opening quotation). The following, necessarily concise, overview indicates

the relationship to those theoretical works that influenced me most. For a comprehensive bibliography see [Pee].

The idea to characterize a process by its alphabet (communication ports) and trace set (allowed sequences of communication events) derives from Trace Theory introduced in [vdS85] and further developed in [Kal86]. The idea to model delay-insensitive circuits in this context is already mentioned in [vdS85]. The application of Trace Theory to the description and design of delay-insensitive circuits goes back to [Udd84, Ebe89]. For that purpose, they split the alphabet into inputs and outputs.

In [Udd84], the starting point is the set $\mathcal{DI}$ of delay-insensitive (DI) processes, defined in terms of the JTU-Rules, as we have called them. Two of the main accomplishments of that work are the formulation of the JTU-Rules and the proof that for $P \in \mathcal{DI}$, the (wired) system $\{P, \backsim P\}^{\frown}$ is free of interference. However, systems and their operation are not formalized in general, and the composition operator, appearing in the form of the *blending* operator, has a very restricted range of applicability, involving the notion of *independent alphabets*. There is also no satisfaction relation on processes. Three subsets of DI processes are distinguished, namely

$$
\begin{aligned}
C_1 &= \{P : P \in \mathcal{DI} \wedge \mathrm{P} \text{ satisfies Rules } \mathcal{Y}' \text{ and } \mathcal{Z}' : P\} \ , \\
C_2 &= \{P : P \in \mathcal{DI} \wedge \mathrm{P} \text{ satisfies Rules } \mathcal{Y}' \text{ and } \mathcal{Z}^{out} : P\} \ , \\
C_3 &= \{P : P \in \mathcal{DI} \wedge \mathrm{P} \text{ satisfies Rule } \mathcal{Y}' : P\} \ .
\end{aligned}
$$

It is shown that classes C_1 and C_2 are closed under the restrictive form of composition, whereas class C_3 is not closed. Note that under our more general form of composition neither of these classes is closed (see Example 5.3.1). Instead, we have pointed out that the set of DI processes satisfying Rules $\mathcal{Y}^{out}$ and $\mathcal{Z}^{out}$ is closed under general composition (see Theorem 5.3.3). Progress is not covered.

In [Ebe89], isochronic system operation, appearing in the form of the *weaving* operator, is fundamental. Connecting wires need not be introduced explicitly when all processes are taken to be DI (compare our Theorem 4.7.7). Composition of processes is not defined as an operator, but there is a refinement relation, going by the name of *decomposition* relation. It expresses when one system is a decomposition (satisfactory substitute) of another system. The decomposition relation resembles the satisfaction relation *sat* of our DI Model. However, there are some subtle differences. It is recognized in [Ebe89] that *sat* allows obviously undesirable implementations (see our Examples 5.5.1 and 5.5.2), which can be ascribed to lack of progress. For that reason, the decomposition relation requires the implementation to be able to produce *exactly* the same outputs as the specification. This excludes toggle $\mathsf{T(a; b, c)}$ as decomposition of fork $\mathsf{F(a; b, c)}$, which in the DI Model is a proper refinement. However, there are also undesirable consequences. On the one hand, toggle $\mathsf{T(a; b, c)}$ is not considered a decomposition of undetermined selector $\mathsf{U(a; b, c)}$. On the other hand, system D of Example 5.5.1 is considered a decomposition of a wire. Thus, decomposition does not treat progress satisfactorily. These issues are discussed in greater depth in [Pee90]. Specifications in [Ebe89] are expressed mostly in terms of *regular expressions* extended with a few extra operators. The extensions allow some nice

design techniques (also see [Ebe88]). It is proved, by construction, that a large class of specifications can be decomposed efficiently using a few simple building blocks (though the one-all of Example 5.4.1 requires an arbiter).

The following three works have contributed to the development of our DI Model. In [UV88], the partial order $\sqsubseteq$ on processes is introduced and studied in the context of closed two-process systems. In [CUV89b], general systems are analyzed. It also discusses the design equation and its minimal solution. Most of the proofs are carried out by induction on the reachability of traces. In [Ver89], enhanced characteristic functions are introduced and used to prove the equivalence of 'P satisfies the JTU-Rules' and 'wired system $\{P, \backsim P\}^{\sim}$ is free of interference'.

In [Dil89], processes are characterized by their input and output alphabets together with *two* trace sets. The *canonical process description* (I, O, S, F) of [Dil89], corresponds to our process (I, O, S) in the DI Model. Trace set F consists of the traces mapped to $\perp$ (interfering) by the enhanced characteristic function of the process. System operation is not explicitly formalized, but a composition operator on processes is given. The main reason for introducing the more general processes with two trace sets is to simplify the definition of composition. This can be compared to our larger process space $\mathcal{ECF}_{1,3,5}$ of Chapter 7, which is closed under tracewise composition and taking of greatest lower bounds. A refinement relation, called *conformation*, is defined. It corresponds directly to our satisfaction relation *sat* in the DI Model. A major contribution of [Dil89] is the development and implementation of a tool to verify delay-insensitive designs. Specifications and designs are described in terms of state graphs, for which a LISP front-end is provided as well. An attempt is also made to deal with progress, but we do not consider it very successful.

In [Jos92], a process model is introduced that distinguishes input and output, and that also captures a progress concern. A process has an input and an output alphabet, and a trace set. The trace set is not necessarily prefix-closed and consists of so-called *failures* (traces in which the process may fail to produce output). These traces correspond to the indifferent traces in our Extended DI Model. Two requirements on processes are crucial to the modeling of interference: output alphabet O is non-empty and the prefix-closure of trace set F is *receptive*, that is, closed under input extension. Input a after trace t now leads to interference when the set

$$\{u : u \in O^* \wedge tau \in F : t\}$$

is infinite. Thus, the prefix-closure of trace set F consists of what we have called the allowed and the interfered traces (those not mapped on $\top$ by the enhanced characteristic function of the process). We find the requirement of non-empty output alphabets somewhat contrived. In [Jos92], it is observed that, due to the input-output distinction, the refusal sets of the Failures Model for CSP have been "simplified out of existence". Composition involves isochronic operation, though the latter is not defined separately. Additional closure conditions may be imposed on processes to deal with anisochronic operation. These conditions involve the reordering of symbols in traces in the same vein as $\supseteq$ at the end of

Section 7.2, but this is not related to the JTU-Rules. The model has a refinement relation based on inclusion of trace sets (failures). It corresponds to the satisfaction relation *sat* of our Extended DI Model, but there are two major differences between the process model of [Jos92] and the Extended DI Model. First of all, there is no reflection operator on the receptive processes. Consequently, the design equation cannot always be solved within that model, the main reason being that input demand is not expressible. Secondly, livelock (unbounded internal communication and non-terminating recursion) is modeled to coincide with interference. We find this view of livelock too pessimistic, though we agree that ignoring livelock, as we do in the DI Models, is too optimistic.

In [JU90, JU93], an algebra for the specification and design of delay-insensitive circuits is described, called the *DI Algebra*. It is closely related to [Jos92]. The DI Algebra can be viewed as a *language* in which processes are specified by expressions constructed from constants and operators, such as *guarded choice* and *recursion*. Process equivalence and refinement are defined axiomatically by a number of *laws*. These laws capture interference and also a progress concern. It should be noted that the syntax and the set of laws have not yet fully stabilized. A major advantage of the DI Algebra is that design verification and, in particular, process composition can be done by *calculation*. In [Luc94], several useful meta-theorems are presented. However, the DI Algebra has its shortcomings as well. There are few (syntactic) *heuristics* for design. Some processes are easy to specify but others are relatively hard to specify, such as larger decision-waits and data converters. Compare also processes P_5 and Q_5 of Example 3.2.5, whose "progressive" counterparts in the DI Algebra of [JU90] are given by

$$\begin{aligned} P_5 &= a?; b?; (d!; [a? \rightarrow \bot \,\Box\, b? \rightarrow \bot \,\Box\, skip \rightarrow e!; P_5] \sqcap \\ &\qquad e!; [a? \rightarrow \bot \,\Box\, b? \rightarrow \bot \,\Box\, skip \rightarrow d!; P_5]\ ) , \end{aligned}$$

$$Q_5 = a?; b?; d!; e!; Q_5 .$$

The introduction of output guards and 'else' clauses reduces the complexity of P_5 a little. It can also be excessively difficult to prove *inequality* of processes. See the discussion of [Jos92] above, for model distinctions concerning livelock and reflection. It would be interesting to extend the DI Algebra with a reflection operator; [Luc94] reports on work in this direction. The problems with the introduction of a reflection operator may be explained as follows in terms of enhanced characteristic functions (ECFs) (see Chapter 7, especially Note 7.3.12). In a sense, DI Algebra expressions correspond to ECFs satisfying predicates $\mathcal{E}_1$, $\mathcal{E}_3$, and $\mathcal{E}_5$. The intended *normal form* of expression E is $\lfloor f \rfloor$, where f is the ECF corresponding to E. The (normalizing) laws correspond, in a way, to our transformations Φ_i. For instance, the law $c!; \bot = \bot$ resembles Φ_4. Unfortunately, the space spanned by $\mathcal{E}_1$, $\mathcal{E}_3$, and $\mathcal{E}_5$ is not closed under reflection. Furthermore, reflection and $\lfloor . \rfloor$ do not commute (see Equation 7.25). Therefore, reflection is only easy to define for near-normal forms, which also satisfy the even $\mathcal{E}_i$ and which are invariant under $\lfloor . \rfloor$. A different approach is required to generalize reflection.

9.4 Towards Circuits

To complement Chapter 2, we now discuss some practical aspects of delay-insensitivity when applied to the design of digital integrated circuits.

Our main concern has been the design of delay-insensitive systems using some set of building blocks. The realization of the building blocks has been considered a separate problem, falling outside the scope of our theory. The advantage of this two-stage approach is that, on the level of systems of building blocks, the correctness of these systems does not depend on assumptions about delays. However, building blocks need to be built to get a working system.

For instance, an integrated circuit can be produced from such a design only if all the building blocks have been realized in terms of transistor networks. When designing these building blocks, we encounter the timing problem again. More detailed models of circuit operation are required to design them and prove their implementation correct, though this needs to be done only once for each building block. For approaches to the design of building blocks we refer to [MFR85, RMCF88, vB92]. The relationship between the (discrete) DI Models and (continuous) physical circuit models has been largely ignored and needs to be pursued more seriously.

Another issue that has been ignored is the initialization of the building blocks at power up. Somehow each state-holding device, such as the C-element and toggle, needs to be put into its proper initial state. For some building blocks it is possible to come up with an implementation whose initialization is accomplished by applying appropriate voltage levels to the inputs. For example, a C-element, which is a sequential circuit, can be implemented in such a way that when its inputs are forced low, the output will go low as well and the C-element ends up in a uniquely determined state. This is not possible for the toggle. The I-wire poses a related problem. It is supposed to produce a transition after power up. Often this just means that the component it is connected to requires a slightly different initial state. But if, for example, an I-wire is driving a C-element, the property of "automatic" initialization when low inputs are applied no longer holds for the combination, and it may end up in the wrong initial state.

Realistic circuit design methodologies also require an answer to the "testing problem". Even when a circuit is fabricated according to a correct design, there may still be variations in the final product due to the inherently stochastic nature of the manufacturing processes. Too large variations may yield unreliable or malfunctioning circuits, which should be eliminated as soon as possible. The standard approach to the detection of fatal fabrication faults is testing, that is, operating the product according to some predefined input sequences for which the expected outputs are known. The "testing problem" is that of finding a small set of input sequences such that from the corresponding outputs a reasonable estimate about the circuit's reliability can be made. For clocked circuits it is easier to control and observe the internal state of the circuit; this may result in simpler tests. On the other hand, in delay-insensitive circuits, which are based on transition signaling, it suffices to test whether each wire can make transitions in both direction (assuming the stuck-at fault model). Testing of delay-insensitive circuits is still an area of active research. Especially

the "VLSI-programming" approach to delay-insensitive circuit design looks promising in this respect [RS93].

It might be interesting to search for a link between fault testing and the satisfaction relation based on the testing paradigm. A high-level fault model that comes to mind is that where the effect of a fault is to change a ∇-label into a $\Box$-label. Usually, such a fault will change an implementation into a non-implementation; this is then detectable by a specific testing environment.

The topic of manipulating data (as opposed to control signals) in delay-insensitive circuits – think, for instance, of computations involving integers – has been not been addressed here. Encoding of data for delay-insensitive transmission is treated in [Ver88].

Performance

The average speed of delay-insensitive circuits is determined by the *average case*, whereas the clock of synchronous circuits must be tuned to the *worst case*. Therefore, one would expect delay-insensitive circuits to "run faster" than their clocked counterparts. In practice, however, delay-insensitive circuits do not perform that well. One reason for this is that the design assumptions are very pessimistic, requiring extra communication actions to ensure proper synchronization of interacting subcomputations, usually via some back-and-forth handshake protocol.

This pessimistic approach is also responsible for the additional area that current delay-insensitive circuits require. For instance, encoding n-bit values with a double-rail code requires $2n$ wires. It is expected that further research will enable us to make faster and smaller delay-insensitive circuits in the future.

On the positive side, the circuits designed with such pessimistic assumptions are quite robust. For example, they can tolerate considerable variations in power supply voltage and ambient temperature.

Another beneficial property of delay-insensitive circuits is that they dissipate little power when implemented in CMOS technology. A CMOS transistor dissipates power only when it switches. In a delay-insensitive circuit transitions occur only when and where they contribute to the computation (as opposed to the always-ticking clock). This makes delay-insensitive CMOS circuits particularly suitable for low-power applications.

Because there is no clock, the spectrum of electromagnetic radiation generated by delay-insensitive circuits is more evenly spread out than for synchronous circuits.

Our approach to the design of delay-insensitive circuits separates functional correctness concerns from efficiency concerns. Thus, it is easy to replace subsystems by faster implementations if that would be beneficial to the speed of the overall computation. Optimization for speed can be accomplished by "local fiddling", without jeopardizing the correctness of the entire circuit.

Acceptance

For a wider acceptance of delay-insensitive circuits it is necessary that—besides attention for education and theory construction—a whole range of support tools be developed. The features of the Extended DI Model can be incorporated into appropriate tools without much difficulty.

The *VLSI-programming* approach taken by Philips (see [vBNRS88, vBKR+91]) addresses the issue of tools in a new way. A system designer writes a TANGRAM program in terms of communicating processes. The TANGRAM compiler translates this program into a network of so-called *handshake components*. The delay-insensitivity of the resulting networks is guaranteed by the use of *four-phase handshake protocols* and *double-rail data encoding* (see [vB93]). Separate tools allow the system designer to evaluate and tune a design, for instance, on the basis of speed, area, and power estimates.

Much more could be said about the acceptance of delay-insensitive circuits, such as the importance of interfacing to existing technologies, in particular, clocked circuits. We will leave it at this.

The issues discussed in this section show that our initial motivations for looking into delay-insensitive design techniques were not always to the point. On the one hand, some expected benefits are in fact negligible or even negative. On the other hand, some positive results came in areas that were not foreseen.

It should be born in mind that our theory is applicable not only to the design of digital electronic circuits. The theory is very general and a source of interesting and beautiful mathematics (and art, for that matter). It may find renewed applications when future technologies, possibly involving optical or quantum phenomena, are developed for information processors.

Appendix A

Ordered Sets and Lattices

In this appendix we briefly summarize some basic definitions and theorems from the theory of ordered sets and lattices. For more details the reader is referred to [Bir84, DP90].

A.1 Relations

Let R be a binary relation on set V, that is, $R \subseteq V \times V$. It is customary to write $u\ R\ v$ for $(u, v) \in R$. Here is a table of common terminology for relations:

R is called	whenever
reflexive	$(\forall\, u :: u\ R\ u)$
anti-reflexive	$(\forall\, u :: \neg(u\ R\ u))$
symmetric	$(\forall\, u, v :: u\ R\ v \equiv v\ R\ u)$
antisymmetric	$(\forall\, u, v : u\ R\ v \wedge v\ R\ u : u = v)$
transitive	$(\forall\, u, v, w : u\ R\ v \wedge v\ R\ w : u\ R\ w)$

where all quantified variables range over V. For relation R on V we have

$$\text{'}R \text{ is symmetric'} \quad \equiv \quad (\forall\, u, v : u \neq v \wedge u\ R\ v : v\ R\ u)\ ,$$
$$\text{'}R \text{ is antisymmetric'} \quad \equiv \quad (\forall\, u, v : u \neq v \wedge u\ R\ v : \neg(v\ R\ u))\ .$$

This clarifies that the relationship between 'symmetric' and 'antisymmetric' is the same as that between 'reflexive' and 'anti-reflexive', namely that of strong negation.

A.2 Ordered Sets

A relation is called a **pre-order** when it is *reflexive* and *transitive*. A relation is called an **equivalence relation** when it is a *symmetric* pre-order. For pre-order $\sqsubseteq$ on set V, the relation $\sim$ defined by

$$u \sim v \quad \equiv \quad u \sqsubseteq v \wedge v \sqsubseteq u \tag{A.1}$$

for all u and v in V, is an equivalence relation on V.

A relation is called a **partial order**, or briefly an **order**, when it is an *antisymmetric* pre-order. For partial order $\sqsubseteq$ on V we call $\langle V; \sqsubseteq \rangle$ a (partially) ordered set or **poset**.

The **converse** of $\sqsubseteq$, denoted by $\sqsupseteq$, is defined by

$$u \sqsupseteq v \quad \equiv \quad v \sqsubseteq u \tag{A.2}$$

for all u and v in V. $\langle V; \sqsubseteq \rangle$ is a poset if and only if $\langle V; \sqsupseteq \rangle$ is a poset.

Partial order $\sqsubseteq$ is called **total** when

$$u \sqsubseteq v \quad \vee \quad v \sqsubseteq u \tag{A.3}$$

for all u and v in V.

We define relation $\sqsubset$ on V by

$$u \sqsubset v \quad \equiv \quad u \sqsubseteq v \wedge u \neq v \tag{A.4}$$

for all u and v in V. Note that relation $\sqsubset$ is anti-reflexive and transitive (and, hence, also antisymmetric). An anti-reflexive and transitive relation is called a **strict order**. Thus, every partial order corresponds to a strict order. The reverse also holds. If some relation $\sqsubset$ on V is a strict order, then relation $\sqsubseteq$, defined by

$$u \sqsubseteq v \quad \equiv \quad u \sqsubset v \vee u = v \tag{A.5}$$

for all u and v in V, is a partial order on V.

Let $\langle V; \sqsubseteq \rangle$ be a poset and U a subset of V. $\langle U; \sqsubseteq' \rangle$, where $\sqsubseteq'$ is the restriction of $\sqsubseteq$ to U, is also a poset. *Note*: It is customary to denote the restricted order $\sqsubseteq'$ again by $\sqsubseteq$.

Let v and w be members of V. We call v a **lower bound** of U when

$$(\forall\, u : u \in U : v \sqsubseteq u) \,. \tag{A.6}$$

We abbreviate this to $v \sqsubseteq U$ when confusion is unlikely (keep in mind that U could also be a member of V, besides being a subset). We have $v \sqsubseteq \varnothing$ for all v. Dually, v is called an **upper bound** of U when

$$(\forall\, u : u \in U : u \sqsubseteq v) \,, \tag{A.7}$$

abbreviated to $U \sqsubseteq v$. We call v **least** in U or **minimum** of U when $v \in U$ and $v \sqsubseteq U$. Dually, v is called **greatest** in U or **maximum** of U when $v \in U$ and $U \sqsubseteq v$. If a minimum (maximum resp.) of U exists then it is unique. The minimum (maximum resp.) of U—if it exists—is denoted by $\min U$ ($\max U$ resp.). We have $\min \{v\} = \max \{v\} = v$.

We call v **minimal** in U when $v \in U$ and

$$(\forall\, u : u \in U \wedge u \sqsubseteq v : u = v) \,. \tag{A.8}$$

Dually, we call v **maximal** in U when $v \in U$ and

$$(\forall\, u : u \in U \wedge u \sqsupseteq v : u = v) \,. \tag{A.9}$$

If v is least in U then v is minimal in U. The converse does not hold generally.

We call v **greatest lower bound**, or infimum, of U when v is greatest in the set of all lower bounds of U. If a greatest lower bound of U exists then it is unique and we denote it by $\sqcap U$. We also write $v \sqcap w$ for $\sqcap \{v, w\}$. Dually, v is called **least upper bound**, or supremum, of U when v is least in the set of all upper bounds of U. It is denoted by $\sqcup U$ when it exists, and we also write $v \sqcup w$ for $\sqcup \{v, w\}$. If $\min U$ ($\max U$ resp.) exists, then $\sqcap U$ ($\sqcup U$ resp.) also exists and they are equal. If $\sqcap U$ ($\sqcup$ resp.) exists and is in U, then $\min U$ ($\max U$ resp.) also exists and they are equal. For poset $\langle V; \sqsubseteq \rangle$, we have

$$\begin{aligned} \sqcap \varnothing &= \max V \ , \\ \sqcup \varnothing &= \min V \ , \\ \sqcap V &= \min V \ , \\ \sqcup V &= \max V \ , \end{aligned}$$

that is, the left-hand side exists if only if the right-hand side exists, and if both exist, they are equal. Element v is the greatest lower bound of U if and only if

$$(\forall\, w : w \in V : w \sqsubseteq v \ \equiv \ w \sqsubseteq U) \ . \tag{A.10}$$

Dually, v is the least upper bound of U if and only if

$$(\forall\, w : w \in V : v \sqsubseteq w \ \equiv \ U \sqsubseteq w) \ . \tag{A.11}$$

A.3 Lattices

We call poset $\langle V; \sqsubseteq \rangle$ a **lattice** when $v \sqcap w$ and $v \sqcup w$ exist for all $v, w \in V$. Let $\langle V; \sqsubseteq \rangle$ be a lattice. We can view $\sqcap$ and $\sqcup$ as (total) binary operators on V. Operators $\sqcap$ and $\sqcup$ are commutative, associative, and idempotent (as binary operators). Furthermore, we have

$$u \sqsubseteq v \sqcap w \ \equiv \ u \sqsubseteq v \wedge u \sqsubseteq w \ , \tag{A.12}$$

$$u \sqcup v \sqsubseteq w \ \equiv \ u \sqsubseteq w \wedge v \sqsubseteq w \ . \tag{A.13}$$

for all $u, v, w \in V$. Consequently, we also have

$$u \sqsubseteq v \ \equiv \ u \sqcap v = u \ , \tag{A.14}$$

$$u \sqsubseteq v \ \equiv \ u \sqcup v = v \ , \tag{A.15}$$

for all $u, v \in V$.

We call $\langle V; \sqsubseteq \rangle$ a **complete lattice** when $\sqcap U$ and $\sqcup U$ exist for every subset U of V. Poset $\langle V; \sqsubseteq \rangle$ is a complete lattice if and only if $\sqcap U$ exists for every subset U of V. For finite V, poset $\langle V; \sqsubseteq \rangle$ is a complete lattice if and only if $\sqcap \varnothing$ (i.e. $\max V$) exists and $v \sqcap w$ exists for all v and w in V. A finite lattice is a complete lattice.

This concludes our overview of the theory of ordered sets.

Appendix B

Some Proofs

In this appendix we have collected some proofs concerning the DI Model.

The following theorem characterizes interference in terms of *weave.S* rather than the system's reachable traces.

B.0.1 Theorem (*See Theorem 4.3.7*) Closed system S is free of interference if and only if

$$(\forall\, t, a, P : t \in \mathit{weave.S} \;\wedge\; P \in S \;\wedge\; a \in \mathbf{o}P \;\wedge\; ta\lceil\mathbf{a}P \in \mathbf{t}P : ta \in \mathit{weave.S})\ . \qquad \text{(B.1)}$$

Proof We prove the two implications separately.

Assuming the left-hand side 'S is closed and free of interference', we prove the right-hand side (B.1). For t, a, and P we derive

$$\begin{array}{ll} & t \in \mathit{weave.S} \;\wedge\; P \in S \;\wedge\; a \in \mathbf{o}P \;\wedge\; ta\lceil\mathbf{a}P \in \mathbf{t}P \\ \equiv & \quad \{\ \mathit{weave.S} = \mathit{reach.S} \text{ on account of the assumption and (4.14)}\ \} \\ & t \in \mathit{reach.S} \;\wedge\; P \in S \;\wedge\; a \in \mathbf{o}P \;\wedge\; ta\lceil\mathbf{a}P \in \mathbf{t}P \\ \Rightarrow & \quad \{\ \text{definition of } \mathit{reach.S}\ \} \\ & ta \in \mathit{reach.S} \\ \equiv & \quad \{\ \mathit{weave.S} = \mathit{reach.S} \text{ on account of the assumption and (4.14)}\ \} \\ & ta \in \mathit{weave.S} \end{array}$$

Hence, the right-hand side holds.

Now assume the right-hand side (B.1). We show that S is free of interference by proving

$$(\forall\, s : s \in \mathit{reach.S} : s \in \mathit{weave.S})$$

by structural induction on s.

Base: $s = \varepsilon$. Assume $\varepsilon \in \mathit{reach.S}$. We have $\varepsilon \in \mathit{weave.S}$ because $\varepsilon \in \mathbf{t}P$ for any P (since trace sets of processes are non-empty and prefix-closed).

Step: $s = ta$ for some trace t and symbol a. The induction hypothesis is $t \in \mathit{reach.S} \Rightarrow t \in \mathit{weave.S}$. For s we derive

$$
\begin{array}{ll}
& s \in \mathit{reach}.S \\
\equiv & \quad \{\ s = ta\ \} \\
& ta \in \mathit{reach}.S \\
\equiv & \quad \{\ \text{definition of } \mathit{reach}.S\ \} \\
& (\exists\, P : P \in S : t \in \mathit{reach}.S \ \wedge\ a \in \mathbf{o}P \ \wedge\ ta\lceil\mathbf{a}P \in \mathbf{t}P) \\
\Rightarrow & \quad \{\ \text{induction hypothesis}\ \} \\
& (\exists\, P : P \in S : t \in \mathit{weave}.S \ \wedge\ a \in \mathbf{o}P \ \wedge\ ta\lceil\mathbf{a}P \in \mathbf{t}P) \\
\Rightarrow & \quad \{\ \text{right-hand side assumed}\ \} \\
& ta \in \mathit{weave}.S \\
\equiv & \quad \{\ ta = s\ \} \\
& s \in \mathit{weave}.S
\end{array}
$$

This concludes the induction, thereby completing the proof. ∎

The next theorem expresses that *equ* is a congruence with respect to *par*.

B.0.2 Theorem (*See Section 4.4*) For systems S, S', T, and T' with S *equ* S' and T *equ* T' we have

$$S \mathit{\ par\ } T \quad \mathit{equ} \quad S' \mathit{\ par\ } T' \ . \tag{B.2}$$

Proof First of all, observe that on account of the properties of *par* mentioned below Example 4.2.2, we have

$$\mathit{Correct}.((S \mathit{\ par\ } T) \mathit{\ par\ } U) \ \equiv\ \mathit{Correct}.(S \mathit{\ par\ } (T \mathit{\ par\ } U)) \tag{B.3}$$

for systems S, T, and U. Now let S, S', T, and T' be systems with S *equ* S' and T *equ* T'. We derive for system U:

$$
\begin{array}{ll}
& \mathit{Correct}.((S \mathit{\ par\ } T) \mathit{\ par\ } U) \\
\equiv & \quad \{\ \text{observation (B.3)}\ \} \\
& \mathit{Correct}.(S \mathit{\ par\ } (T \mathit{\ par\ } U)) \\
\equiv & \quad \{\ S \mathit{\ equ\ } S' \text{ assumed, property (4.18) of } \mathit{equ}\ \} \\
& \mathit{Correct}.(S' \mathit{\ par\ } (T \mathit{\ par\ } U)) \\
\equiv & \quad \{\ \text{observation (B.3) and commutativity of } \mathit{par}\ \} \\
& \mathit{Correct}.(T \mathit{\ par\ } (S' \mathit{\ par\ } U)) \\
\equiv & \quad \{\ T \mathit{\ equ\ } T' \text{ assumed, property (4.18) of } \mathit{equ}\ \} \\
& \mathit{Correct}.(T' \mathit{\ par\ } (S' \mathit{\ par\ } U)) \\
\equiv & \quad \{\ \text{observation (B.3) and commutativity of } \mathit{par}\ \} \\
& \mathit{Correct}.((S' \mathit{\ par\ } T') \mathit{\ par\ } U))
\end{array}
$$

Therefore, we have S *par* T *equ* S' *par* T' on account of (4.18). ∎

The following lemma is used when proving that $\langle \mathcal{PROC}; \sqsubseteq \rangle$ is a partially ordered set.

B.0.3 Lemma For processes P, Q, and R (not $\bot$ or $\top$) we have

$$P \sqsubseteq Q \sqsubseteq R \;\Rightarrow\; \mathbf{t}P \cap \mathbf{t}R \subseteq \mathbf{t}Q \;. \tag{B.4}$$

Proof Assuming $P \sqsubseteq Q \sqsubseteq R$, we prove

$$(\forall\, s : s \in \mathbf{t}P \wedge s \in \mathbf{t}R : s \in \mathbf{t}Q) \tag{B.5}$$

by structural induction on s. Note that the assumption implies

$$\mathbf{i}P = \mathbf{i}Q = \mathbf{i}R \quad \wedge \quad \mathbf{o}P = \mathbf{o}Q = \mathbf{o}R \;. \tag{B.6}$$

Base: $s = \varepsilon$. On account of $Q \in \mathcal{PROC}$ we have $s = \varepsilon \in \mathbf{t}Q$.

Step: $s = ta$ for some trace t and symbol a. The induction hypothesis is

$$t \in \mathbf{t}P \wedge t \in \mathbf{t}R \;\Rightarrow\; t \in \mathbf{t}Q \;. \tag{B.7}$$

We now derive

$$\begin{array}{ll}
 & s \in \mathbf{t}P \wedge s \in \mathbf{t}R \\
\equiv & \quad \{\ s = ta\ \} \\
 & ta \in \mathbf{t}P \wedge ta \in \mathbf{t}R \\
\equiv & \quad \{\ P, R \in \mathcal{PROC}, \text{ hence } \mathbf{t}P \text{ and } \mathbf{t}R \text{ are prefix-closed}\ \} \\
 & t \in \mathbf{t}P \wedge t \in \mathbf{t}R \wedge ta \in \mathbf{t}P \wedge ta \in \mathbf{t}R \\
\Rightarrow & \quad \{\ \text{induction hypothesis (B.7)}\ \} \\
 & t \in \mathbf{t}Q \wedge ta \in \mathbf{t}P \wedge ta \in \mathbf{t}R \\
\Rightarrow & \quad \{\ \text{case analysis on } a \in \mathbf{i}Q \vee a \in \mathbf{o}Q, \text{ using respectively } P \sqsubseteq Q \text{ or } Q \sqsubseteq R\ \} \\
 & ta \in \mathbf{t}Q \\
\equiv & \quad \{\ ta = s\ \} \\
 & s \in \mathbf{t}Q
\end{array}$$

This concludes the proof. ∎

B.0.4 Theorem (*See Section 4.5*) $\langle \mathcal{PROC}; \sqsubseteq \rangle$ is a poset.

Proof We show that relation $\sqsubseteq$ on $\mathcal{PROC}$ is a partial order.

1. $P \sqsubseteq P$ follows immediately from the definition of $\sqsubseteq$. Hence, $\sqsubseteq$ is reflexive.
2. Assuming $P \sqsubseteq Q \wedge Q \sqsubseteq P$, we show $P = Q$. The assumption implies $\mathbf{i}P = \mathbf{i}Q$ and $\mathbf{o}P = \mathbf{o}Q$. Thus, it remains to prove $\mathbf{t}P = \mathbf{t}Q$. We derive

$$P \sqsubseteq Q \wedge Q \sqsubseteq P$$

$$
\begin{array}{ll}
\equiv & \{ \text{ calculus } \} \\
& P \sqsubseteq Q \sqsubseteq P \ \wedge \ Q \sqsubseteq P \sqsubseteq Q \\
\Rightarrow & \{ \text{ Lemma B.0.3 (twice) } \} \\
& \mathbf{t}P \cap \mathbf{t}P \subseteq \mathbf{t}Q \ \wedge \ \mathbf{t}Q \cap \mathbf{t}Q \subseteq \mathbf{t}P \\
\equiv & \{ \text{ set theory } \} \\
& \mathbf{t}P = \mathbf{t}Q
\end{array}
$$

This proves the antisymmetry of $\sqsubseteq$.

3. Assuming $P \sqsubseteq Q \ \wedge \ Q \sqsubseteq R$, we show $P \sqsubseteq R$. The assumption implies

$$\mathbf{i}P = \mathbf{i}Q = \mathbf{i}R \quad \wedge \quad \mathbf{o}P = \mathbf{o}Q = \mathbf{o}R \ . \tag{B.8}$$

So it remains to prove the third and fourth conjunct in definition (4.20) with $Q := R$. We do only the third conjunct, since the fourth follows from symmetry.

From Lemma B.0.3 and the assumption we infer $\mathbf{t}P \cap \mathbf{t}R \subseteq \mathbf{t}Q$. Now we derive for trace t and symbol a

$$
\begin{array}{ll}
& a \in \mathbf{o}P \ \wedge \ t \in \mathbf{t}P \ \wedge \ ta \in \mathbf{t}R \\
\equiv & \{ \ R \in \mathcal{PROC}, \text{ hence } \mathbf{t}R \text{ is prefix-closed } \} \\
& a \in \mathbf{o}P \ \wedge \ t \in \mathbf{t}P \ \wedge \ t \in \mathbf{t}R \ \wedge \ ta \in \mathbf{t}R \\
\Rightarrow & \{ \ \mathbf{t}P \cap \mathbf{t}R \subseteq \mathbf{t}Q \ \} \\
& a \in \mathbf{o}P \ \wedge \ t \in \mathbf{t}Q \ \wedge \ ta \in \mathbf{t}R \\
\Rightarrow & \{ \ \mathbf{o}P = \mathbf{o}Q \text{ and } Q \sqsubseteq R \ \} \\
& a \in \mathbf{o}P \ \wedge \ ta \in \mathbf{t}Q \\
\Rightarrow & \{ \ P \sqsubseteq Q \ \} \\
& ta \in \mathbf{t}P
\end{array}
$$

This completes the proof of $\sqsubseteq$'s transitivity.

∎

The following lemma is fundamental to the understanding of the Factorization Theorem.

B.0.5 Lemma For DI processes P and Q we have

$$P \sqsupseteq Q \ \equiv \ \mathit{Correct}.\{P, \rightsquigarrow Q\} \ .$$

Proof Follows from Theorems 4.5.2 and 4.7.7. ∎

B.0.6 Theorem (*See Theorem 4.9.1*) For DI processes P, Q, and R we have

$$P \parallel Q \sqsupseteq R \ \equiv \ P \sqsupseteq \rightsquigarrow(Q \parallel \rightsquigarrow R) \ .$$

Proof We derive

$$
\begin{array}{ll}
 & P \parallel Q \sqsupseteq R \\
\equiv & \quad \{ \text{ Lemma B.0.5 } \} \\
 & \mathit{Correct}.\{P \parallel Q, \backsim R\} \\
\equiv & \quad \{ \ P \parallel Q \ \mathit{equ} \ \{P, Q\} \ \} \\
 & \mathit{Correct}.\{P, Q, \backsim R\} \\
\equiv & \quad \{ \ \{Q, \backsim R\} \ \mathit{equ} \ Q \parallel \backsim R \ \} \\
 & \mathit{Correct}.\{P, Q \parallel \backsim R\} \\
\equiv & \quad \{ \ \backsim \text{ is its own inverse } \} \\
 & \mathit{Correct}.\{P, \backsim\backsim(Q \parallel \backsim R)\} \\
\equiv & \quad \{ \text{ Lemma B.0.5 } \} \\
 & P \parallel \sqsupseteq \backsim(Q \parallel \backsim R)
\end{array}
$$

References

[Bir84] Garrett Birkhoff. *Lattice Theory*, volume 25 of *Colloquium Publications*. American Mathematical Society, Providence, RI, third edition, 1984.

[CC79] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[Cla67] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336, Atlantic City, NJ, 1967. Academic Press.

[CM73] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[CM84] K. Mani Chandy and Jayadev Misra. Reasoning about networks of communicating processes. Unpublished paper presented at INRIA Advanced Nato Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, La Colle-sur-Loupe, France, 1984.

[CUV89a] Wei Chen, Jan Tijmen Udding, and Tom Verhoeff. Networks of communicating processes and their (de)-composition. Computing Science Notes 89/05, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1989.

[CUV89b] Wei Chen, Jan Tijmen Udding, and Tom Verhoeff. Networks of communicating processes and their (de)-composition. In Jan L. A. van de Snepscheut, editor, *The Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 174–196. Springer-Verlag, 1989.

[DEC93] DEC. Digital's Alpha chip project. *Communications of the ACM*, 36:30–83, February 1993.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[dNH83] R. de Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83-133, 1983.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Ebe88] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. Computing Science Notes 88/10, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1988.

[Ebe89] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[Ebe90] Jo C. Ebergen. Arbiters: An exercise in specifying and decomposing asynchronously communicating components. Research Report CS-90-29, Computer Science Dept., Univ. of Waterloo, Canada, July 1990.

[End77] Herbert E. Enderton. *Elements of Set Theory*. Academic Press, New York, 1977.

[Fan86] Ting-Pien Fang. On decomposition of delay-insensitive modules by factoring. Technical Memorandum 314, Computer Systems Laboratory, Washington Univ., St. Louis, MO, July 1986.

[GD85] L. A. Glasser and D. W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. McGraw-Hill, 1985.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes*. Series in Foundations of Computing. The MIT Press, Cambridge, Mass., 1988.

[HJ86] C. A. R. Hoare and He Jifeng. The weakest prespecification: Part I. *Fundamenta Informaticae*, 9:51–84, 1986.

[HJ87] C. A. R. Hoare and He Jifeng. The weakest prespecification. *Information Processing Letters*, 24(2):127–132, January 1987.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge Univ. Press, 1986.

[Hur75] Marco Hurtado. *Structure and Performance of Asymptotically Bistable Dynamical Systems*. PhD thesis, Sever Institute of Technology, Washington Univ., St. Louis, MO, 1975.

[JHJ89] Mark B. Josephs, C. A. R. Hoare, and He Jifeng. A theory of asynchronous processes. Technical Report PRG-TR-6-89, Oxford Univ., Computing Laboratory, 1989.

[JNH93] C. R. Jesshope, I. M. Nedelchev, and C. G. Huang. Compilation of process algebra expressions into delay-insensitive circuits. *IEE Proceedings-E*, 140(5):261–268, 1993.

[Jos92] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992.

[JU90] Mark B. Josephs and Jan-Tijmen Udding. The design of a delay-insensitive stack. Technical Report CS 9004, Dept. of C.S., Univ. of Groningen, The Netherlands, 1990.

[JU93] Mark B. Josephs and Jan Tijmen Udding. An overview of DI algebra. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, January 1993.

[Kal86] Anne Kaldewaij. *A Formalism for Concurrent Processes*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1986.

[KC87a] Lindsay Kleeman and Antonio Cantoni. Metastable behavior in digital systems. *IEEE Design & Test of Computers*, 4:4–19, December 1987.

[KC87b] Lindsay Kleeman and Antonio Cantoni. On the unavoidability of metastable behavior in digital systems. *IEEE Transactions on Computers*, C-36(1):109–112, January 1987.

[Kel74] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974.

[LS84] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification*. Wiley-Teubner Series in Computer Science. Wiley, 1984.

[Luc94] Paul G. Lucassen. *A Denotational Model and Composition Theorems for a Calculus of Delay-Insensitive Specifications*. PhD thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, May 1994.

[Mar81] Leonard R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.

[MB59] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[MC80] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[MFR85] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[ML86] Saunders Mac Lane. *Mathematics: Form and Function*. Springer, 1986.

[OH86] E.-R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23:9 66, 1986.

[Pee] Ad Peeters. The 'asynchronous' bibliography. Universal Resource Locator `ftp://ftp.win.tue.nl/pub/tex/async.bib.Z`. Corresponding e-mail address: `async-bib@win.tue.nl`.

[Pee90] Ad Peeters. Decomposition of delay-insensitive circuits. Computing Science Notes 90/04, Dept. of Math. and C.S., Eindhoven Univ. of Technology, April 1990.

[Pop83] Karl R. Popper. *Realism and the Aim of Science*. Hutchinson, London, 1983.

[Pra91] I. S. W. B. Prasetya. Solving the design equation in the failures model. Master's thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, July 1991.

[Rei85] Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[RMCF88] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005 1018, September 1988.

[Ros88] A. W. Roscoe. Two papers on CSP. Technical Monograph PRG-67, Oxford Univ. Computing Laboratory, Programming Research Group, 8 11 Keble Road, Oxford OX1 3QD, U.K., July 1988.

[RS93] Marly Roncken and Ronald Saeijs. Linear test times for delay-insensitive circuits: a compilation strategy. In S. Furber and M. Edwards, editors, *Proceedings of IFIP Working Conference on Asynchronous Design Methodologies*, pages 13–27, Manchester, UK, 31 March – 2 April 1993, 1993. Elsevier Science Publishers.

[Sch92] Huub Schols. *Delay-insensitive Communication*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, December 1992.

[Seg91] C-J. Seger. On the existence of speed-independent circuits. *Theoretical Computer Science*, 86(2):343–364, 1991.

[Sei79] Charles L. Seitz. Self-timed VLSI systems. In Charles L. Seitz, editor, *Proceedings of the 1st Caltech Conference on Very Large Scale Integration*, pages 345–355, Pasadena, CA, January 1979. Caltech C.S. Dept.

[Sei80] Charles L. Seitz. System timing. In Mead and Conway [MC80], chapter 7.

[Udd84] Jan Tijmen Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1984.

[UV88] Jan Tijmen Udding and Tom Verhoeff. The mathematics of directed specifications. Technical Report WUCS-88-20, Dept. of C.S., Washington Univ., St. Louis, MO, June 1988.

[vB92] Kees van Berkel. Beware the isochronic fork. *INTEGRATION, the VLSI journal*, 13(2):103–128, June 1992.

[vB93] Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. International Series on Parallel Computing. Cambridge University Press, 1993.

[vBKR+91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[vBNRS88] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proceedings ICCD'88 (IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 150 166, Rye Brook, New York, 1988.

[vdS85] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[vdSU86] Jan L. A. van de Snepscheut and Jan Tijmen Udding. An alternative implementation of communication primitives. *Information Processing Letters*, 23(5):231–238, 1986.

[Ver85] Tom Verhoeff. Notes on delay-insensitivity. Master's thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1985.

[Ver88] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[Ver89] Tom Verhoeff. Characterizations of delay-insensitive communication protocols. Computing Science Notes 89/06, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1989.

[Ver94] Tom Verhoeff. The testing paradigm applied to network structure. Computing Science Notes 94/10, Dept. of Math. and C.S., Eindhoven Univ. of Technology, March 1994.

[Waa89] Martin Waardenburg. Composition and classification of components. Master's thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1989.

[WE93] N. Weste and K. Eshragian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, second edition, 1993.

Index

Statements

accompanying the dissertation

A Theory of Delay-Insensitive Systems

by

Tom Verhoeff

Eindhoven University of Technology

20 May 1994

1. The testing paradigm is not only suitable for comparing behavioral aspects of systems, but also for comparing structural aspects.

   See this dissertation and [1].

   [1] Tom Verhoeff, *The Testing Paradigm Applied to System Structure*, Computing Science Notes 94/10, EUT, March 1994.

2. In contrast to what is claimed in [2], nondeterminacy cannot always be reduced.

   See Chapter 8 of this dissertation.

   [2] Edsger W. Dijkstra, *A Detailed Derivation of a Very Simple Program*, EWD1162, October 1993.

3. Let $p(a,b,c,d)$ be the probability that in a bridge game the players North, East, South, and West have a, b, c, and d spades, respectively. Let $q(a,b,c,d)$ be the probability that a hand at bridge, say of North, will consist of a spades, b hearts, c diamonds, and d clubs. Since $p(a,b,c,d) = q(a,b,c,d)$, the answers to exercises 32 and 34 in Section II.10 of [3] are inexcusably misleading.

   [3] William Feller, *An Introduction to Probability Theory and Its Applications*, Third Edition, Volume 1, John Wiley & Sons, 1968.

4. Rubik's Magic, the puzzle with the eight ingeniously hinging tiles, has 1351 spatial configuration classes, of which only two are planar.

   [4] Tom Verhoeff, *Magic and Is Nho Magic*, Cubism for Fun, Nr. 15, 1987. [The title, including h, is inspired by Simon Stevin's motto *Wonder en is gheen wonder.*]

5. Even for linked lists, Quicksort is a faster sorting algorithm than Merge sort.

   [5] Tom Verhoeff, *Quicksort for Linked Lists*, Computing Science Notes 93/03, EUT, January 1993.

6. The Prisoner's Dilemma—a discrete non-zero-sum two-player game, formulated for instance in [6]—has an interesting continuous version, which implies that, for a strategy to do well, the severity of retaliation should be strictly less than the severity of provocation. It remains to be explained why people often behave in the opposite way.

   [6] Robert Axelrod, *The Evolution of Cooperation*, Basic Books, 1984.

   [7] Tom Verhoeff, *A Continuous Version of the Prisoner's Dilemma*, Computing Science Notes 93/02, EUT, January 1993.

7. In TEX (see [8]), vertical alignment of boxes is harder to adjust than horizontal alignment: the former requires glue juggling, the latter not. This can be attributed to an unnecessary break of symmetry in the design of TEX. The reference point of a box is restricted to lie either on the left-most side of the box or, for negative width, on the right-most side. Its vertical position, being determined by height and depth, is not so restricted.

   [8] Donald E. Knuth, *The TEXbook*, Addison-Wesley, 1984.

8. (a) Computing science students ought to pay ample attention to mathematics.

   (b) Mathematics students ought to pay ample attention to computing science.

9. The three options *accept*, *accept after rewriting*, and *reject*, usually available to referees of journal articles for expressing their judgment, should be extended with the option *reject after rewriting*.

10. Inclusion of a frivolous index entry in a dissertation helps eliminate errors.

11. Manual Therapy, in spite of its name, is practised mostly by brain.

```
(* Mathematica program for front illustration *)

c  = Sqrt[1/3] ; (* Cos[ half obtuse angle ] *)
s  = Sqrt[2/3] ; (* Sin[ half obtuse angle ] *)
r  = 1 / (20s) ; (* relative beam width *)

a1 = {  c,  0,  s } ;
a2 = {  c,  0, -s } ;
a3 = { -c,  s,  0 } ;
a4 = { -c, -s,  0 } ;

cpr[u_,v_] := (* cross product *)
  { u[[2]] v[[3]] - u[[3]] v[[2]]
  , u[[3]] v[[1]] - u[[1]] v[[3]]
  , u[[1]] v[[2]] - u[[2]] v[[1]] }

vee[u_,v_] := (* half a rhombus frame *)
  With[ { m = { u, v, cpr[u,v]/c }
        , p = 1 - r
        , q = 1 + 2r }
      , With[ { u1 = {r,r,r}.m , u2 = {r,r,-r}.m
              , u3 = {p,r,r}.m , u4 = {p,r,-r}.m
              , u5 = {p,p,r}.m , u6 = {p,p,-r}.m
              , u7 = {q,0,0}.m , u8 = {q,q, 0}.m }
            , { Polygon[{ u2, u1, u3, u4 }]
              , Polygon[{ u4, u3, u5, u6 }]
              , Polygon[{ u7, u4, u6, u8 }]
              , Polygon[{ u7, u8, u5, u3 }] }
      ]     ]

rmb[u_,v_] := (* rhombus frame *)
  { vee[u,v], vee[v,u] }

obj = { rmb[a1,a2], rmb[a1,a3], rmb[a1,a4]
      , rmb[a2,a3], rmb[a2,a4], rmb[a3,a4] }

grf = Graphics3D[ obj, Boxed -> False ]

Show[ grf, ViewPoint -> { -0.1, -2.4, 0.8 } ]
```