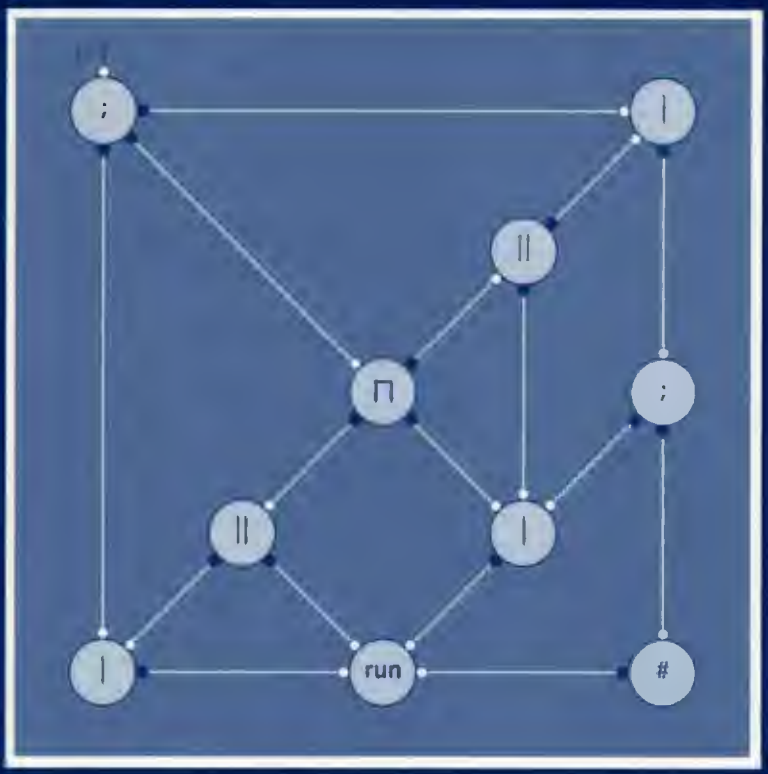
Handshake circuits : an intermediary between communicating processes and VLSI

Handshake circuits: an intermediary between communicating processes and VLSI

Kees van Berkel



Handshake circuits:
an intermediary between communicating processes and VLSI

Aan Takako, Tazuko, Koos en Leon

Cover: a fantasy handshake circuit shaped according to the ancient Chinese Tangram puzzle

Handshake circuits: an intermediary between communicating processes and VLSI

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 woensdag 6 mei 1992 om 16.00 uur

door

Cornelis Hermanus van Berkel

geboren te Leimuiden

Dit proefschrift is goedgekeurd door de promotor prof. dr. M. Rem.

The work described in this thesis has been carried out at the Philips Research Laboratories Eindhoven as a part of the Philips Research programme.

Acknowledgements

The work presented here grew out of the project "VLSI programming and silicon compilation" being conducted at Philips Research Laboratories Eindhoven since 1986. This project combines the research efforts of Ronan Burgess, Joep Kessels, Marly Roncken, Ronald Saeijs, Frits Schalij and myself. Together we defined the VLSI-programming language Tangram, built a silicon compiler, developed interesting demonstrators, and tested functional silicon. This thesis could only be written on the fertile ground of this inspiring and pleasant cooperation.

I am grateful to the management of Philips Research Laboratories, in particular to Theo Claasen and Eric van Utteren, for their support of the project, the provision of a very stimulating working environment, and their encouraging me to write this thesis.

Special thanks go to Cees Niessen. Numerous illuminating, critical, stimulating, and curious discussions with him helped me in choosing directions and setting priorities.

I am indebted to Martin Rem who supervised the work on this thesis. He also helped me in focusing this thesis on handshake circuits and separating essential issues from side issues. Also, his active interest in the topic provided a constant source of inspiration and motivation.

A number of people have given me substantial constructive criticism on all or parts of a draft version of this thesis. For their help I would like to thank Jos Baeten, Ronan Burgess, Ton Kalker, Joep Kessels, Frans Kruseman Aretz, Ad Peeters, Marly Roncken, Frits Schalij, and Kees Vissers.

Finally, this thesis could not have been built without TEX and LATEX, for which I thank Donald Knuth and Leslie Lamport.

Contents

Chapter 0

Introduction

This thesis is about the design of digital VLSI circuits. Whereas LSI circuits perform basic functions such as multiplication, control, storage and digital-to-analog conversion, VLSI circuits contain complex compositions of these basic functions. In many cases all data and signal processing in a professional or consumer system can be integrated on a few cm^2 of silicon. Examples of such "systems on silicon" can be found in

- Compact Disc (CD) players,
- Compact Disc Interactive (CDI) players,
- Digital Compact Cassette (DCC) players,
- Digital Audio Broadcast (DAB) receivers,
- cellular radios and mobile telephones,
- High-Definition TeleVision (HDTV) sets,
- digital video recorders,
- display processors,
- car-navigation systems,
- image processors, and
- digital test and measurement systems.

These systems generally process analog as well as digital signals, but the digital circuits dominate the surface of an IC. The memory needed for storing intermediate results often covers a significant fraction of the silicon area.

Systems on silicon tend to become more complex and tend to increase in number. The increase in complexity follows from advancements in VLSI technology, and the rapid growth of the number of transistors integrated on a single IC. The constant reduction of the costs of integration makes integration economically attractive for an increasing number of systems. Also, the rapid succession of generations of a single product increases the pressure on design time. The ability to integrate systems on silicon effectively, efficiently, and quickly has thus become a key factor in the global competition in both consumer and professional electronic products. This recognition has lead to a quest for design methods and tools that increase design productivity and reduce design times.

At Philips Research a number of approaches to this goal are being investigated [WD89,NvBRS88,LvMvdW*91]. One of these, viz. "VLSI programming and compilation to asynchronous circuits" forms the background of the research reported in this thesis. The central idea is that of viewing VLSI design as a programming activity, and thereby capitalizing on the achievements in computing science with regard to complexity bridling [Sei80,Rem81,vdS85,Mar89].

VLSI programming assumes a VLSI-programming language that provides the programmer with a suitable abstraction from the VLSI technology and circuit techniques. This abstraction allows systems on silicon to be designed by system (e.g. digital audio) specialists without detailed involvement of IC specialists. Ideally, this avoids the costly, time-consuming and error prone transfer of design data from system specialists to VLSI-circuit specialists. The degree of abstraction is constrained by the required cost and performance of the resulting IC. A VLSI programming language is thus a compromise between programming convenience and silicon efficiency.

The automatic translation of VLSI programs into VLSI circuits is often called *silicon compilation*. This thesis proposes a compilation scheme that results in asynchronous circuits. This relatively uncommon circuit style has specific advantages with regard to system modularity and IC power consumption.

The central contribution of this thesis is that of *handshake circuits*: an intermediary between VLSI programs and VLSI circuits. A handshake circuit is a network of asynchronous components connected by point-to-point channels along which components interact by means of handshake signaling. The role of an intermediary is generally that of separation of – more or less orthogonal – concerns. This introductory chapter continues with taking stock of these concerns and ends with an overview of this thesis.

First we shall have a closer look at a particular system on silicon: a Compact Disc Decoder IC. This example shows the variety in interfaces, protocols and data types involved in system design.

The next section examines the VLSI medium by means of the mainstream VLSI technology CMOS. A computation will be viewed in terms of voltage transitions on wires. Differences between synchronous and asynchronous circuits are explained by discussing how to deal with the phenomenon called *interference*.

The final section contains a roadmap to this thesis and positions the handshake circuits as an intermediate form between VLSI programs and VLSI circuits.

0.0 VLSI systems

One of the key modules of the Compact Disc (CD) player is its chip set. Other key modules are: a laser-optical pick-up, a turn table, and a user interface consisting of a key-board and a display. Typically, the chip set consists of a servo controller, a decoder, a digital filter, a digital-to-analog converter, a DRAM, and a micro processor [Phi90]. There is a tendency towards single-chip solutions. The decoder has been selected to illustrate a number of issues relevant to VLSI programming.

The main function of the decoder is to convert the digital signal from the optical disc into a digital (stereo) audio signal. The block diagram of the decoder in Figure 0.0 has been adapted from [Phi90]. The main parts of the interface of the decoder are:

- *clock*: crystal oscillator input (11.2896 MHz),
- A: bit stream from the optical pick-up (*average* bit frequency: 4.32 MHz),
- B: disc-motor control signal, pulse-width modulated (88.2 kHz, duty factor ranges from 1.6 % - 98.4 %),
- C: interface to external DRAM of 16k $\times$ 4 bit (12 *clock* cycles for a single read or write access),
- D: bit serial output of stereo samples (2 $\times$ 16 bit) with an error flag per sample in parallel (rate: *clock*/4 $\approx$ 2.82 MHz),
- E: subcode signal to external microprocessor (bit-serial, in bursts of 10 bit at 2.82 MHz; one handshake per burst).

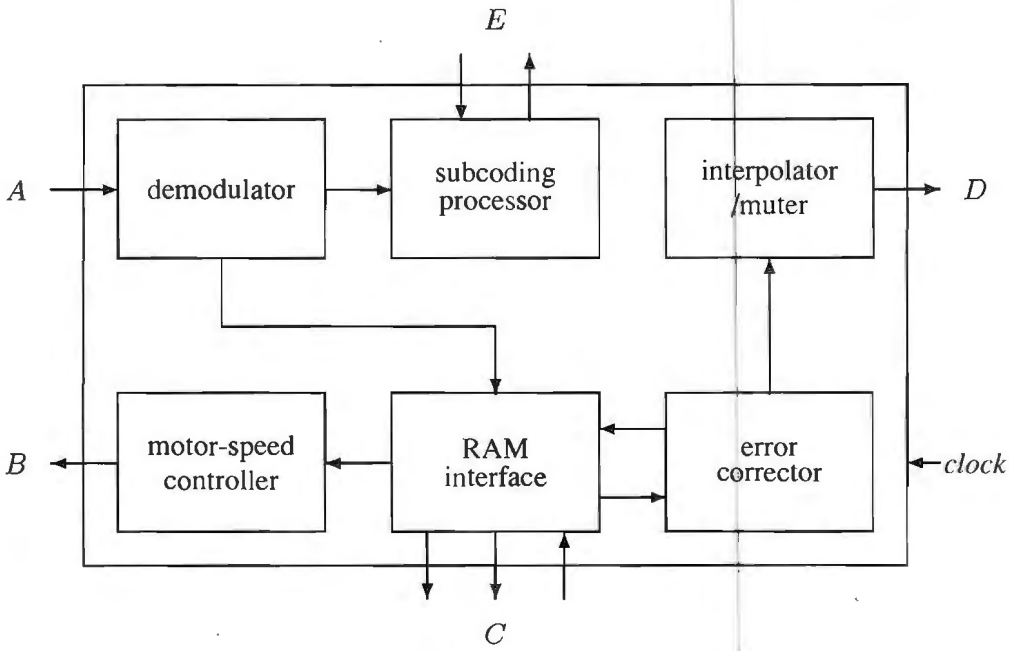


Figure 0.0: Simplified block diagram of the Compact Disc Decoder SAA7310.

The above overview indicates a variety in data rates and clock frequencies. Most frequencies are derived from *clock* by division. The incoming bit stream A, however, contains an implicit clock that must be recovered through frequency and phase detection. This clock frequency varies with the rotation speed of the disc. A feed back mechanism guarantees that *on the average* the incoming data rate matches the outgoing data rate.

There is also a considerable variety of data types involved. The incoming data are a formated stream of bits. Frames of 32 symbols of 8 bit are extracted internally. The arithmetic applied to these symbols, as required for error correction, is in the Galois Field $GF(2^8)$. The interfacing to the RAM requires address computations in natural numbers in the range $[0..2^{16})$. Audio samples are pairs of integers in the range $[-2^{15}..2^{15})$. A general-purpose VLSI-programming language must be able to deal with this variety in data rates and data types.

The behavior of VLSI programs is restricted to discrete events and discrete values. Conversion to and from time-continuous and/or continuous values requires adapter circuits such as digital-to-analog converters.

The main submodules of the decoder are (with reference to Figure 0.0):

- *Demodulator*: extracts a clean digital signal *and* a clock signal from the disc signal. This digital signal is then demodulated and converted into frames of 32 symbols of 8 bit, error flags and subcode information. The rate of the extracted clock signal follows the rotation speed of the disc. This clock is local to the Demodulator.

- *Subcoding processor*: accumulates subcode words of 96 bit, performs a cyclic redundancy check (CRC), and sends the corrected word (80 bit) to an external microprocessor on an external clock.

- *RAM interface*: controls the traffic "Demodulator → RAM → Error corrector → RAM → Error corrector". The external RAM is used for two distinct purposes: that of a first-in first-out queue (FIFO) to buffer the irregularly produced data from disc, and that of a store for de-interleaving the symbol stream.

- *Motor-speed controller*: controls the speed of the disc motor based on the degree of occupancy of the FIFO.

- *Error corrector*: corrects the code words according to Cross Interleaved Reed-Solomon Code (CIRC) with a maximum of 2×2 errors per frame of 32 symbols.

- *Interpolator/Muter*: converts symbols in stereo audio samples, interpolates single errors and mutes in the presence of two or more successive erroneous samples.

These submodules operate in parallel. It is therefore hard to describe the behavior of the decoder in a traditional imperative programming language (such as Pascal, C or Fortran). The behavior of each submodule, however, can be conveniently described in such a language extended with appropriate primitives for input and output.

This describes exactly the idea of Communicating Sequential Processes (CSP) as proposed by Hoare in [Hoa78], and forms the basis of the VLSI-programming language Tangram [0] developed at Philips Research.

[0] Tangram is the name of an ancient Chinese puzzle [Elf76]. It consists of a few, simple forms (five triangles of three different sizes, one square and one parallelogram), a simple composition rule (forms may not overlap), and allows the construction of a large variety of intricate and fascinating shapes. This view on design also shaped our VLSI-programming language Tangram.

0.1 VLSI circuits

A typical 1991 VLSI circuit is almost 1 cm^2 in size and consists of about 100,000 transistors, 100 meter wiring and 100 bonding pads. During its operation, when connected to a power supply, more than a 10^{11} events (voltage transitions) may occur each second, of which often less than one percent are observable at the bonding pads. An event consumes about a picojoule of energy; the power consumption of a chip is usually less than one Watt.

These rounded numbers apply to digital VLSI circuits manufactured in CMOS, the dominant VLSI technology of today. For state-of-the-art chips the above numbers may be multiplied by an order of magnitude. The yearly world production of integrated transistors is in the order of 10^{15}, or about a thousand transistors per world citizen per day[1].

Transitions

The voltage transitions observable at the bonding pads are the only evidence of a computation going on inside a VLSI chip, apart from indirect evidence such as power consumption. We shall therefore first concentrate on such events, in particular on their occurrences on wires.

Wires are metal conductors usually connecting two or more (distant) transistors. Electrically they can be regarded as capacitors to the IC substrate. Except for the very long wires, the metal area may be considered *equipotential*: differences in potential along the wire tend to equalize in a time period shorter than the switching time of a transistor [2]. For long wires and wires made of material with a high sheet resistance such as polysilicon this approximation is not valid. Then the transmission delays caused by wire resistance and the speed of light may no longer be neglected.

A wire may be charged ("pulled up") through a path of transistors connected to the power-supply rail. Such a pull-up path usually consists of pMOS transistors, a type of transistor that conducts when its gate potential is low, i.e. connected to an uncharged wire. Similarly, wires may be discharged ("pulled down") by a path of nMOS transistors connected to *ground*. An nMOS transistor conducts when its gate potential is high. Often such paths may be merged, i.e.

[1] A similar type of estimate was presented during an invited lecture delivered by G. Moore at the Decennial Caltech Conf. on VLSI, 1989.

[2] For equipotential wires the image of a voltage transition propagating along a wire is false; when applied with care, the metaphor (as e.g. applied in the foam-rubber wrapper principle [Udd84]) may be useful.

individual transistors or combinations of transistors may be part of more than one path.

Generally, the situation in which both a pull-up path and a pull-down path compete in charging and discharging a wire is avoided, or at least restricted to a very short duration. For longer durations this form of short-circuit dissipation may form a considerable power drain.

When a wire is neither pulled up nor pulled down (it "floats"), its potential may not be constant due to charge leakage. A circuit is *static* if it has the property that its wires never float. If the floating of wires is essential for the operation of a circuit, the circuit is called *dynamic*.

Interference

So far, it was tacitly assumed that voltage transitions are complete, i.e. they proceed all the way from the ground to the supply voltage or vice versa. But what if the (dis-)charging of a wire is interrupted? Figure 0.1 depicts two wires a and b and an nMOS transistor n.

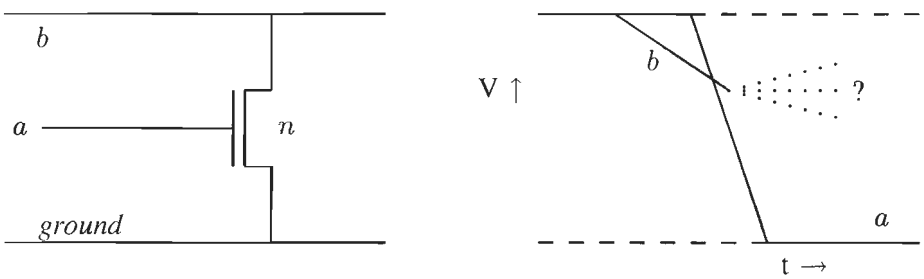


Figure 0.1: Interference occurs e.g. when wire a is discharged during the discharging of b.

When wire a has a high potential, the nMOS transistor forms a conducting path between wire b and ground. Assume that b is being discharged through n, and the potential on a drops to the ground level: the discharging of b is interrupted. Wire b is discharged partially and its potential is somewhere between the ground and the supply voltage. In such a situation, the transition on a is said to *interfere* with the transition on b. The transistors controlled by b may or may not have changed their state of conductance, and may or may not be involved in (dis-)charging other wires, et cetera. If b is subsequently recharged, the effect of this "runt" pulse on other wires critically depends on sizes of currents, capacitors,

threshold voltages of transistors and the time interval between the two events. The runt pulse may have caused a multitude of partial or complete transitions on other wires, or not. Similar complications occur when the discharging of b is interrupted by a short period of charging. Figure 0.2 gives examples of a proper transition (monotonic and complete), a runt pulse and a non-monotonic transition.

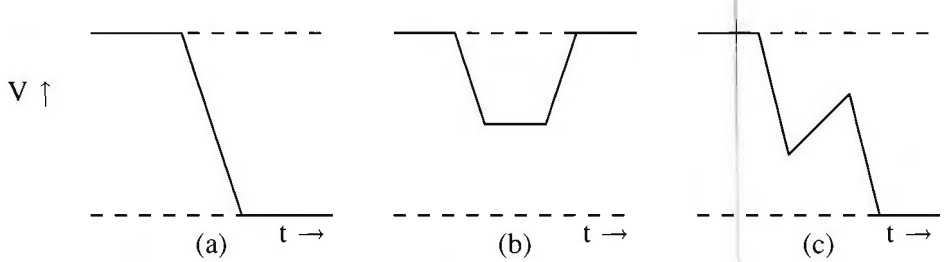


Figure 0.2: A proper transition (a), a runt pulse (b), and a non-monotonic transition (c).

There are two ways of dealing with interference:

- Accept the possibility of interference, but insist that at specific moments the mess has cleared, i.e. the circuit is in a quiescent state and all wires are stable at well-defined potentials. Synchronous timing disciplines are based on this principle: an externally supplied clock defines the moments at which the circuit *must* be quiescent.

- Avoid interference: guarantee that all transitions are monotonic and complete. Many *asynchronous* timing disciplines are based on this principle, *self-timed* and *delay-insensitive* being two of them.

The overwhelming majority of today's digital VLSI circuits are synchronous.

Proper transitions

Our interest in asynchronous circuits justifies some elaboration on the notion of *proper* transition. A nice and effective way to capture all requirements on proper transitions is by means of a phase diagram as proposed in [Bro89]. The evolution of the voltage of a wire in time is then recordered by a so-called trajectory in the space (V, dV/dt). The values of V and dV/dt are bounded by a doughnut shape as in Figure 0.3. With this choice of axis orientations, changes in V result in

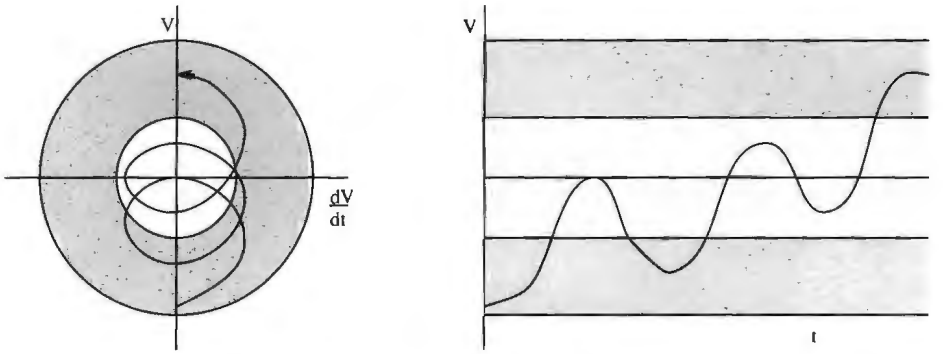


Figure 0.3: Phase and time diagram for a runt pulse followed by a non-monotonic transition.

counter-clockwise trajectories. Lower bounds on $(V,\ dV/dt)$ exclude runt pulses and non-monotonic transitions as illustrated in Figure 0.3.

The thickness of the doughnut determines amongst others the margins in the voltage to count as logical *false* or *true*. Within these margins runt pulses may occur, as illustrated in Figure 0.4. The doughnut also bounds the slope of a transition. This is significant, because different transistors may change their state of conductance at different voltage levels of the controlling gate. Transistors controlled by the same wire may then "observe" the same transition at different moments. Bounds on the slope of a transition therefore effectively limit these time differences (cf. isochronic forks in Section 7.5).

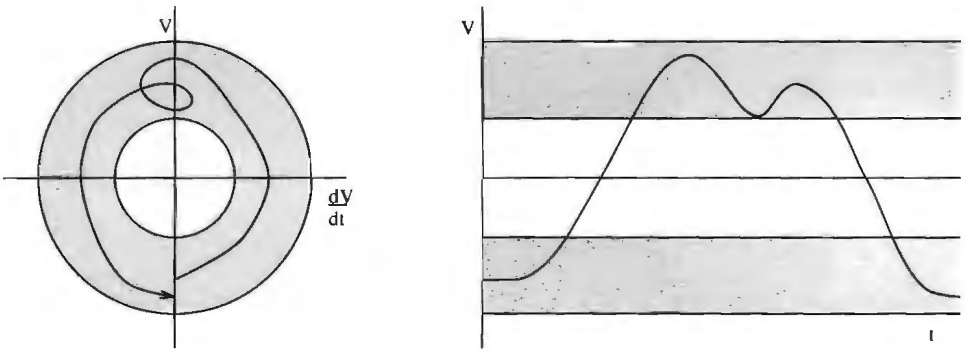


Figure 0.4: Phase and time diagram for two proper transitions in succession.

Asynchronous circuits

The method addressed in this thesis aims at the design of asynchronous VLSI circuits. The key question is then how to guarantee absence of interference. How to control the timing and dependencies of billions of transitions in such a way that the integrity of every single transition is assured?

The central idea is to construct the circuit from elementary circuits that indicate by means of transitions on output wires that transitions on specific input wires will not cause interference[3]. Circuits organized according to this principle are said to be *self-timed* [Sei80]

Delay-insensitive circuits are a restricted form of self-timed circuits. A circuit is delay-insensitive if the absence of interference does not depend on any assumption about delays of these elementary circuits and wires [vdS85]. In delay-insensitive circuits only point-to-point wires between elementary circuits are allowed, i.e. wires that connect one output to one input. A major advantage of delay-insensitive circuits is their *modularity*: a delay-insensitive composition of subcircuits will operate correctly, regardless of the response times of the subcircuits and regardless of the delays introduced by the connecting wires. An appraisal of delay-insensitive circuits is given in Section 8.1.

The constituent elementary circuits of self-timed and delay-insensitive circuits may be of arbitrary size, ranging from an inverter to an embedded self-timed RAM. It is attractive to have a finite set of elementary circuits from which delay-insensitive circuits can be constructed for any specification. Such a universal basis of elementary circuits has been proposed in [Ebe89]. Unfortunately, circuits constructed from this basis exclusively tend to be unpractically large.

A more practical set of elementary circuits has been proposed by Martin [Mar85b]. These elementary circuits are called *VLSI operators* and form a generalization of the traditional gates. Unfortunately, the class of delay-insensitive circuits that can be constructed from VLSI operators is severely restricted [BE90]. With the so-called *isochronic fork* as only concession to delay insensitivity [Mar90] reasonably efficient circuits can be constructed (cf. Section 7.5).

[3]The requirement of absence of interference restricts the behavior of the environment as well. In some cases this may be too restrictive. For instance, the handling of concurrent requests for a single resource requires a circuit that assures mutual exclusion. Inside such a mutual-exclusion circuit interference cannot be avoided. It is a good idea to localize the interfering transitions inside elementary circuits such as arbiters.

Cost and performance issues

A 1991 CMOS transistor is less than 10 μm^2 in area. This would allow for a packing density of over 100,000 transistors per mm^2. The densest practical circuits are embedded memories with about 10,000 transistors per mm^2. The average density in other VLSI circuits is almost an order of magnitude below this number. The almost two orders of magnitude difference between possible and practical transistor density is caused by wires. A quick glance at any VLSI circuit layout shows that wires dominate the circuit area and therefore production costs. The area of an IC is still a most critical resource: 20 % area overhead in a competitive market is considered a serious handicap, and 50 % area overhead is usually acceptable only for prototype circuits, or for small series.

The time it takes to (dis-)charge a wire is proportional to its capacitance, and, for a given width, this is proportional to its length. For average wires this is about 1 nanosecond. For longer wires this may exceed 10 nanoseconds. The switching time of a transistor is well below 1 nanosecond. Clearly, the wires determine the operating speed of a VLSI circuit.

For a given power-supply voltage, the energy consumed by a single event is proportional to the capacitance of the wire on which it occurs. Consequently, the energy required for a computation depends on the number of events and the lengths of the wires involved. For a given set of events the wires determine the energy consumption of a circuit. More and more often these systems on silicon end up in portable products such as walkmans and notebooks. Efficient usage of battery power is then an important design consideration. Asynchronous circuits potentially consume less energy, because there is no energy used for clock distribution and no energy is wasted in interference.

In summary: wires dominate concerns for cost and performance in every respect. The wires determine the area, the computation time and the energy consumption [SM77,Sei84]. Every VLSI design method, existing or novel, must acknowledge this fact.

Testing

The VLSI fabrication process is extremely complicated. For moderately sized circuits the yield is about 50 %, i.e. 50 % of the manufactured circuits function correctly. For complex circuits in an advanced technology the yield may well be below 10 %. To make things worse, for larger circuits the yield decreases exponentially with the circuit area. This has two important consequences: circuit area is a most critical resource and there is a *test problem.*

The problem of testing is how to discriminate between a correct circuit and a faulty circuit. This bears no relation with software testing. It is assumed that the circuit *design* is correct and that a possible malfunctioning is caused by a defect introduced during the fabrication of the circuit. For advanced production technologies such defects cannot be avoided: their density is about 1 per cm^2.

The problem of testing consists of two parts:

- bring the circuit into a state where an assumed fault makes a difference in the subsequent computation;
- detect this possible difference.

Given the exorbitant number of possible faults and circuit states on the one hand and the limited number of pads to control and observe the circuit behavior on the other hand, it is clear that the test problem is a hard one, in every respect. Testing of circuits is also costly: provisions for enhancing the testability of a circuit and executing tests may well account for 10 to 30 % of the price of an IC.

Given the complexity of testing, the user of an IC is not in a position to test an IC effectively. It is the joint responsibility of the circuit designer and the manufacturer. A novel VLSI-circuit design method without a systematic, effective and affordable test method simply is not viable.

0.2 Overview of this thesis

Handshake circuits are intended as an intermediary between VLSI programs and VLSI circuits. It thus separates concerns for systematic and efficient VLSI-programming from concerns at the VLSI-circuit level, such as absence of interference, data encoding, initialization, and testing (cf. Figure 0.5). The narrow waist of the "hourglass" is intended to reflect the clear separation realized by handshake circuits.

Handshake circuits can also be considered as a VLSI *architecture*. According to Webster's Ninth New Collegiate Dictionary [Mis87] one of the meanings of architecture is "a unifying or coherent form or structure". Handshake circuits unite control, storage, communication, logic and arithmetic in a single structure, supported by a single form of interaction: that of handshake signaling.

An overview of this thesis is presented with reference to Figure 0.6. By means of a variety of examples Chapter 1 presents an informal introduction to Tangram and handshake circuits. Concerns for cost and performance get special

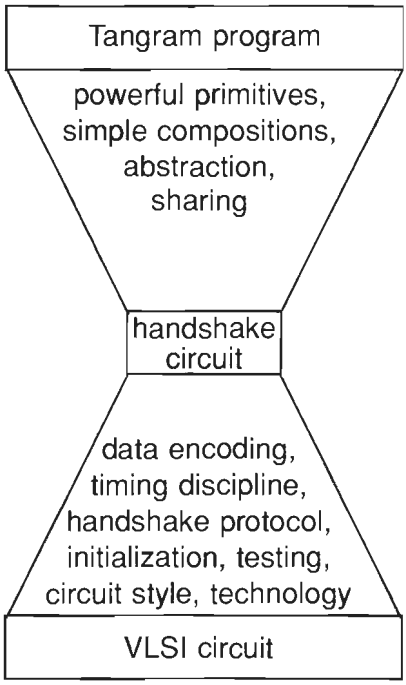


Figure 0.5: Handshake circuits: a separation of VLSI programming and VLSI circuits concerns.

attention, as they make VLSI programming different from (and also more difficult than) traditional computer programming.

The body of this thesis is a theory for handshake circuits. The key notion is that of *handshake process*. A handshake process is a mathematical object that describes a handshake-communication *behavior*. This handshake behavior may be that of the components of a handshake circuit (Chapter 2).

A handshake circuit is a set of handshake processes that satisfy a simple composition rule (Chapter 3). The behavior of the handshake circuit is defined through parallel composition '||' of its constituent components, and is, again, a handshake process. In Appendix A the delay insensitivity of handshake circuits is related to the theory reported in the literature.

Chapter 4 develops a calculus for handshake processes. This calculus allows concise descriptions of behaviors of handshake components.

In Chapter 5 a precise definition of Tangram is given. For a subset of Tan-

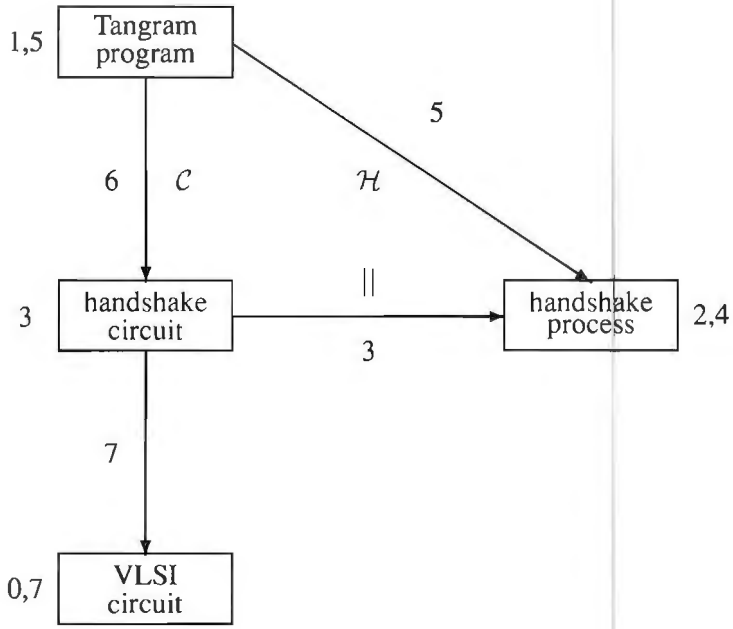


Figure 0.6: An overview of this thesis (numbers denote chapters).

gram, which we call *Core Tangram*, a formal denotation '$\mathcal{H}$' is given in terms of handshake processes. Here we use the calculus of Chapter 4. In Appendix B a link to the well-known failure semantics of CSP is established.

Chapter 6 describes the translation of Tangram programs into handshake circuits by means of the mathematical function '$\mathcal{C}$'. For Core Tangram it is proven that the behavior of the compiled handshake circuit is equivalent to that of the original program in a well-defined sense.

The realization of handshake circuits in VLSI is the subject of Chapter 7. Due to the variety of issues and the large number of choices involved we have not pursued completeness here. Issues such as peephole optimization, handshake refinement, data encoding, decompositions into VLSI-operator networks, initialization and testing are discussed in varying degrees of depth and completeness.

Chapter 8 discusses some practical experiences with VLSI programming and silicon compilation at Philips Research.

Chapter 1

Introduction to Tangram and handshake circuits

1.0 Introduction

This thesis pursues a programming approach to the design of digital VLSI circuits. In such an approach the VLSI-system designer constructs a program in a suitable high-level programming language. When he is satisfied with his program the designer invokes a so-called *silicon compiler* which translates this program into a VLSI-circuit layout.

The choice of the programming language is a crucial one, for it largely determines the application area, the convenience of design, and the efficiency of the compiled circuits. A good VLSI-programming language

0. is *general purpose* in that it allows the description of all digital functions;

1. encourages the *systematic* and *efficient* design of programs by abstracting from circuit, geometry and technology details;

2. allows the *automatic* translation into *efficient* VLSI circuits and test patterns.

Below follows a motivation for these requirements.

0. A wide range of applications is required to justify the investment in tools and training.

1. A major gain in design productivity can be expected by designing in a powerful high-level language. Furthermore, system designers do not need to

resort to VLSI specialists. Systematic design methods, supported by mathematical reasoning, are required to deal with the overwhelming complexity involved in the design of VLSI systems.

2. Automatic translation to VLSI circuits avoids the introduction of errors at the lower abstraction levels. It also becomes attractive to design alternative programs and compare the translated circuits in costs and performance.

Any such language is of necessity a compromise between convenience of design and efficiency of the result.

Traditional programming languages such as Pascal and C can be considered for this purpose. However, these languages were conceived for the sequential execution on a specific architecture. It is not at all clear how to benefit from the parallelism so abundantly available in VLSI, when the program describes a total order of all elementary computation steps. On the other hand, these so-called *imperative* programming languages are successful in that they are general purpose and offer a good compromise between convenience of design and efficiency of the compiled machine code.

In an effort to add parallelism to the traditional sequential programming languages, Hoare developed Communication Sequential Processes [Hoa78]. CSP soon became an important vehicle for the (theoretical) study of *concurrency* in computing science. It also was the basis for OCCAM [INM89], a language suitable for programming networks of microprocessors. The suitability of CSP-based languages for VLSI programming has been addressed in [Mar85a,vBRS88].

In terms of CSP a VLSI circuit can be described as a fixed network of processes connected by channels. These processes are simultaneously active and co-operate by synchronization and the exchange of messages along channels. The behavior of each process can be described in a C or Pascal-like language to which proper primitives for synchronization and communication have been added.

If one considers the translation of such programs into circuits, it is attractive to preserve the parallelism of the program by translating each process into a subcircuit, and by translating each channel into a set of wires. This "transparent" way of translating programs into circuits has the advantage that the programmer has control over the efficiency and performance of his circuits.

Figure 1.0 depicts an example of a network of three communicating processes P, Q and R. The arrows indicate the directions of data transport along the channels. Channel a is an input to P, b is an input channel that forks to two processes, and e is an output channel. Channels c, d and f do not convey data: they are used for synchronization only.

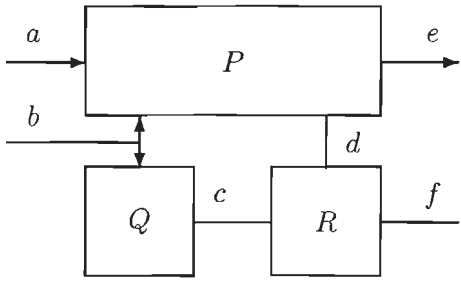


Figure 1.0: Communicating Sequential Processes.

One of the attractions of CSP is that it allows arbitrary numbers of processes of arbitrary complexity. The table below gives an impression of what can be realized in a single 100,000 transistor IC in terms of communicating processes. For a single IC the product of the degree of parallelism and the grain size (size of each process, measured e.g. in number of transistors) is more or less constant.

degree of parallelism	# processes	# transistors / process	example
sequential	1	100k	microprocessors
coarse-grained	10	10k	digital audio (CD)
fine-grained	100	1k	systolic arrays

So far the notion "process" has been used rather loosely. In the sequel it is used to denote the set of observable communication behaviors of an object, irrespective of how the object is organized internally. The behavior of a network of processes can also be described as a single process. A program is an alternative way to define a process.

This thesis uses Tangram as a VLSI programming language. Tangram has been developed at Philips Research. It is based on Hoare's CSP [Hoa85] and includes Dijkstra's guarded-command language [Dij75].

The translation of Tangram programs into VLSI circuits has so-called *handshake circuits* as an intermediary. Handshake circuits are networks of elementary asynchronous processes that communicate according to a handshake protocol. These elementary processes are called handshake components. The translation of Tangram programs into handshake circuits requires a modest set of different handshake components. The translation method is highly transparent, which allows the VLSI programmer to infer cost and performance of the compiled circuit

fairly directly from his Tangram program.

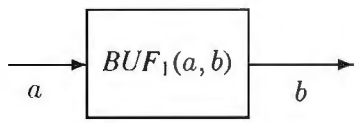
This chapter gives an informal introduction to Tangram by means of a series of small programs. For many of the programs or program fragments the corresponding handshake circuits are presented. The examples have been chosen so as to cover most of Tangram, and to give a flavor of VLSI programming and the translation of Tangram programs into handshake circuits.

1.1 Some simple Tangram programs

One of the simplest Tangram programs is $BUF_1(a, b)$, a one-place buffer:

$$(a?W \;\&\; b!W)\cdot|[x : \textbf{var}\ W \mid \#[a?x; b!x]]|$$

where W is an arbitrary type, e.g. **bool** or the integer range [0..256). The opening pair of parenthesis contains the declarations of the external ports of $BUF_1(a, b)$. Port a is an input port of type W and port b is an output port of the same type.



The segment of Tangram text following the dot defines the behavior of the program. This behavior is described by a so-called *command* (statement). Here the behavior is described by a *block* command, in which the local variable x of type W is introduced. The brackets '|[' and ']|' delineate the scope of x. The bar '|' separates the local declarations from the command.

Command $\#[a?x; b!x]$ defines an infinite repetition of input action $a?x$ followed by output action $b!x$. Execution of command $a?x$ amounts to the reception of an incoming value through a and the storage of that value in variable x. Command $b!x$ denotes the sending of the value of x through b.

In summary, $BUF_1(a, b)$ repeatedly receives a value through a and sends that value through b. The variable x is merely a container to store the incoming value between the two communication actions. The identity of x and its role in the operation of the buffer cannot be observed externally, since x is effectively concealed by the scope brackets. Only the external communications through a and b can be observed. If type W is the range [0..10), a possible observation of $BUF_1(a, b)$ is:

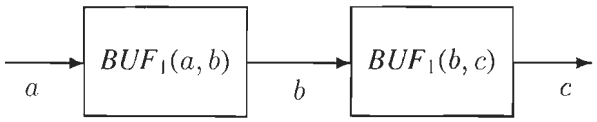
$$a{:}3\ \ b{:}3\ \ a{:}9\ \ b{:}9\ \ a{:}9$$

where $a:v$ denotes the communication of value v through port a. Such a finite sequence of communications is called a trace.

A slightly more interesting program is that of two-place buffer $BUF_2(a, c)$:

$$(a?W \ \& \ c!W)\cdot|[b : \textbf{chan}\ W \mid (BUF_1(a, b) \mid\mid BUF_1(b, c))]|$$

This two-place buffer is a cascade of two instances of BUF_1. The output of the first instance is connected to the input of the second. Both instances operate in parallel, as denoted by '||'. Cascades of instances of BUF_1 are called "ripple buffers".



The internal communication along channel b has two aspects. Firstly, it requires simultaneous participation of sender and receiver. In other words, the output action of the left BUF_1 and the input action of the right BUF_1 form a single communication action. Secondly, a communication has the effect of the assignment action $xr := xl$, where xl and xr are aliases for the variable x in the left and right buffer instance respectively.

Communications along b are concealed by the scope brackets around the declaration of channel b. The communication behavior of $BUF_2(a, c)$ is more interesting than that of $BUF_1(a, c)$. In addition to all the traces of the one-place buffer (with their output port renamed), a trace such as:

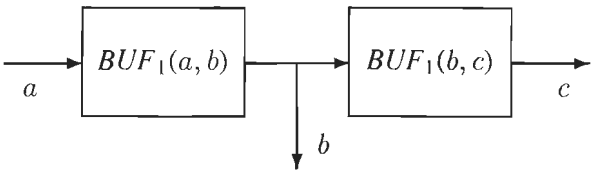
$$a{:}3 \ \ a{:}9 \ \ c{:}3 \ \ c{:}9 \ \ a{:}0 \ \ c{:}0$$

may be observed. True to its name, the two-place buffer allows the number of input communications to exceed the number of output communications by two.

A quite different program is $TEE(a, b, c)$:

$$(a?W \ \& \ b!W \ \& \ c!W)\cdot(BUF_1(a, b) \mid\mid BUF_1(b, c))$$

It is a two-place buffer where the intermediate channel b is not concealed, but declared as an output port:



A communication along a channel that connects a single sender to multiple receivers is sometimes called a *broadcast* . A broadcast requires simultaneous participation of the sender and all receivers. A possible observation of $TEE(a, b, c)$ is:

$$a{:}3\ \ b{:}3\ \ a{:}9\ \ c{:}3\ \ b{:}9\ \ c{:}9\ \ a{:}0\ \ b{:}0\ \ c{:}0$$

The external behavior of the program below is identical to that of $BUF_2(a, c)$. The program is named $WAG(a, c)$ [vBRS88], because it behaves in a wagging fashion internally:

$$(a?W\ \&\ c!W)\cdot|[x, y:\ \textbf{var}\ W\ |\ a?x; \#[(a?y\ ||\ c!x); (a?x\ ||\ c!y)]]|$$

Inputs are alternatingly written into variables x and y. Similarly, the outputs are alternatingly read from the same variables. After the first input $WAG(a, c)$ may proceed with a second input ("buffer full") or with an output ("buffer empty"). A second input must then be followed by an output or vice versa. Etcetera.

$WAG(a, c)$ is interesting, because its behavior cannot be distinguished from that of $BUF_2(a, c)$. Still, the compiled circuits differ considerably, as do their cost and performance. This will be shown in the next sections.

1.2 Some simple handshake circuits

This section presents the handshake circuits for the various buffer programs of the previous section. Its main purpose is to develop an intuitive understanding of the operation of handshake circuits and the way they are generated from Tangram programs. The formal definition of handshake circuits is presented in Chapter 3. The translation method is described in depth in Chapter 6.

Handshake circuit for $BUF_1(a, b)$

Figure 1.1 shows a handshake circuit for $BUF_1(a, b)$. It consists of 5 handshake components (depicted by circles), 5 channels (labeled c, d, e, wx and rx) and 3 ports (labeled $\triangleright$, a and b). Handshake components communicate through (handshake) ports. A channel connects one passive port (depicted by an open circle) to one active port (depicted by a fat dot). The communication along these channels is by means of a simple two-phase handshake protocol, in which the active side requests for a communication and the passive side responds by returning an acknowledgement.

In the handshake circuit for $BUF_1(a, b)$ the active ports a and b correspond to the Tangram ports with these names. The passive port $\triangleright$ (pronounced as "go")

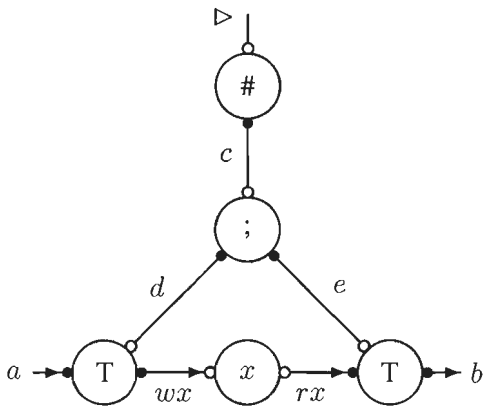


Figure 1.1: Handshake circuit for $BUF_1(a, b)$.

is the activation port of the handshake circuit. The environment activates the buffer by a request along $\triangleright$. Only in the case of a terminating program, which $BUF_1(a, b)$ is not, does the handshake circuit acknowledge termination through the same port.

The handshake component labeled with a semicolon is a *sequencer*. Once activated along c it sequentially performs handshakes along d and e, before it returns an acknowledgement along c. It implements the semicolon that separates the input and output commands in the Tangram program. Unless explicitly indicated otherwise, the activation of the two active ports is counter-clockwise.

The component labeled with a '#' implements infinite repetition and is therefore called a *repeater*. Once activated along $\triangleright$ it repeatedly executes handshakes along c, causing the repeated activation of the sequencer. The repeater never returns an acknowledgement along $\triangleright$.

Component x is a *variable*. A value can be written into x by sending it along channel wx. The acknowledgement along wx signals completion of the write action. Similarly, reading the variable starts by sending a request along rx (against the direction of the arrow). Component x responds by sending the most recently written value.

The two components labeled with a T are so-called *transferrers*. A request along d results in an active fetch of a value along a; this value is subsequently passed actively along wx. The left transferrer implements $a?x$ and the right transferrer implements $b!x$.

Observe that the structure of the handshake circuit of $BUF_1(a, b)$ clearly

reflects the syntactic structure of the Tangram program.

Handshake circuits for $BUF_2(a, c)$

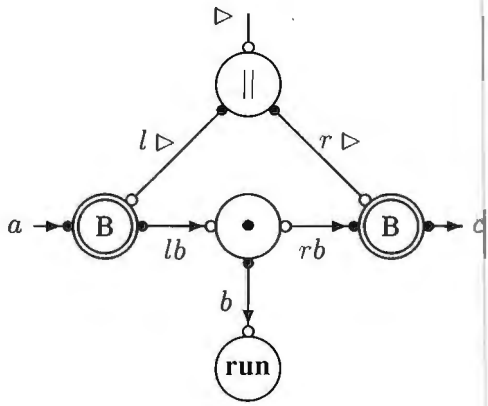


Figure 1.2: Handshake circuit for $BUF_2(a, c)$.

A handshake circuit for $BUF_2(a, c)$ is shown in Figure 1.2. It introduces three handshake components. The two concentric circles enclosing a B represent instances of the handshake circuit of the one-place buffer. The two one-place buffers are activated along $l \triangleright$ and $r \triangleright$ at the same time by the parallel component after a request on $\triangleright$. Only when the parallel component receives an acknowledgement through both its active ports it will acknowledge through $\triangleright$.

The handshake component labeled with a bullet is a *synchronizer*. It implements the concept of "communication and synchronization" of Tangram. If a request for communication arrives along both lb and rb the message arriving along lb is actively output along b. A subsequent acknowledgement along b results in a concurrent acknowledgements along lb and an output along rb.

The concealment of the Tangram channel b is realized by connecting a **run** component to handshake channel b. This component simply acknowledges each message it receives. Removing component **run** results in a handshake circuit for program $TEE(a, b, c)$, with output ports a and b.

By expanding the two one-place buffers in the circuit of Figure 1.2 the handshake circuit of Figure 1.3 is obtained. The circuit clearly reflects the syntactic structure of the original program. The applied translation method is *syntax directed* in that it closely follows the syntactic composition of the program in

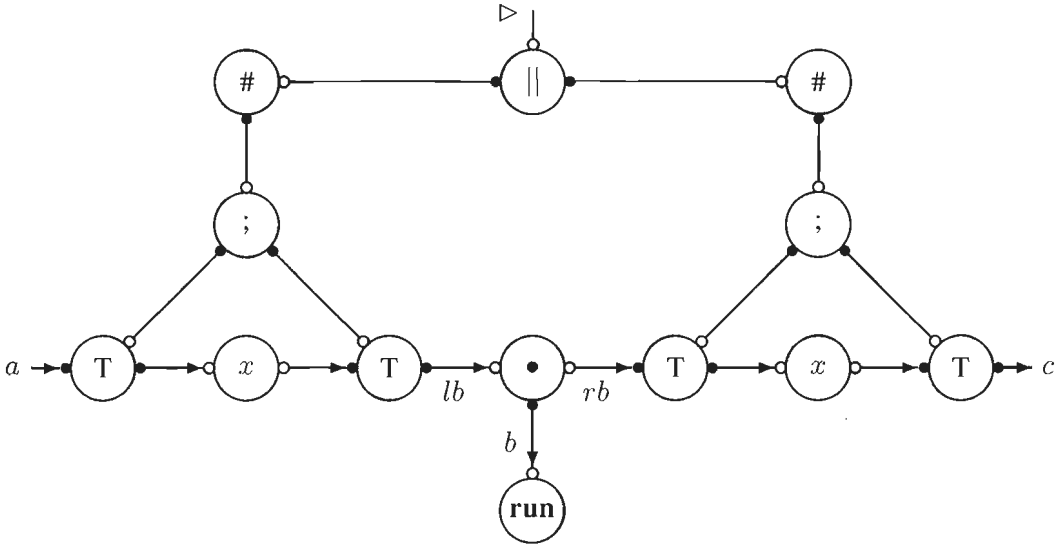


Figure 1.3: Expanded handshake circuit for $BUF_2(a, c)$.

constructing the corresponding handshake circuit.

Such a syntax-directed translation may incur inefficiencies where subcircuits are combined in a way that only depends on their syntactic relation. Such inefficiencies can be removed by replacing small subcircuits by equivalent but cheaper subcircuits. This form of substitution is known as peephole optimization. One form of peephole optimization can be applied to the buffer of Figure 1.3: the result is shown in Figure 1.4. The component labeled '•' is again a synchronizer.

The handshake components introduced so far all implement Tangram primitives. Given the relatively small number of such primitives, the set of handshake components is modest in size. By providing an "equivalent" VLSI circuit for each handshake component and by wiring them according to the structure of the handshake circuit a VLSI circuit can be obtained. The circuits for many handshake components are simple and fixed. For handshake components such as variables and transferrers the circuit structure depends on the number of bits required to encode the relevant data types.

Handshake circuits are clockless. All synchronization is explicit by means of handshake actions between neighboring components. The scheduling problem of assigning a time slot to every primitive action is thus avoided. Furthermore,

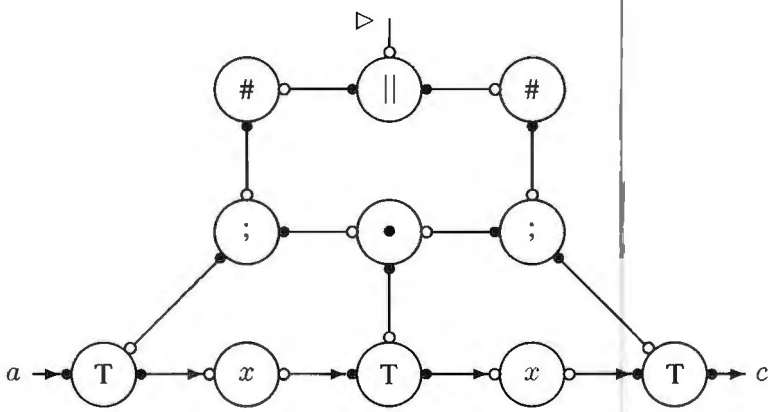


Figure 1.4: Optimized handshake circuit for $BUF_2(a, c)$.

the absence of global synchronization avoids the timing overhead of aligning all primitive actions to clock transitions. Clockless operation combined with the locality of data and control make handshake circuits potentially faster than synchronous circuits.

Although the buffers are about the simplest Tangram programs one can think of, the design of *synchronous* "elastic" buffers offers considerable challenges. In particular, the synchronization of clocked input and output actions with an asynchronous environment involves complex circuitry and fundamental reliability problems [Ano73,Sei80].

Handshake circuit for $WAG(a, c)$

A handshake circuit for $WAG(a, c)$ is presented in Figure 1.5. The three components labeled '|' are so-called *mixers*. They have in common that handshakes through their passive ports are passed to their active ports.

The component connected to a behaves like a demultiplexer. A request from either passive port is passed along a. The incoming message is passed to the side of the request.

The mixer connected to c passes the incoming message from one of its two passive input ports to the active output. The acknowledgement along b is passed to the side of the last incoming message (cf. multiplexer).

The third mixer is a multiplexer for synchronizing handshakes only. It allows

the transferrer connected to its active side to be activated by either of the two handshake components connected to its active ports.

These mixers make the wagging implementation of the two-place buffer more expensive in area than the ripple implementation.

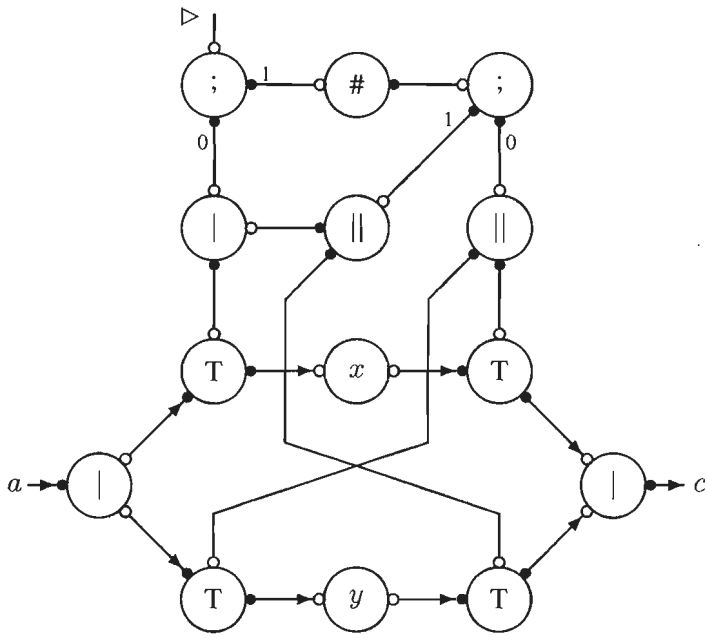


Figure 1.5: Optimized handshake circuit for $WAG(a, c)$. The numbers near the sequencers indicate the order of activation.

1.3 Cost-performance trade-offs

The programs for $BUF_2(a, c)$ and $WAG(a, c)$ in Section 1.1 are functionally identical. The corresponding handshake circuits of Figures 1.4 and 1.5, however, differ considerably. In general, a range of Tangram programs can be designed that satisfy a single functional specification. The corresponding compiled circuits will differ in cost and performance. The best Tangram program is then the one that results in the smallest compiled circuit that satisfies the specified performance requirements.

With this view on VLSI programming it is important that for a given functional specification a range of Tangram programs can be constructed, covering a wide part of the cost-performance spectrum. An N-place shift register serves as a vehicle to demonstrate the idea.

Specification

An N-place shift register, $N \geq 1$, is a component with an input port, a, and an output port, b, of the same data type, with the following behavior:

- communications along a and b strictly alternate, starting with an output along b;
- the sequence of values along b consists of N unspecified values, followed by the sequence of input values.

This shift register will be denoted by $SR_N(a, b)$. Shift registers are common building blocks in VLSI systems, e.g. in various types of digital filters in signal-processing applications. The processed messages are usually called samples.

Note that in both aspects the behavior of a shift register differs from that of an N-place buffer. For an N-place buffer the difference in the number of a and b communications may vary in time over the range $[0..N]$. Also, the sequence of output values is a plain copy of the input sequence.

Implementations of shift registers will be compared in cost and performance. Cost ultimately denotes silicon area and performance refers to the sample rate, i.e. the number of processed samples per second. In a final comparison nine different VLSI programs of an 8-place 8-bit shift register will be compared in terms of transistor count and average cycle time of the corresponding VLSI circuits. The transistor count is a reasonable measure for the silicon area, by which we ignore the variation observed in wiring area. The cycle time is the time between two successive inputs (or outputs) and is the inverse of the sample rate.

A cheap realization

The simplest realization of $SR_1(a, b)$ is denoted by $SRA(a, b)$ and is defined by the Tangram program

$$(a?W \;\&\; b!W)\cdot|[x : \textbf{ var } W \mid \#[b!x; a?x]]|$$

where W denotes the data type of the samples. Note that the repetition command closely resembles that of a 1-place buffer. The only difference is in the order of the input and output command.

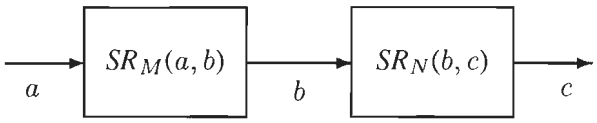


Figure 1.6: $SR_{(M+N)}(a,c)$ composed of two shift registers with a smaller capacity.

For the construction of an N-place shift register a well-known *cascade property* of shift registers is used (see Figure 1.6):

$$|[b : \textbf{chan } W \mid SR_M(a,b) \mid\mid SR_N(b,c)]| = SR_{(M+N)}(a,c)$$

A realization for $SR_N(a,b)$ can now be obtained by cascading N instances of *SRA*. This solution will be denoted by A^N.

Note that A^N is capable of producing N outputs before doing its first input. Moreover, after these initial N outputs the behavior is that of an N-place buffer. On closer inspection, is A^N really an implementation of $SR_N(a,b)$?

It depends. If the environment relies on the alternation of b and a communications then definitely not. If the environment enforces this alternation, A^N is an acceptable realization. In order to avoid further confusion, the first requirement of the specification of $SR_N(a,b)$ is redefined as: the following composition must not deadlock:

$$SR_N(a,b) \quad || \quad |[x,y : \textbf{var } W \mid \#[b?x \mid\mid a!y] \ ]|$$

Note that the specification is relaxed to the extent that the ith input and the ith output may occur concurrently.

What can be said about the performance of A^N? After its first output, the last cell in the cascade is ready to do an input: "it is vacant". This vacancy then propagates backwards to the input of A^N and it takes $N-1$ successive internal assignments before an input action can occur. The cycle time is therefore proportional to N. The time an 8-bit assignment takes will be chosen as a time unit. A reasonable value for this time unit in current VLSI CMOS technologies is 25 nanoseconds. For an 8-place 8-bit shift register the cycle time is then 8 time units, or 200 nanoseconds. It depends on the performance *requirements* whether 200 nanoseconds are acceptable or not.

The cost of A^N is modest. It takes only N variables, which is obviously a lower bound for $SR_N(a,b)$.

Fast realizations

An alternative realization of $SR_1(a,b)$ is $SRB(a,b)$. It consists of an instance of *SRA* and a 1-place buffer:

$$(a?W \;\&\; b!W)\cdot|[x,y:\ \textbf{var}\ W \;\&\; c:\ \textbf{chan}\ W \mid \#[c!x; a?x] \ ||\ \#[c?y; b!y]]|$$

SRB resembles a traditional synchronous shift register composed of master-slave flipflops. The *SRA* part assumes the role of the master, the 1-place buffer that of the slave. Input and output actions may overlap in time. In contrast to a synchronous shift register (and with *SRA* for that matter) *SRB* may start with an input. Shift register $SRB(a,b)$ can be rewritten into $SRC(a,b)$:

$$(a?W \;\&\; b!W)\cdot|[x,y:\ \textbf{var}\ W \mid \#[(b!y \ ||\ a?x); y := x]]|$$

SRB and *SRC* are very close in terms cost and performance (*SRC* is slightly cheaper, because of its simpler control structure).

By cascading N instances of *SRC* we obtain a second realization of $SR_N(a,b)$. This realization will be denoted by C^N. Because each *SRC* section has its own vacancy, the behavior of C^N is markedly different from that of A^N. For the analysis it is assumed that when the environment is ready to participate in an input or output action, it does so without delay. Then the input and output actions of each individual *SRC* occur simultaneously, and all N stages operate in harmony. As a result the sample rate is independent of N, and the cycle time amounts to only two time units (50 nanoseconds). The price for this nice performance is substantial: C^N requires $2N$ variables, twice that of A^N.

Given the substantial difference in cost and performance between A^N and C^N one may wonder if intermediate solutions exist. Indeed, by introducing one or more *SRC* cells in a sequence of *SRA* cells, intermediate solutions can be obtained of the form $C^K A^{N-K}$, with $0 \leq K < N$. The solid circles in Figure 1.8 indicate the average cycle time and transistor counts for five shift registers of this kind, with K equal to 0, 1, 2, 4 and 8 respectively. The cycle times were obtained by simulation of the compiled handshake circuits. The timing models of the handshake components have been calculated from the timing characteristics of their constituent VLSI operators.

The realizations described so far have in common that the messages ripple through a cascade of N cells. Hence, they will be referred to as *ripple* shift registers.

Still faster realizations

One may wonder whether C^N is the fastest possible shift register. Equivalently, is two time units the minimum cycle time? By putting two shift registers in parallel, and by serving them alternatingly, faster shift registers can be constructed (see Figure 1.7). Shift registers based on this structure will be referred to as *wagging* shift registers. In order to keep matters simple, N is restricted to even values.

$SRD(a, c, e)$ de-interleaves the incoming sequence by sending the incoming values alternatingly along c and e:

$$(a?W \;\&\; c!W \;\&\; e!W)\cdot|[x, y : \textbf{var}\ W \mid \#[(c!x \parallel a?y); (e!y \parallel a?x)]]|$$

$SRE(d, f, b)$ interleaves the incoming sequences by receiving inputs alternatingly along d and f:

$$(d?W \;\&\; f?W \;\&\; b!W)\cdot|[x, y : \textbf{var}\ W \mid \#[(b!x \parallel d?y); (b!y \parallel f?x)]]|$$

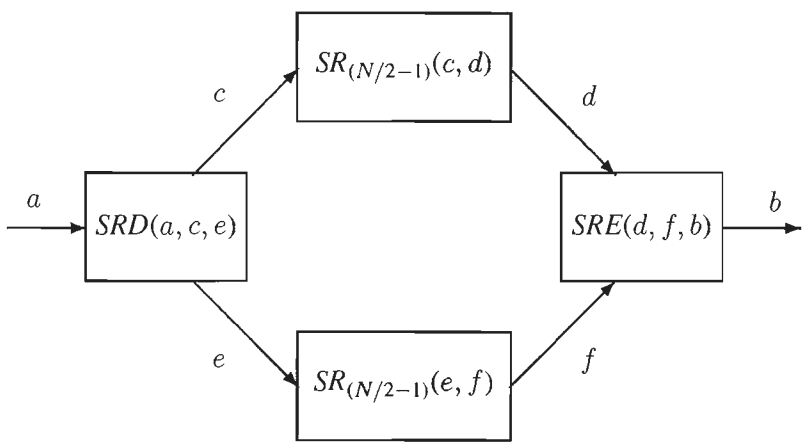


Figure 1.7: The wagging shift register.

Both *SRD* and *SRE* have a cycle time of one unit, measured at the input and output respectively. Unfortunately, due to some additional overhead in control and data routing, the real cycle time is somewhat larger. For 8-bit messages 30 nanoseconds is realistic.

For the two parallel shift registers ripple implementations can be used, e.g. composed of *SRA* and *SRC* cells. A regular communication behavior is obtained by taking identical cell sequences for $SR_{(N/2-1)}(c, d)$ and $SR_{(N/2-1)}(e, f)$.

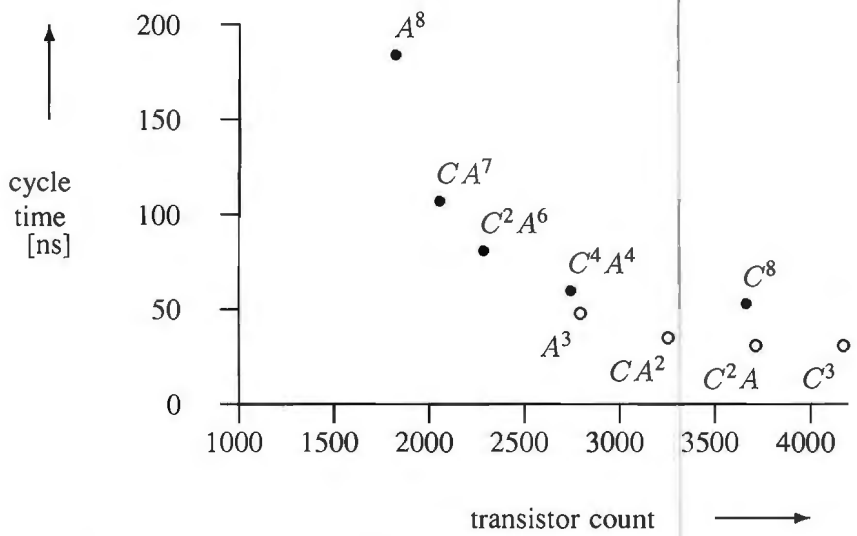


Figure 1.8: Cycle time versus transistor count for nine different 8-place 8-bit shift registers. Solid circles represent ripple solutions, open circles wagging solutions.

In Figure 1.8 the compilation and simulation results for four triplets for $SR_3(c,d)$ and $SR_3(e,f)$ (from slow/small to fast/large): A^3, CA^2, C^2A and C^3. Due to the aforementioned overhead, C^2A is barely an improvement over CA^2. For the same reason C^3 offers no advantages over C^2A.

Discussion

One type of realization has not been considered: array-based solutions. Such an array can be mapped on a Random Access Memory (RAM). Array-based realizations are attractive for large N, because of the very small area/bit of a RAM. For small N, array-based solutions are less attractive due to an area overhead of circuitry for timing, control and addressing.

Despite the simplicity of the specification of $SR_N(a,b)$ an interesting range of implementations has been realized. If we ignore the wagging solution with cell sequence C^3, all implementations have different cycle times and different costs. Among these eight solutions there is *no* best solution. Depending on throughput requirements, each of these eight implementations may be the best, except C^8.

Even totally different considerations may part of the cost/performance trade-off, for example energy consumption. An 8-bit assignment consumes approxi-

mately a quarter of a nanoJoule. A reasonable measure for the energy consumption of a ripple buffer is then the number of moves made by a single message from input to output, viz. $K+N$. The *power* consumption of A^8 running close to its maximum speed (5 Mhz sample rate) will then be approximately 10 mWatt. For C^8 at 20 Mhz this amounts to 80 mWatt. Wagging solutions, on the other hand, are markedly economic in their energy consumption, because the path traversed by a message is only half in length compared to that of ripple solutions. For instance, a message takes five moves to ripple through A^3. At 20 Mhz this results in a power consumption of 25 mWatt. Of course, more accurate estimates for size, timing and power need to be provided by the silicon compiler and simulation tools.

These concerns for cost and performance make VLSI programming different from and also more difficult than conventional programming. Especially high-performance systems (e.g. digital video systems) may require detailed performance analysis. But also for low-performance systems (e.g. digital audio systems) with critical requirements on silicon area, balancing the performance of subsystems is important.

From a VLSI-circuit perspective, these asynchronous shift registers also provide an interesting insight. A^N requires one latch per section per bit whereas a master-slave flipflop requires two. In principle, no solution based on master-slave flipflops can beat A^N in circuit size. Mimicking the behavior of A^N with synchronous circuits requires complex timing/control circuitry.

Although the sample rate of A^N is low, the vacancy travels at maximal speed from output to input. With a few *SRC* cells, the internal timing behavior becomes highly irregular. This form of irregularity is hard to capture in clocked circuits.

In complex VLSI systems the different input and output ports often have different sample rates. Sometimes the samples are offered or consumed irregularly in time. Even if these rates are constant and identical, this need not be so for internal channels. Certain subcomputations have data-dependent processing times, or are invoked at irregular intervals, e.g. for handling exceptional situations. In such situations the absence of a clock opens new architectural possibilities and trade-offs.

1.4 More examples

Buffers and shift registers are not very interesting from a data-processing point of view: the sequence of output message is basically a copy of the sequence of input messages. This section introduces Tangram constructs by which more interesting

programs can be described. Various small programs are used as illustrations. Where appropriate, the corresponding handshake circuits are given.

An adder

$ADD(a, b, c)$ is a simple process that repeatedly accepts two values of type W along its two inputs a and b and outputs their sum along c:

$$ADD(a, b, c) = (a?W \;\&\; b?W \;\&\; c!W)\cdot|[x, y : \textbf{var}\ W \mid \#[(a?x \parallel b?y); c!(x + y)]]|$$

Expressions such as $x + y$ may occur in assignments, outputs commands and guards (see later in this chapter). It will be assumed that the evaluation of an expression always terminates. However, if the result value does not "fit" its destination, the consequent behavior is not specified: the program may deadlock, or may proceed in some erratic way. *ADD* continues properly as long as the value of $x + y$ is of type W.

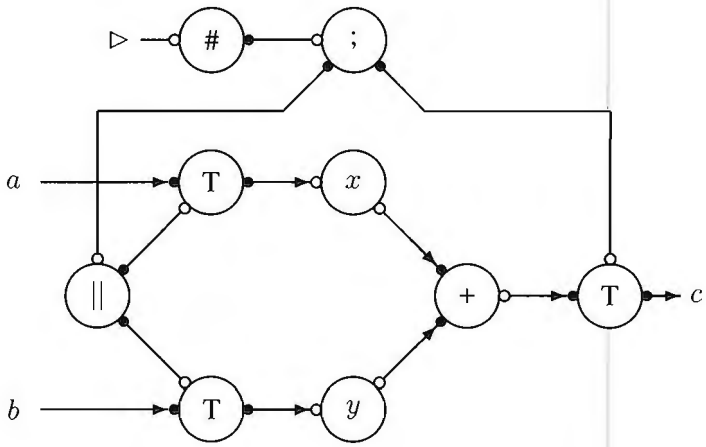


Figure 1.9: Handshake circuit for $ADD(a, b, c)$.

The handshake circuit for *ADD* is presented in Figure 1.9. The handshake component labeled '+' is an *adder*. The expression $x + y$ is evaluated in a demand-driven fashion:

0. a request for a sum is passed to the passive output of the adder;

1. the adder forks this request to its active inputs;
2. the input values arrive at the inputs of the adder;
3. their sum is output along the output.

A simple FIR filter

A Finite Impulse Response (FIR) filter is a process with a single input and a single output. The input and output communications strictly alternate, starting with an input. For a FIR filter of order N the output values are specified as follows. The value of the ith output, $i \geq N$, is generally a weighted sum of the N most recent input values. The N weights are generally referred to as the filter coefficients. The first N output values are left unspecified.

A very simple FIR filter of order N can now be constructed by connecting *ADD* with a shift register:

$$(a?W \ \& \ b!W)\cdot|[b : \textbf{chan}\ W \mid ADD(a,b,c) \mid\mid SR_N(a,b)]|$$

The ith output, $i \geq N$, is the sum of the ith input and the input with index $i - N$. This composition is depicted in Figure 1.10. Clearly, the input channel a is connected to both *ADD* and *SR*. In general, any number of receivers may be connected to a channel. The connected receivers must all participate in each communication along that channel. This is another example of broadcast. There may be at most one sender.

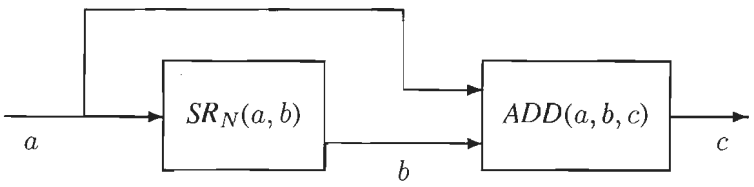


Figure 1.10: A simple FIR filter constructed from $SR_N(a,b)$ and $ADD(a,b,c)$.

A more general FIR filter program is discussed in [vBRS88], in which the degree of parallelism is parameterized. The program is a linear systolic array of N **div** M cells, where M is a measure of the grain size of the parallelism. If $M = N$ the program is sequential and requires a single multiplier. The other extreme solution is $M = 1$: an array of N cells guarantees maximum throughput, but requires N multipliers.

A median filter

A median filter repeatedly outputs the median value of the three most recent inputs. (Similar to the FIR filter, the sequence of output messages starts with a few unspecified values.) A Tangram program for the median filter is given by:

```
(a?W & b!W).
|[x, y, z : var W & xy, yz, zx : var bool
|  #[ (z := y; y := x; a?x || yz := xy)
     ; xy := x ≤ y || zx := z ≤ x
     ; if  zx = xy  →  b!x
       []  xy = yz  →  b!y
       []  yz = zx  →  b!z
       fi
  ]
]|
```

The program segment **if** .. **fi** is a selection command [Dij75]. The **if** **fi** bracket pair encloses three so-called guarded commands. Guarded commands have the form $B \rightarrow S$, where B is a Boolean expression and S a command. The execution of the selection command starts with the evaluation of the guards. If all guards are *false* the henceforth behavior program is left unspecified. If at least one guard evaluates to *true* the command corresponding to a *true* guard is executed. If more than one guard is *true*, the choice which of command to execute is not specified. This nondeterminism can be resolved at compile time, or even at run time. Of course, for purposes of efficiency the programmer may strengthen the guards to make them non-overlapping.

The following may help to understand the Tangram description of the median filter. Just prior to the execution of the selection command, the variables x, y and z contain the three most recent input values, in increasing age. At that point the three Boolean variables xy, yz and zx have the values $x \leq y$, $y \leq z$ and $z \leq x$. The expression $zx = xy$ is then equivalent to "x is the median value". The two other guards can be read similarly.

Note that if $x = y = z$ all three guards evaluate to *true*. The corresponding nondeterministic selection of one of the three output commands cannot be observed externally. However, the internal operation does depend on how the nondeterminism is resolved.

The median filter nicely demonstrates an advantage of this form of command selection. The symmetry among the three guards can only be captured in a language with explicit and overlapping guards for the three alternatives. The

reader may try for instance an if-then-else command to replace the above selection command to convince himself.

The handshake circuit for the **if fi** section of the median filter program is depicted in Figure 1.11. The write ports of the variables are left unconnected. After an activation along ▷ the selection command is executed in two phases.

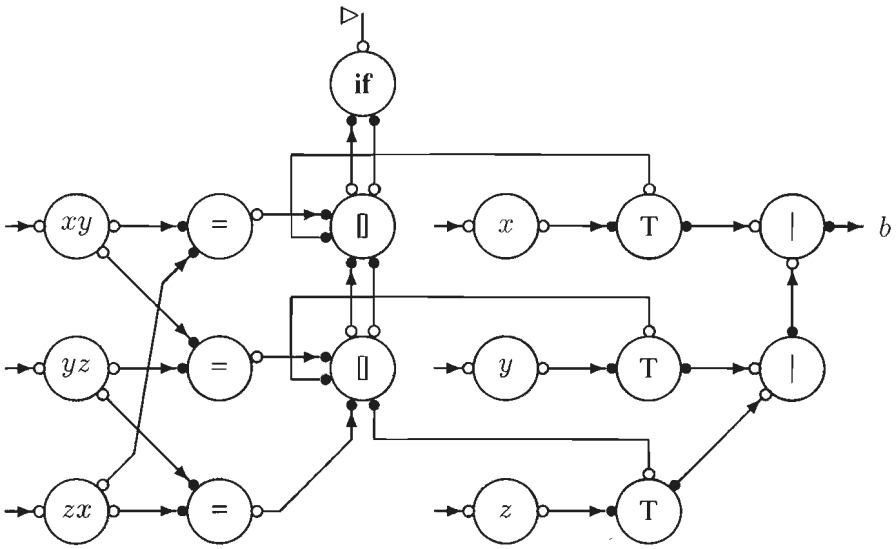


Figure 1.11: Handshake circuit for the **if fi** section of the median filter.

In the first phase the **if** component actively collects the disjunction of the guards. The component labeled ' [] ' passes on a request on its output to both its Boolean inputs; upon the reception of these Booleans their disjunction is transmitted along its output. Note that the guards are evaluated in parallel.

The effect of the second phase depends on the collected disjunction of the guards. If this value equals *false*, the **if** component remains passive and the circuit deadlocks. If the value equals *true*, as is always the case for the median filter, component **if** activates the topmost [] component. This component activates the circuit corresponding to an input from which it received the value *true*. In this solution the nondeterminism is resolved in the [] component.

This implementation scheme works for an arbitrary number of guards. When the [] components are organized according to a binary tree, the computation of

the disjunction of the guards and the selection of the appropriate command can be done in $O(\log N)$ time, where N denotes the number of guards.

The handshake component labeled '=' is another example of a binary operator. The variables xy, yz and zx have two read ports, from which concurrent read actions are allowed.

The median filter is an instance of a so-called rank-order filter. After each input value the rank-order filter outputs the value with rank K among the last N inputs ($0 \leq K < N$). A value has rank k among N values if it has position k in the ascendingly ordered list of these N values. The median filter is a rank-order filter with $N = 3$ and $K = 1$. [KR89,KU91] present programs for rank-order filters. Both solutions are linear systolic arrays of N cells. Except for the two end cells, all cells are identical. The cells communicate with neighboring cells in a regular, systolic manner.

The greatest common divisor

Program *GCD* repeatedly computes and outputs the greatest common divisor of the two most recent inputs:

```
(a?W & b?W & c!W).
|[x, y : var W
| #[ (a?x || b?y)
    ; do  x > y  →  x := x − y
       []  y > x  →  y := y − x
      od
    ; c!x
  ]
]|
```

The algorithm goes back to Euclid; this particularly elegant version is based on [Dij76]. The program segment **do** .. **od** is a *guarded repetition*, a generalization of the well-known **while** command. As long as at least one guard evaluates to *true* one of the *true* guards is selected and the corresponding command is executed. If all guards fail, the repetition command terminates. If one of the inputs equals 0 the guarded iteration will not terminate.

In [Dij76] a similar algorithm is given for computing the least common multiple. The Tangram program for a sequential multiplier [vBS88] also resembles *GCD*. The number of cycles required to compute the greatest common divisor strongly depends on the two input values. The computation time and energy are proportional to this number.

Note that the communication behavior of *GCD* is identical to that of *ADD*. The handshake circuits have the same external ports, viz. an activation port, two input ports and one output port. The fact that *GCD* contains an iterative algorithm is completely hidden for the user of the circuits. The proposed method of compiling Tangram programs into handshake circuits leads to a form of distributed control. Information is kept local, with the associated advantages of shorter wires and minimum timing overhead.

Modulo-N counters

The modulo-N counter is presented to introduce the choice command. There are two ways in Tangram by which the environment can influence the future course of action of a program. Firstly, the environment may select the value to send through an input port. The incoming value is stored in a variable and may subsequently occur in the guards of a guarded command. The input value may thus determine the future pattern of communications and computations.

Secondly, the environment may have the *choice* among a set of ports through which it may synchronize or communicate. The binary form of this choice is exemplified by the program $CE\ (a, b, c, d)$ (*CE* is an acronym for Count Even, as will become clear later):

$$(a\ \&\ b\ \&\ c\ \&\ d)\cdot\#[[c; a; a \mid d; b]]$$

Operator ';' binds more strongly than '|'. The environment is repeatedly offered the choice between a synchronization on c or on d. These two commands act as guards in the choice command. For each c, process *CE* performs two synchronizations on a; for each d, it performs only one synchronization on b.

Using the choice command, a program for a modulo-N counter will be constructed ($N \geq 1$). The program has two external synchronization ports, a and b. A modulo-N counter repeatedly performs N synchronizations on a followed by a single on b. Let this behavior be denoted by

$$\#[\#N[a]; b]$$

The simplest counter is a modulo-1 counter $C1$:

$$\#[a; b]$$

For even values of N the modulo-N counter can be written as

$$\#[\#M[a; a]; b]$$

where $M = N$ **div** 2. By introducing a modulo-M counter this can decomposed into

$$|[c, d : \textbf{chan} \mid (CE\ (a, b, c, d) \mid\mid \#[\#M[c]; d])]|$$

In words, the modulo-M counter performs M synchronizations on c, which are effectively doubled by CE and passed through a. A closing d is simply passed by CE as b.

For odd values of N, $N > 1$, a similar decomposition can be derived, in which $CO\ (a, b, c, d)$ extends a modulo-M counter to a modulo-N counter:

$$(a\ \&\ b\ \&\ c\ \&\ d)\cdot\#[a; [c; a \mid d; b]]$$

A program for the modulo-N counter can now be constructed with the cells $C1$, CE and CO for any value of N. This requires $^2\log N + 1$ cells. For instance, the number 11 can be written as $1 + 2 * (1 + 2 * (2 * 1))$, yielding the following program for an modulo-11 counter

$$\begin{array}{l} |[\ \ c, d, e, f, g, h : \textbf{chan} \\ \mid\ \ CO\ (a, b, c, d) \mid\mid CO\ (c, d, e, f) \mid\mid CE\ (e, f, g, h) \mid\mid C1(g, h) \\ ]| \end{array}$$

Figure 1.12 depicts handshake circuits for CE and CO. Note that both circuits consist of the same handshake components, albeit connected differently.

The choice component (labeled '[|]') implements the choice construct. After activation through its topmost passive port, the choice component is prepared to participate in a handshake through one of the other two passive ports. The choice between the two is made by the environment. Subsequently, the choice component actively handshakes on the active port opposite to the selected passive port, and then returns an acknowledgement along its activation channel.

There is something special about this type of modulo-N counter: the rate of counting is independent of N and independent of the state of the counter. (It is the head cell that determines the rate of counting; the next cell needs to count at only half the rate.) This property seems to be unique, as traditional counters slow down with $\log N$. The resulting rate of counting is probably as fast as can be. In [Kes91a] the program has been carried over to the domain of synchronous circuits. Presumably, this systolic modulo-N counter can be carried over to the domain of synchronous circuits. But, would it have been invented without the Tangram example?

Counting also appears to be a very useful primitive in a VLSI programming language. In many cases an action has to be repeated N times, where N is a

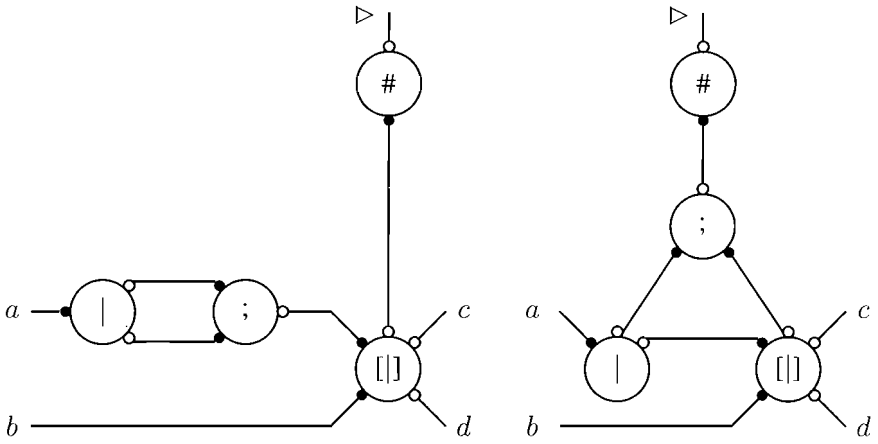


Figure 1.12: Handshake circuits for $CE\ (a, b, c, d)$ and $CO\ (a, b, c, d)$.

constant (e.g. the number of pixels in a video line, the number of samples in a block, or the number of bits in a word). For this reason the construct $\#N[S]$ is part of Tangram and means: repeat S exactly N times. The implementation of this construct contains a handshake circuit composed of the cells of Figure 1.12.

This section on choice and counters ends on a less fortunate note. It turns out that the full implementation of the choice construct is fairly complicated. It was decided *not* to include its treatment in this thesis.

1.5 Epilogue

The programs and handshake circuits of this chapter give an impression of Tangram and its compilation into handshake circuits. The following remarks place the preceding experiments in a wider perspective.

Full Tangram

The programs of this chapter were written in a subset of Tangram. The main omissions of this chapter are

- functions and procedures;
- the data type constructors tuple and array;

- operators such as tuple forming, tuple selection, type cast and fit;
- explicit sharing of functions and procedures.

With these extensions arithmetic operators such as $*$, **div** , **mod** and shift can be defined as Tangram functions. This also extends to operators on Galois-field symbols [SvBB*91], complex numbers, and other forms of less common arithmetic.

The circuits corresponding to functions and procedures can be shared, provided that they are used sequentially. Sharing avoids unnecessary duplication of resources, but has an overhead in terms of (de-)multiplexing circuitry. Whether sharing actually saves circuitry and whether the associated penalties in delays can be afforded depends on the size of the shared resource and the timing specification. e.g. the sharing of a large multiplier is more likely to be advantageous than the sharing of a one-bit adder. In line with the choice for transparent compilation, Tangram supports explicit sharing of functions and procedures.

OCCAM [INM89] is another CSP-based language. Essential differences between Tangram and OCCAM are related to differences in the implementation target: VLSI circuits versus Transputer networks. In particular, OCCAM does not support broadcast and sharing, and has limited facilities for type construction and operators.

Applications

To which extent are Tangram and handshake circuits general purpose? Neither Tangram nor handshake circuits contain constructs or notions that are specific to certain application areas such as controllers or signal processors.

Applications range from very small circuits (e.g. the buffers of Section 1.1) to quite sizable circuits (e.g. a graphics processor [SvB88], or a high-throughput RSA encryptor [Kes91b]).

Tangram and handshake circuits support fine-grained parallelism (as for instance in systolic arrays [Rem87]), and coarse-grained parallelism (e.g. in Compact Disc subsystems [KS90,KvBB*92]).

Many diverse applications were cited along with the programs. Other examples of programs in CSP-based notations include: a systolic block sorter [vBRS88], a micro-processor [MBL*89], and a regular-expression recognizer [KZ90].

Chapter 2

Handshake processes

2.0 Introduction

A handshake is a means of synchronization among communicating mechanisms. In its simplest form it involves two mechanisms connected by a pair of so-called *links*, one for sending signals and one for receiving signals. The sending of a signal and the reception of a signal are atomic actions, and constitute the possible events that may occur between the mechanisms.

A signal sent by one mechanism is bound to arrive at the other mechanism, after a finite, *non-zero* amount of time. Hence, this form of communication is asynchronous; the sending and the arrival of a signal correspond to two distinct events. It is assumed that a link allows at most one signal to be on its way. Consequently, a signal sent must arrive at the other end of the link before a next one can be sent. When the traveling time of a signal along the link is unknown, the *only* way to know that a signal has arrived at the other side, is to be so informed by the other mechanism via a communication along the other link.

Such a causally ordered sequence of events is called a handshake. The two mechanisms involved play different (dual) roles in a handshake. One mechanism has the *active* role: it starts with the sending of a request and then waits for an acknowledgement. The other mechanism has the *passive* role: it waits for a request to arrive and responds by acknowledging. A handshake realizes synchronization among mechanisms; it can and will occur only if both mechanisms are ready to participate.

Some useful terminology is introduced next. The pair of links forms a so-called *channel*; the two terminals of a channel are called *ports*. This study only considers channels with a fixed division of the passive and active roles during

handshakes. Hence, a port is either passive or active, depending on its handshake role.

In a more general setting, a channel consists of two finite, non-empty sets of links. A handshake then consists of a request along a link of one set, acknowledged by a signal along a link from the other set. By choosing a particular link, a mechanism can convey additional information. For instance, a pair of output links suffices to communicate a Boolean value with each handshake. A mechanism may be connected to several other mechanisms, by an arbitrary number of channels, and may therefore have multiple ports, of different activities.

This chapter presents a formalism for mechanisms that interact exclusively by means of handshakes. The central notion is that of *handshake process*, a mathematical object that can be used to specify all possible behaviors of such a handshake mechanism. In Chapter 3 it is shown that handshake processes can also be used to describe the external behavior of handshake circuits, i.e. of networks of handshake processes. In Chapter 5 the semantics of a subset of Tangram is expressed in terms of handshake processes.

In the context of VLSI circuits, a mechanism may correspond to a CMOS circuit, a link to a wire and a signal to a voltage transition. The requirement that at most one signal may be on its way on a single link agrees with the required absence of interference (cf. Section 0.1). Chapter 7 discusses in more detail the relationship between handshake processes and VLSI circuits.

Before we delve into the mathematics of handshake processes we shall review some notational conventions used in this thesis.

2.1 Notational conventions

Functions

A function f in $A \rightarrow B$ has domain A and range B. Function application is denoted with an infix '$\cdot$' as in $f \cdot x$, instead of the more traditional $f(x)$. Operator '$\cdot$' is right binding and it binds more strongly than all other operators except for subscription and superscription (e.g. exponentiation).

Guarded selection

An expression may have the form of a *guarded selection* as in:

$$\begin{array}{lll} \textbf{if} & B_0 & \rightarrow E_0 \\ [] & B_1 & \rightarrow E_1 \\ & \vdots & \\ [] & B_{N-1} & \rightarrow E_{N-1} \\ \textbf{fi} & & \end{array}$$

Guarded expression $B_i \rightarrow E_i$ consists of *guard* B_i and *expression* E_i. A guard is a Boolean expression. The order of the alternatives is irrelevant. In the guarded selections of this thesis we shall see to it that at least one of the guards evaluates to *true*, and that if both B_i and B_j evaluate to *true* expressions E_i and E_j have the same value. For instance, the minimum of two integers, denoted by x **min** y may be defined as

$$\begin{array}{lll} \textbf{if} & x \leq y & \rightarrow x \\ [] & y \leq x & \rightarrow y \\ \textbf{fi} & & \end{array}$$

The notation for guarded selection strongly resembles that of guarded commands [DS90] (cf. the median filter in Section 1.4).

Quantified expressions

Universal quantification is a generalization of conjunction. Its format [DS90] is

$$(\forall \text{ dummies} : \text{range} : \text{term})$$

where

- "dummies" stands for an unordered list of local variables whose scope is delineated by the parenthesis pair;
- "range" is a predicate that delineates the domain of the dummies; and
- "term" denotes the quantified expression.

Similarly, existential quantification (with quantifier $\exists$) is a generalization of disjunction. Quantification over an empty range yields the unit element of the quantifier: *true* for $\forall$ and *false* for $\exists$. When the range is evident from the context it is usually omitted.

The above format is also applied to generalizations of other symmetric and associative binary operators. For instance, the continued union of the set of sets A is denoted by $(\cup a : a \in A : a)$. If $A = \emptyset$ this expression yields $\emptyset$, the unit element of set union.

For set construction a similar format is used, viz

$$\{\text{dummies} : \text{range} : \text{term}\}$$

For instance, the image of function f with domain D can be written as

$$\{x : x \in D : f \cdot x\}$$

Derivations

Let $\preceq$ be a partial order, i.e. $\preceq$ is a reflexive, antisymmetric and transitive relation on some set. For convenience's sake we often abbreviate a conjunction of the form $(E = F) \wedge (F \preceq G)$ to $E = F \preceq G$. In particular, a proof of $E \preceq G$ may take the form (cf. [DS90])

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

for some judiciously chosen F. The above example naturally generalizes to a list of (in-)equalities. From a derivation of the form $E \preceq F \preceq \cdots \preceq E$ it may be concluded that all related elements are equal, on account of transitivity and antisymmetry of $\preceq$. Note that implication ('$\Rightarrow$') and set inclusion ('$\subseteq$') are examples of partial orders. An example of a derivation is given below.

Closures

Let $(A, \preceq)$ be a preordered set, i.e. $\preceq$ is a reflexive and transitive relation on A. We shall often base a closure operator and a closedness predicate on such a preorder as follows.

Definition 2.0 (closure, closed)

0. For any subset B of A the $\preceq$-*closure* of B in A, denoted by $B^{\preceq}$, is defined by
$$B^{\preceq} = \{a : a \in A \wedge (\exists b : b \in B : a \preceq b) : a\}$$
This closure is also known as the *downward* closure of B.

1. B is called $\preceq$-*closed*, denoted by $(\preceq)\cdot B$, if $B = B^{\preceq}$.

□

Operator $^{\preceq}$ is indeed a closure operation, since ([DP90] page 36):

Property 2.1

0. $B \subseteq B^{\preceq}$ (extensive)

1. $B^{\preceq} = (B^{\preceq})^{\preceq}$ (idempotent)

2. $B \subseteq C \Rightarrow B^{\preceq} \subseteq C^{\preceq}$ (order preserving)

□

The closure operation binds more strongly than any other operation. As an example of the proof style applied, we prove idempotence of the above closure operation.

Proof of idempotency :

$(B^{\preceq})^{\preceq}$

= { definition of $\preceq$-closure }

$\{a : a \in A \wedge (\exists b : b \in B^{\preceq} : a \preceq b) : a\}$

= { definition of $\preceq$-closure }

$\{a : a \in A \wedge (\exists b : b \in \{c : c \in A \wedge (\exists d : d \in B : c \preceq d) : c\} : a \preceq b) : a\}$

= { calculus }

$\{a : a \in A \wedge (\exists b : b \in A \wedge (\exists d : d \in B : b \preceq d) : a \preceq b) : a\}$

= { trading }

$\{a : a \in A \wedge (\exists b, d : b \in A \wedge d \in B : a \preceq b \wedge b \preceq d) : a\}$

= { $\preceq$ is reflexive and transitive }

$$\begin{array}{ll} & \{a : a \in A \wedge (\exists d : d \in B : a \preceq d) : a\} \\ = & \quad \{ \text{ definition of } \preceq\text{-closure } \} \\ & B^{\preceq} \end{array}$$

□

For later use we mention without proof:

Property 2.2

0. $\emptyset^{\preceq} = \emptyset$, hence $(\preceq) \cdot \emptyset$.
1. $B^{\preceq} \cup C^{\preceq} = (B \cup C)^{\preceq}$.
2. Hence, $(\preceq) \cdot B \wedge (\preceq) \cdot C \Rightarrow (\preceq) \cdot (B \cup C)$.

□

2.2 Handshake structures

Ports and port structures

A handshake through a port consists of two events: a request followed by an acknowledgement. We shall identify these events by symbols, such as r and a. A port consists of a set of request symbols and a set of acknowledgement symbols. These two symbol sets of port p must be non-empty, disjoint and finite, and will be denoted by $\mathbf{0}p$ and $\mathbf{1}p$ respectively. A handshake consists of an occurrence of an event from $\mathbf{0}p$ followed by an occurrence of an event of $\mathbf{1}p$. A port structure is a set of ports, partitioned into a set of passive ports and a set of active ports.

Definition 2.3 (port structure)

0. A *port* p is a pair of disjoint, finite and non-empty sets of symbols $\langle \mathbf{0}p, \mathbf{1}p \rangle$. $\mathbf{a}p$ denotes the symbol set of p, viz. $\mathbf{0}p \cup \mathbf{1}p$.
1. A *port set* P is a finite (possibly empty) set of ports. $\mathbf{a}P$ is the set of symbols of P, viz. $(\cup p : p \in P : \mathbf{a}p)$.
2. A *proper port set* P is port set with disjoint symbol sets:

$$(\forall a, b : a \in P \wedge b \in P : a = b \ \vee \ \mathbf{a}a \cap \mathbf{a}b = \emptyset)$$

3. A *port structure* A is a pair $\langle A^\circ, A^\bullet \rangle$, of port sets, such that $A^\circ \cup A^\bullet$ is a proper port set. A° is called the *passive* port set of A and $A^\bullet$ the *active* port set of A. Note that $\mathbf{a}A^\circ \cap \mathbf{a}A^\bullet$ need not be empty.

   $\mathbf{a}A$ denotes the set of symbols of A, viz. $\mathbf{a}A^\circ \cup \mathbf{a}A^\bullet$. Set $\mathbf{a}A$ is called the *alphabet* of A.

4. Elements of $A^\circ \cap A^\bullet$ are the *internal* ports of port structure A, and elements of $(A^\circ \setminus A^\bullet) \cup (A^\bullet \setminus A^\circ)$ are the *external* ports of A.

5. Port structures A and B are *compatible* if $A^\circ \cup A^\bullet \cup B^\circ \cup B^\bullet$ is a proper port set.

6. The *union* of compatible port structures A and B, denoted by $A \cup B$, is the port structure $\langle A^\circ \cup B^\circ, A^\bullet \cup B^\bullet \rangle$.

7. The *difference* of compatible port structures A and B, denoted by $A \setminus B$, is the port structure $\langle A^\circ \setminus B^\circ, A^\bullet \setminus B^\bullet \rangle$.

□

The symbols $^\circ$ and $^\bullet$ are used as postfix operators on port structures; they bind more strongly than any other operator. If p is a passive port, $\mathbf{0}p$ is the set of the input symbols of p and $\mathbf{1}p$ the set of output symbols of p. For active ports this is the other way around:

Definition 2.4 (input and output symbols)

Let A be a port structure and let $a \in A$. Then

0. $\mathbf{i}a$ denotes the set of input symbols of port a:

$$\begin{array}{lll} \mathbf{i}a = \mathbf{if} & a \in A^\circ & \rightarrow \mathbf{0}a \\ \quad [] & a \in A^\bullet & \rightarrow \mathbf{1}a \\ \quad \mathbf{fi} & & \end{array}$$

1. Operator $\mathbf{i}$ is lifted to port structures by $\mathbf{i}A = (\cup a : a \in A : \mathbf{i}a)$.

2. $\mathbf{o}a$ denotes the set of output symbols of port a:

$$\begin{array}{lll} \mathbf{o}a = \mathbf{if} & a \in A^\circ & \rightarrow \mathbf{1}a \\ \quad [] & a \in A^\bullet & \rightarrow \mathbf{0}a \\ \quad \mathbf{fi} & & \end{array}$$

3. Operator $\mathbf{o}$ is lifted to port structures by $\mathbf{o}A = (\cup a : a \in A : \mathbf{o}a)$.

□

Obviously, we have $\mathbf{i}A \cup \mathbf{o}A = \mathbf{a}A$. If A has no internal ports then $\mathbf{i}A \cap \mathbf{o}A = \emptyset$. Ports may be directed as well. An *input port* is a port that consists of multiple input symbols and a single output symbol. Accordingly, an *output port* is a port that consists of a single input symbol and multiple output symbols. A port that consists of two singleton sets will be referred to as a *nonput port*; it serves for mere synchronization. If both symbol sets contain more than one symbol, the port permits bidirectional transfer of data during each handshake; it may be referred to as a *biput port*. Bidirectional data transfer will not recur in the sequel.

Definition 2.5 (port definition)

A port definition defines a port structure that consists of a single port; it may have one of the six forms below. Let a be a name and T be a type, i.e. a non-empty set of values.

0. a°, $a^\circ?T$ and $a^\circ!T$ define port structures of the form $\langle \{p\}, \emptyset \rangle$, where p is a single (passive) port. For the three port definitions p denotes $\langle \{a_0\}, \{a_1\} \rangle$, $\langle \{a_0\} \times T, \{a_1\} \rangle$, and $\langle \{a_0\}, \{a_1\} \times T \rangle$ respectively.

1. Likewise, $a^\bullet$, $a^\bullet?T$ and $a^\bullet!T$ define port structures of the form $\langle \emptyset, \{p\} \rangle$, where p is a single (active) port. For the three port definitions p denotes $\langle \{a_0\}, \{a_1\} \rangle$, $\langle \{a_0\}, \{a_1\} \times T \rangle$, and $\langle \{a_0\} \times T, \{a_1\} \rangle$ respectively.

2. The name in the port definition will be used as *port name*.

3. A list of port definitions separated by commas defines a port structure, provided that the port names are distinct. The port structure is obtained by taking the union of the port structures specified by the individual port definitions.

□

Note that a° and $a^\bullet$ define nonput ports, that $a^\circ?T$ and $a^\bullet?T$ define input ports and that $a^\circ!T$ and $a^\bullet!T$ define output ports.

Example 2.6

0. $a^\circ?Bool$ defines a port structure consisting of a single passive input port of type *Bool*, viz. the port $\langle \{a_0{:}false, a_0{:}true\}, \{a_1\} \rangle$. The input symbols are

$a_0 : \mathit{false}$ and $a_0 : \mathit{true}$ and the output symbol is a_1, the acknowledgement to an input.

1. $a^{\circ}!\mathit{Bool}$ defines a port structure consisting of a single passive output port of type *Bool*, viz. the port $\langle \{a_0\}, \{a_1 : \mathit{false}, a_1 : \mathit{true}\} \rangle$. The input symbol a_0 denotes a request for an output. The output symbols are $a_1 : \mathit{false}$ and $a_1 : \mathit{true}$.

□

The alphabet of port structure A, denoted by $\mathbf{a}A$, was defined as the set of symbols that occur in the port definitions of port structure A. The symbols of an alphabet are used to denote individual communications between a mechanism and its environment. Finite symbol sequences are called traces.

Definition 2.7 (trace)

0. A *trace* is a finite sequence of symbols.
1. The *empty trace* is denoted by ε.
2. $\mathit{len} \cdot t$ denotes the length of t.
3. The *concatenation* of traces s and t is obtained by juxtaposition, as in st.
4. Trace s is called a *prefix* of t, denoted by $s \leq t$, if there exists a trace u such that $su = t$.
5. The set of all traces over alphabet B is denoted by B^* .
6. The *projection* of trace t on alphabet B, denoted by $t\lceil B$, is defined by

$$\begin{array}{rcl} \varepsilon\lceil B & = & \varepsilon \\ (at)\lceil B & = & \textbf{if} \quad a \notin B \quad \rightarrow \quad t\lceil B \\ & & \;[]\quad a \in B \quad \rightarrow \quad a(t\lceil B) \\ & & \textbf{fi} \end{array}$$

If A is a port structure, $t\lceil A$ is used as a shorthand for $t\lceil(\mathbf{a}A)$.

□

Sets of traces will be used to characterize mechanisms. Each trace records a possible sequence of communication events in which a mechanism has engaged up to some moment in time. Prefix order is a partial order on traces. Hence, in

accordance with Section 2.1, the *prefix closure* of trace set B is denoted by $B^{\leq}$, and the *prefix closedness* of trace set B by $(\leq)\cdot B$.

Given a port structure A, we consider traces over alphabet $\mathbf{a}A$. A *handshake trace* is a trace in which the occurrences of **0**-symbols and **1**-symbols of each port strictly alternate, and in which the first symbol occurrence of each port is a **0**-symbol.

Definition 2.8 (handshake trace)

The set of handshake traces with port structure A is denoted by A^H and is defined as

$$\{t : t \in (\mathbf{a}A)^* \wedge (\forall a, s : a \in A \wedge s \leq t : 0 \leq len\cdot(s\lceil\mathbf{0}a) - len\cdot(s\lceil\mathbf{1}a) \leq 1) : t\}$$

□

Property 2.9

0. $\langle\emptyset,\emptyset\rangle^H = \{\varepsilon\}$
1. $(\leq)\cdot A^H$
2. $\langle A^\circ, A^\bullet\rangle^H = \langle A^\bullet, A^\circ\rangle^H$

□

Definition 2.10 (handshake structure)

0. A *handshake structure* S is a pair $\langle \mathbf{p}S, \mathbf{t}S\rangle$, in which $\mathbf{p}S$ is a port structure and $\mathbf{t}S$ a set of handshake traces, i.e. $\mathbf{t}S \subseteq (\mathbf{p}S)^H$.
1. The prefix closure is extended to handshake structures by
$$S^{\leq} = \langle \mathbf{p}S, (\mathbf{t}S)^{\leq}\rangle$$
2. Similarly, a handshake structure is $\leq$-closed if its trace set is.

□

Symbols **p** and **t** are used as operators on handshake structures. In the sequel R and S denote handshake structures. Also, the following shorthands are used:

0. $\mathbf{p}^{\circ}S = (\mathbf{p}S)^{\circ}$
1. $\mathbf{p}^{\bullet}S = (\mathbf{p}S)^{\bullet}$
2. $\mathbf{a}S = \mathbf{a}(\mathbf{p}S)$
3. $\mathbf{i}S = \mathbf{i}(\mathbf{p}S)$
4. $\mathbf{o}S = \mathbf{o}(\mathbf{p}S)$

2.3 Handshake processes

A handshake process will be defined as a handshake structure that satisfies five conditions. Handshake processes are used to represent the external behavior of a mechanism. Hence, handshake process $\langle A, T\rangle$ has no internal ports, i.e. $A^{\circ} \cap A^{\bullet} = \emptyset$ (condition 0). Furthermore, trace set T is required to be non-empty (condition 1).

In addition to the absence of internal ports and the presence of at least one handshake trace, three more conditions will be imposed, relating to progress, insensitivity to delays, and readiness to accept further inputs.

So far, we clearly distinguished physical objects such as mechanisms and events, from mathematical objects such as processes and symbols. Following Hoare [Hoa85] and others, this distinction will be adhered to less strictly. Whenever convenient, we use phrases such as "After process P has engaged in trace t it is ready to accept input a".

Quiescence

Let t be a trace of handshake process P. After engaging in t, the environment may be unable to obtain further output from P. Usually, this happens simply because the behavior of P does not permit P to extend t with any output symbol. Even if t can be extended with an output symbol, P may (nondeterministically) choose not to do so, and remain idle. In either case, t is called a *quiescent trace* [Mis84,Jon85]. Process P may leave quiescence after the environment has supplied further input.

A handshake process will be represented by its quiescent traces. The set of all (observable) traces is then the prefix closure of the set of quiescent traces. The quiescent-trace set of a handshake process must include the traces that have no output successors, i.e. it must include all its *passive traces*.

Definition 2.11 (passive traces)

Let S be a handshake structure ($\mathbf{p}S$ may have internal ports), and let t be a trace of $\mathbf{t}S^{\leq}$.

0. The *successor set* of t with respect to S, denoted by $suc\cdot(t, S)$, is

$$\{a : a \in \mathbf{a}S \ \wedge \ ta \in \mathbf{t}S^{\leq} : a\}$$

1. t *is passive in* S, denoted by $pas\cdot(t, S)$, when $suc\cdot(t, S) \cap \mathbf{o}S = \emptyset$.

2. S is *passive* when $pas\cdot(\varepsilon, S)$.

3. The *passive restriction of* S, denoted by $Pas\cdot S$, is the handshake structure

$$\langle \mathbf{p}S, \{t : t \in \mathbf{t}S^{\leq} \wedge pas\cdot(t, S) : t\}\rangle$$

□

The following property follows directly from the above definition.

Property 2.12

For handshake structure S we have $Pas\cdot S = Pas\cdot S^{\leq}$.

□

For handshake process P, $\mathbf{t}P$ represents the set of quiescent traces. Condition 2 in the definition of a handshake process is therefore $\mathbf{t}Pas\cdot S \subseteq \mathbf{t}S$.

An alternative way to look at the notion of quiescence is suggested by the following property.

Property 2.13

Let S be a handshake structure such that $\mathbf{t}Pas\cdot S \subseteq \mathbf{t}S$. Then

$$t \in \mathbf{t}S^{\leq} \ = \ (\exists u : u \in (\mathbf{o}S)^* : tu \in \mathbf{t}S)$$

Or put into words, for any trace t in $\mathbf{t}S^{\leq}$, there is a trace u consisting of output symbols only, such that tu is in $\mathbf{t}S$. Phrased differently, a handshake process can always become quiescent by producing outputs only. Since tu is a handshake trace, sequence u contains at most one symbol of each port.

Proof We derive:

$$
\begin{aligned}
& t \in \mathbf{t}S^{\leq} \\
= \quad & \{\ suc\cdot(t,S) \subseteq \mathbf{i}S \cup \mathbf{o}S\ \} \\
& t \in \mathbf{t}S^{\leq} \wedge (pas\cdot(t,S)\ \vee\ (\exists a : a \in \mathbf{o}S : ta \in \mathbf{t}S^{\leq})) \\
\Rightarrow \quad & \{\ \mathbf{t}Pas\cdot S \subseteq \mathbf{t}S;\ S \text{ is a handshake structure}\ \} \\
& t \in \mathbf{t}S\ \vee\ (\exists a : a \in \mathbf{o}S : ta \in \mathbf{t}S^{\leq}\ \wedge\ a \notin suc\cdot(ta,S))
\end{aligned}
$$

Repeating the last two steps for the remaining outputs in $suc\cdot(ta,S)$ completes the proof.

□

Reordering

Let P be a handshake process and let t be a an element of $\mathbf{t}P^{\leq}$. Assume that, after engaging in t, the environment sends signals along links a and b to P, in that order, and that P is ready to receive them, that is, tab is also in $\mathbf{t}P^{\leq}$. When no assumptions are made about the delays involved, a and b may arrive in the opposite order. Under such circumstances it is reasonable to require that tba is also $\mathbf{t}P^{\leq}$. Trace tba is said to *reorder* trace tab [Mis84,JHJ89]. A similar reordering must be allowed for two output symbols of P: for outputs c and d, trace tdc reorders tcd.

A slightly more subtle reordering is relevant when two symbols of opposite direction are involved, say input a and output c. Suppose tca is an element of $\mathbf{t}P^{\leq}$. Apparently the input a was not required by P in order to output c. When a had experienced less delay it would have arrived before the output of c. Hence, trace tac reorders tca, *provided* that both tac and tca are handshake traces. The converse, tca reorders tac, does not hold, because input a may be a prerequisite for output c. A formalization of reordering of handshake traces is given by means of a binary relation $\mathbf{r}_B$, where B is a port structure (B may have internal ports).

Definition 2.14 **(reorders)**

$\mathbf{r}_B$ is the smallest binary relation on B^H with for all symbols $a, b \in (\mathbf{i}B \setminus \mathbf{o}B)$, all symbols $c, d \in (\mathbf{o}B \setminus \mathbf{i}B)$, and all symbols $e \in (\mathbf{i}B \cap \mathbf{o}B)$:

0. $ab\ \mathbf{r}_B\ ba$

1. $cd\ \mathbf{r}_B\ dc$
2. $ac\ \mathbf{r}_B\ ca$
3. $ae\ \mathbf{r}_B\ ea$
4. $ec\ \mathbf{r}_B\ ce$

and for all handshake traces $r, s, t, u \in B^H$:

5. $t\ \mathbf{r}_B\ t$
6. $r\ \mathbf{r}_B\ s\ \wedge\ s\ \mathbf{r}_B\ t\ \Rightarrow\ r\ \mathbf{r}_B\ t$
7. $r\ \mathbf{r}_B\ s\ \wedge\ t\ \mathbf{r}_B\ u\ \Rightarrow\ rt\ \mathbf{r}_B\ su$

provided that rt and su are handshake traces as well.

□

This definition is based on the reorder relation "$\sqsubseteq$" of [JHJ89], with two differences. Firstly, $\mathbf{r}$ is restricted to handshake traces. Secondly, the relation is extended to alphabets with common input and output symbols: Properties 3 and 4 are derived from 0 to 2 by requiring e to assume both the role of input and that of output. When $\mathbf{i}B \cap \mathbf{o}B = \emptyset$, relation $\mathbf{r}$ reduces to $\sqsubseteq$ cited above, albeit restricted to handshake traces. Note that $s\ \mathbf{r}\ t\ \Rightarrow len \cdot s = len \cdot t$.

$\mathbf{r}_B$ will usually be shortened to $\mathbf{r}$ when B is obvious from the context. Since $\mathbf{r}$ is a preorder on B^H, $T^{\mathbf{r}}$ denotes the *reorder closure* of handshake-trace set T and $(\mathbf{r}) \cdot T$ denotes the reorder closedness of T (cf. Section 2.1). Both operators are lifted to handshake structures in the obvious way.

Property 2.15

Let B and C be compatible port structures, let $s, t \in B^H$, and let S be handshake structure. Then

0. $(\mathbf{r}) \cdot B^H$
1. $s\ \mathbf{r}_{B \cup C}\ t\ \Rightarrow\ s\ \mathbf{r}_B\ t$
2. $s\ \mathbf{r}\ t\ \Rightarrow\ s\lceil C\ \mathbf{r}\ t\lceil C$
3. $(\mathbf{r}) \cdot S\ \Rightarrow\ (\mathbf{r}) \cdot (S\lceil C)$
4. $(\mathbf{r}) \cdot S \Rightarrow (\mathbf{r}) \cdot S^{\leq}$

□

Property 1 is a consequence of the judicious choice of the extension of $\mathbf{r}$. Property 2 follows from 2.14.7, and Property 3 is a corollary of 2.

Proof of 4. Let $t \in \mathbf{t}S^{\leq}$ and $u \in (\mathbf{p}S)^H$. We derive:

$$
\begin{array}{ll}
 & t \in \mathbf{t}S^{\leq} \wedge u\ \mathbf{r}\ t \\
= & \quad \{ \text{ definition of } \leq\text{-closure } \} \\
 & (\exists v : v \in (\mathbf{a}S)^* : tv \in \mathbf{t}S) \wedge u\ \mathbf{r}\ t \\
\Rightarrow & \quad \{ \text{ definition of } \mathbf{r} \text{ ; calculus } \} \\
 & (\exists v : v \in (\mathbf{a}S)^* : tv \in \mathbf{t}S \wedge uv\ \mathbf{r}\ tv) \\
\Rightarrow & \quad \{ (\mathbf{r}) \cdot S \} \\
 & (\exists v : v \in (\mathbf{a}S)^* : uv \in \mathbf{t}S) \\
= & \quad \{ \text{ definition of } \leq\text{-closure } \} \\
 & u \in \mathbf{t}S^{\leq}
\end{array}
$$

□

Example 2.16

Implication $(\leq) \cdot S \Rightarrow (\leq) \cdot S^{\mathbf{r}}$ does not hold in general, as shown by the following S. $\mathbf{p}S$ consists of two passive ports, viz. a° and b°. By definition, $\mathbf{i}S = \{a_0, b_0\}$ and $\mathbf{o}S = \{a_1, b_1\}$. The trace set of S is given by $\mathbf{t}S = \{a_0 b_0\}^{\leq}$.

Obviously, we have $(\leq) \cdot S$ and $a_0 b_0 \in \mathbf{t}S$. Also, with $u = b_0 a_0$, we have $u \in S^{\mathbf{r}}$. However, prefix b_0 of u is not an element of $S^{\mathbf{r}}$.

□

Condition 3 in the definition of a handshake process states that $\mathbf{t}P$ must be closed under reordering. By Property 2.15.6, $\mathbf{t}P^{\leq}$ is then reorder closed as well.

Receptiveness

A non-empty set of handshake traces that includes all its passive prefixes and that is closed under reordering is a good candidate for the definition of handshake processes. However, it turns out that certain operations on such handshake processes, including parallel composition, are complicated in their definitions and usage. A useful class of handshake processes with surprisingly simple properties is obtained by imposing an additional requirement.

Definition 2.17 (input extension)

Let B be a port structure, and let $t, tu \in B^H$. Handshake trace tu is an input extension of t in B, denoted by $tu\ \mathbf{x}_B\ t$, if $u \in (\mathbf{i}B \setminus \mathbf{o}B)^*$.

□

When B is obvious from the context, $\mathbf{x}_B$ will be shortened to $\mathbf{x}$. Also $\mathbf{x}$ is a clearly a preorder. Hence, $T^{\mathbf{x}}$ denotes the input-extension closure of handshake-trace set T, and $(\mathbf{x})\cdot T$ denotes input-extension closedness. Both operators are lifted to handshake structures in the obvious way. A handshake structure whose *prefix closure* is closed under input extension is called *receptive*. The notion of receptiveness is similar to that of [Dil89,JHJ89,Jos90], albeit restricted to handshake traces.

Property 2.18

Let B and C be compatible port structures, let $s, t \in B^H$, and let S be handshake structure. Then:

0. $(\mathbf{x})\cdot B^H$
1. $s\ \mathbf{x}_{B\cup C}\ t \ \Rightarrow\ s\ \mathbf{x}_B\ t$
2. $s\ \mathbf{x}\ t \ \Rightarrow\ s\lceil C\ \mathbf{x}\ t\lceil C$
3. $(\mathbf{x})\cdot S \ \Rightarrow\ (\mathbf{x})\cdot (S\lceil C)$
4. $(\mathbf{x})\cdot S \Rightarrow (\mathbf{x})\cdot S^{\mathbf{r}}$
5. $(\leq)\cdot S \Rightarrow (\leq)\cdot S^{\mathbf{x}}$
6. $(\mathbf{x})\cdot S^{\leq} \Rightarrow (\mathbf{x})\cdot (Pas\cdot S)^{\leq}$

□

Condition 4 in the definition of a handshake process states that trace set $\mathbf{t}P$ must be receptive. As a result, the *only* obligation to be met by the environment is that it must adhere to the handshake protocol.

Handshake processes

By collecting the conditions stated so far, we obtain the complete definition of handshake processes:

Definition 2.19 (handshake process)

A handshake process is a handshake structure $\langle A, T \rangle$ that satisfies the following conditions:

0. $A^\circ \cap A^\bullet = \emptyset$ (no internal ports)
1. $T \neq \emptyset$ (non-empty trace set)
2. $\mathbf{t}(Pas \cdot \langle A, T \rangle) \subseteq T$ (quiescence for passive traces)
3. $(\mathbf{r}) \cdot T$ (reorder closed)
4. $(\mathbf{x}) \cdot T^{<}$ (receptive)

$\prod \cdot A$ denotes the set of all handshake processes with port alphabet A.

□

Like CSP, handshake-process theory has no notion of fairness. Unlike CSP, there is not a notion of divergence. Consequently, the various causes for quiescence cannot be distinguished. In the sequel P and Q (possibly subscripted) denote handshake processes. Unless stated otherwise, the word process is used as a shorthand for handshake process.

For a port structure A the following generic processes are defined.

Definition 2.20

0. $CHAOS \cdot A$ is the least predictable handshake process. It can engage in a handshake through any port at any time, and it can become quiescent at any time:
$$CHAOS \cdot A = \langle A, A^H \rangle$$
1. $STOP \cdot A$ never engages in a handshake communication through any of its ports. Nevertheless, it does not refuse any input through a passive port; it simply does not respond to such an input:
$$STOP \cdot A = \langle A, \{\varepsilon\}^{\mathbf{x}} \rangle$$
2. $RUN \cdot A$ is always willing to engage in a handshake through any of its ports:
$$RUN \cdot A = Pas \cdot CHAOS \cdot A$$

□

Note that $CHAOS \cdot \langle \emptyset, \emptyset \rangle = RUN \cdot \langle \emptyset, \emptyset \rangle = STOP \cdot \langle \emptyset, \emptyset \rangle = \langle \langle \emptyset, \emptyset \rangle, \{\varepsilon\} \rangle$. There is only one process with the empty port structure. More examples of handshake processes are presented next.

Example 2.21

0. P_0 is prepared to engage once in a handshake through a°:

$$\langle a^\circ, \{\varepsilon, a_0 a_1, a_0 a_1 a_0\} \rangle$$

1. P_1 is prepared to engage once in a handshake through $a^\bullet$:

$$\langle a^\bullet, \{a_0, a_0 a_1\} \rangle$$

2. P_2 behaves like P_0: it participates in a handshake through a°, but it refuses to acknowledge an input through b°:

$$\langle a^\circ \cup b^\circ, \{\varepsilon\} \cup \{a_0 a_1\}^{\mathbf{x}\,\mathbf{r}} \rangle$$

which equals

$$\begin{aligned} \langle a^\circ \cup b^\circ, \; & \{\varepsilon, b_0 \\ & , a_0 a_1, a_0 a_1 b_0, a_0 b_0 a_1, b_0 a_0 a_1 \\ & , a_0 a_1 a_0, a_0 a_1 a_0 b_0, a_0 a_1 b_0 a_0, a_0 b_0 a_1 a_0, b_0 a_0 a_1 a_0\} \rangle \end{aligned}$$

Even for such a simple behavior as P_2 the number of traces becomes considerable (11 quiescent traces!).

State graphs

A clarifying representation of handshake processes of modest complexity is the *state graph* accompanied by a port structure. A state graph is a directed graph in which arcs are labeled by symbols of the alphabet of the process. The nodes of a state graph are partitioned into a set of *quiescent* nodes and a set of *transient* nodes.

A non-empty subset of the nodes contains the so-called start nodes; often there is exactly one start node. A path from a start node corresponds to the trace that is obtained by listing the labels of the consecutive arcs in the path. The empty path corresponds to the empty trace. A path ending in a quiescent node corresponds to a quiescent trace. Different paths corresponding to the same trace

must either all end in quiescent nodes or all end in transient nodes. A state graph is said to represent a handshake structure if the set of traces corresponding to the paths that end in a quiescent node equals the set of quiescent traces (provided that the accompanied port structures match as well).

The following conventions are used when drawing state graphs:

0. Quiescent nodes are depicted by open circles, and transient nodes by filled ones.

1. A start node is enclosed by a concentric circle.

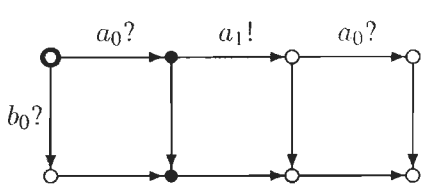
2. To avoid clutter, a node is occasionally depicted more than once. These multiple occurrences are labeled with a number unique to that node.

3. For clarity's sake, a question mark (exclamation mark) is attached to labels denoting input (output) symbols.

4. In some regularly drawn state graphs, the labeling of the arcs is incomplete. Arcs forming two opposite sides of a rectangle are then assumed to have the same label.

Example 2.22

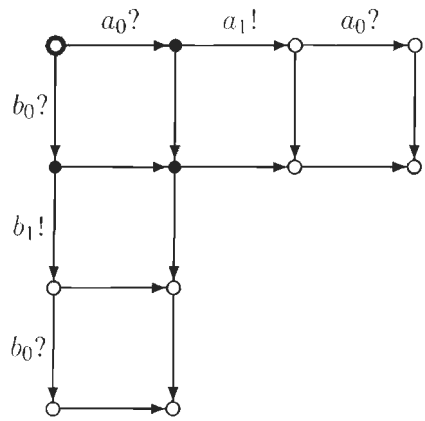
0. Process P_2 of the previous example is depicted by the following state graph:



1. Process P_3 is prepared to engage once in a handshake through either a° or b°:

$$\langle a^\circ \cup b^\circ, \{\varepsilon\} \cup \{a_0a_1, b_0b_1\}^{\mathbf{x}\,\mathbf{r}} \rangle$$

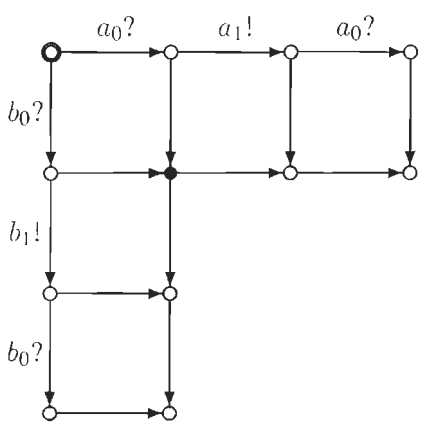
P_3 contains 19 quiescent traces as depicted by:



2. Process P_4 is like P_3, except that the choice between the two handshakes is made by the process itself:

$$\langle a^\circ \cup b^\circ, \{\varepsilon, a_0, b_0\} \cup \{a_0a_1, b_0b_1\}^{\mathbf{x\,r}} \rangle$$

P_4 contains 21 quiescent traces as depicted by:



Trace a_0 is a quiescent trace. If P_4 choses internally for a handshake through b°, no progress is made after trace a_0. Similarly, P_4 may refuse to complete a handshake through a°.

[ ]

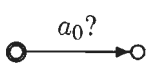
The handshake processes below occur in handshake circuits obtained by compilation of Tangram programs. Each handshake process is identified by a name

postfixed by a list of names enclosed by parentheses. From the names in the list ports are constructed by means of port definitions. The $^\circ$ and $^\bullet$ postfixes hint at the activity of these ports. Substitution of a name by another name yields an equivalent process, modulo renaming of the symbols. Alternatively, the process descriptions may be regarded as function definitions with lists of names as domain and processes as codomain.

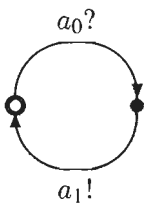
Example 2.23

The handshake processes below specify the handshake components required for the translation of the undirected subset of Tangram. Two examples, viz. *NMIX* and *NVAR* describe non-receptive handshake structures. The directed handshake components are presented in example 4.37. The behavior of most components is represented by a state graph. The graphic symbol introduced for each component will be used in handshake-circuit diagrams in later chapters. Let a, b and c be distinct names.

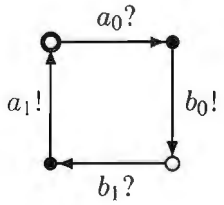
0. *STOP*$\cdot(a^\circ)$ has port structure a°. It does not respond to a request through port a°.



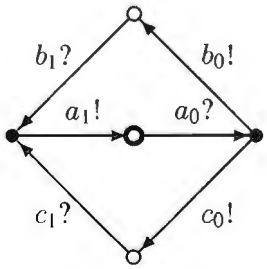
1. *RUN*$\cdot(a^\circ)$ has port structure a°. It acknowledges a request through a° and returns to its initial state.

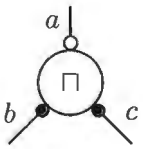


2. *CON*$\cdot(a^\circ, b^\bullet)$ is a *connector*. It has port structure $a^\circ \cup b^\bullet$ and each handshake through a° encloses a handshake through $b^\bullet$.

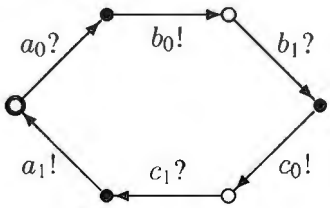


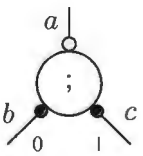
3. $OR\cdot(a^\circ, b^\bullet, c^\bullet)$ has port structure $a^\circ \cup b^\bullet \cup c^\bullet$. Each handshake through a° encloses either a handshake through $b^\bullet$ or one through $c^\bullet$.





4. $SEQ\cdot(a^\circ, b^\bullet, c^\bullet)$ is a *sequencer*. It has port structure $a^\circ \cup b^\bullet \cup c^\bullet$. A handshake through a° encloses both a handshake through $b^\bullet$ and one through $c^\bullet$, in that order.

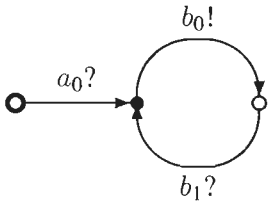




The numbers at the ports b and c specify the order of handshakes. When omitted, the order is counterclockwise, starting from the passive port.

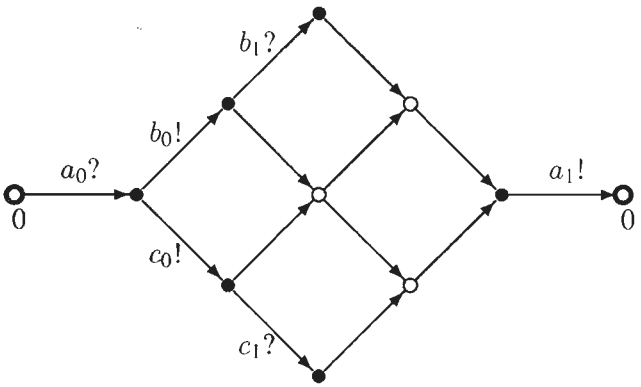
5. $DUP\cdot(a^\circ, b^\bullet)$ is a *duplicator*. It has port structure $a^\circ \cup b^\bullet$. A handshake through a° encloses two handshakes through $b^\bullet$. The state graph can be obtained from the state graph of the sequencer by renaming events c_0 and c_1 to b_0 and b_1 respectively.

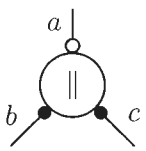
6. $REP\cdot(a^\circ, b^\bullet)$ is a *repeater*. It has port structure $a^\circ \cup b^\bullet$. A handshake through a° encloses an infinite repetition of handshakes through $b^\bullet$.



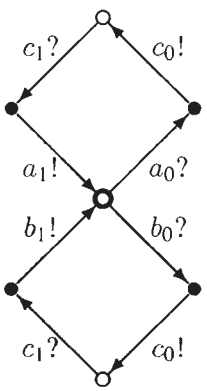
Note that the handshake through port a is never completed.

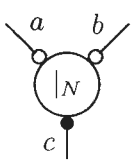
7. *PAR*$\cdot(a^\circ, b^\bullet, c^\bullet)$ has port structure $a^\circ \cup b^\bullet \cup c^\bullet$. A handshake through a° encloses both a handshake through $b^\bullet$ and one through $c^\bullet$, in parallel.





8. *NMIX*$\cdot(a^\circ, b^\circ, c^\bullet)$ has port structure $a^\circ \cup b^\circ \cup c^\bullet$. A handshake through $c^\bullet$ is enclosed by a handshake through either a° or b°.

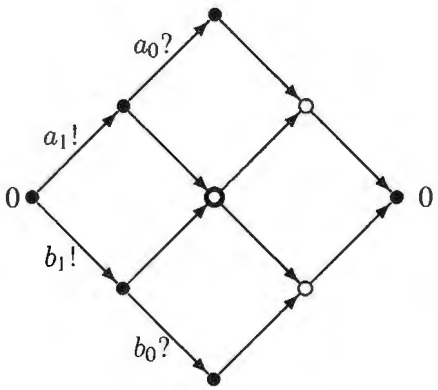


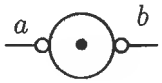


Note that *NMIX*$\cdot(a^\circ, b^\circ, c^\bullet)$ is *not* a handshake process, because it is not receptive: for instance, trace a_0 may not be extended with b_0. (*NMIX* stands for Non-receptive Mixer; cf. Section 7.2.)

9. $MIX\cdot(a^\circ, b^\circ, c^\bullet)$ has the same port structure as $NMIX\cdot(a^\circ, b^\circ, c^\bullet)$ and is defined as $(Pas\cdot NMIX\cdot(a^\circ, b^\circ, c^\bullet)^{\leq \mathbf{X}})^{\mathbf{r}}$. Process $MIX\cdot(a^\circ, b^\circ, c^\bullet)$ is receptive on account of Property 2.18.6. Hence, $MIX\cdot(a^\circ, b^\circ, c^\bullet)$ is a handshake process. In environments where handshakes through a° and b° never overlap, $MIX\cdot(a^\circ, b^\circ, c^\bullet)$ may be replaced by a $NMIX\cdot(a^\circ, b^\circ, c^\bullet)$ (cf. Section 7.2).

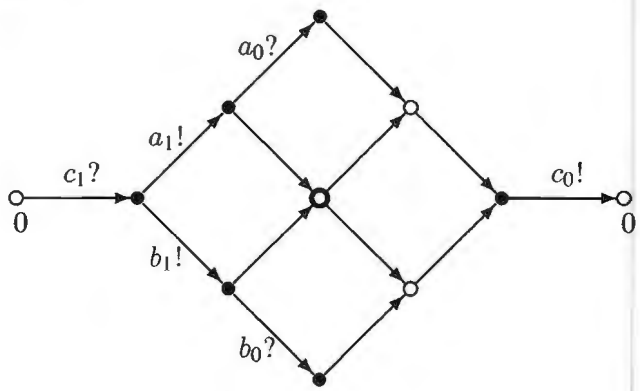
10. $PAS\cdot(a^\circ, b^\circ)$ has two passive ports, viz. a° and b°. It synchronizes each handshake through a° with a handshake through b°.

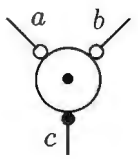




In [Kal86] $PAS\cdot(a^\circ, b^\circ)$ was introduced as a *passivator*.

11. $JOIN\cdot(a^\circ, b^\circ, c^\bullet)$ resembles $PAS\cdot(a^\circ, b^\circ)$, but has an additional active port $c^\bullet$. A handshake through $c^\bullet$ is enclosed by handshakes through both a° and b°.

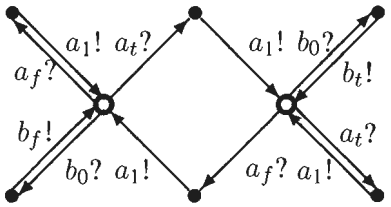


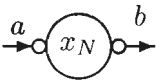


Note that $JOIN\cdot(a^\circ, b^\circ, c^\bullet)$ and $PAR\cdot(a^\circ, b^\bullet, c^\bullet)$ only differ in their arc labelings and in their start nodes.

12. $NVAR_{Bool}\cdot(a^\circ, b^\circ)$ is a Boolean variable with write port $a^\circ?Bool$ and read

port $b^\circ!Bool$. In the state graph below, the symbol names $a_0{:}false$, $a_0{:}true$, $b_1{:}false$ and $b_1{:}true$ are abbreviated by a_f, a_t, b_f and b_t respectively.





(x is used here as an instance name of the *NVAR* component; the type of the ports is assumed to be clear from the context.) Note that the state diagram has two initial states. The environment may start with a read request. *NVAR* then chooses nondeterministically between the values *false* and *true*.

For the same reason as *NMIX*, *NVAR* is not a handshake component: it is not receptive. The receptive counterpart of *NVAR* is obtained by taking input-extension closure of *VAR* followed by the reorder closure and the passive restriction:

$$VAR_{Bool}\cdot(a^\circ, b^\circ) \;=\; (Pas\cdot NVAR_{Bool}\cdot(a^\circ, b^\circ)^{\leq \mathbf{x}})^{\mathbf{r}}$$

Process *VAR* tolerates a write request during read handshake and vice versa.

□

Note that all handshake processes of Example 2.23 are passive.

Pas and *after*

The passive restriction of P, denoted $Pas\cdot P$, has been defined as a handshake structure with port structure $\mathbf{p}P$; the trace set of $Pas\cdot P$ contains those prefixes of $\mathbf{t}P$ that have input successors only (see Definition 2.11). *Pas* is clearly idempotent, and the passive restriction of a handshake process cannot have an empty trace set. Furthermore, it preserves receptiveness (Property 2.18.6). However, *Pas* does not always preserve reorder closedness. Hence, $Pas\cdot P$ is in general *not* a handshake process.

Example 2.24

With reference to Example 2.22:

0. $Pas \cdot P_2 = P_2$
1. $Pas \cdot P_4 = P_3$
2. all handshake processes of Example 2.23 are fixpoints of Pas

□

Next we define the behavior of a handshake process after trace t has been observed.

Definition 2.25 (after)

Let t be a handshake trace. The handshake structure $after \cdot (t, S)$ is defined by:

$$after \cdot (t, S) = \langle \mathbf{p}S, \{u : tu \in \mathbf{t}S : u\} \rangle$$

□

Clearly, $P = after \cdot (\varepsilon, P)$. In general, however, $after \cdot (t, P)$ is not a handshake process. For instance, a trace of $after \cdot (t, P)$ may start with an acknowledgement to a request that occurred in t. For *closed* traces this is not a problem. A handshake trace is closed if every handshake started has also been completed.

Definition 2.26 (closed trace)

Handshake trace t, $t \in A^H$ is *closed*, denoted by $closed \cdot t$ if $len \cdot (t\lceil \mathbf{0}a) = len \cdot (t\lceil \mathbf{1}a)$ for all ports $a \in A$.

□

Property 2.27

If $t \in \mathbf{t}P^{\leq}$ and $closed \cdot t$ then $after \cdot (t, P)$ is a handshake process.

□

Example 2.28

With reference to Examples 2.22 and 2.23:

0. $after \cdot (a_0 a_1, P_2) = STOP \cdot (a^\circ \cup b^\circ)$
1. $after \cdot (a_0 a_1, RUN \cdot a^\bullet) = RUN \cdot a^\bullet$ and
2. $after \cdot (a_0 b_0 b_1 a_1, CON \cdot (a^\circ, b^\bullet)) = CON \cdot (a^\circ, b^\bullet)$

□

2.4 The complete partial order $(\prod \cdot A, \sqsubseteq)$

In this section we analyze the structure of $\prod \cdot A$, i.e. the set of all processes with port structure A. All processes in this section have port structure A. For convenience, $CHAOS \cdot A$ will be abbreviated to *CHAOS*. This section follows the lead of [BHR84,Hoa85].

Refinement order

First we introduce an order relation $\sqsubseteq$ among handshake structures.

Definition 2.29 (refinement)

Let S and T be handshake structures. S refines to T, denoted by $S \sqsubseteq T$, if $\mathbf{t}S \supseteq \mathbf{t}T$.

□

Let P and Q be handshake processes. Paraphrasing Hoare ([Hoa85], page 132) we may say that $P \sqsubseteq Q$ now means that Q is equal to P or better in that it is less likely to become quiescent. Q is more predictable than P, because if Q can do something undesirable, P can do it too; and if Q can become quiescent, so can P. *CHAOS* can do anything at any time, and can become quiescent at any time. True to its name, it is the least predictable and controllable of all handshake processes; or in short the worst. Refinement $\sqsubseteq$ is clearly a partial order on $\prod \cdot A$, with *CHAOS* as least element.

Expression $P \sqsubseteq Q$ can also be read as "P specifies Q", "Q satisfies P", or "Q implements P". One of the main reasons to choose a nondeterministic specification, is to allow the implementor to select the cheapest or otherwise most

attractive implementation that satisfies the specification. Conversely, nondeterminism in specifications permits the specificator to abstract from many of the implementation details.

Example 2.30

0. $P \sqsubseteq Pas \cdot P$.

And with reference to Example 2.22:

1. $P_4 \sqsubseteq P_2$, and $P_4 \sqsubseteq P_3$.

2. $P_3 \not\sqsubseteq P_2$, and $P_2 \not\sqsubseteq P_3$. Neither P_2, nor P_3 can be refined any further.

□

Process P is maximal under $\sqsubseteq$ if any proper subset of tP violates the conditions of the definition of handshake process. With the aid of a state graph, this maximality can be easily checked. In terms of state graphs, there are two ways to reduce the trace set of the process it represents. One way is to turn a quiescent node into a transient node. This is allowed only when at least one output arc leaves that node (for instance the node reachable by trace a_0 in P_4). So, in the state graph of a maximal process all outputs must leave from a transient node. The other way to reduce the trace set is to remove one or more arcs. However, elimination of an input arc generally results in a process that violates receptiveness. Elimination of an output arc is allowed only if its source node is quiescent, or if it shares its transient source node with another output arc. (Arc b_1 leaving the node reached by trace b_0 in process P_4 may be removed.) Of course, the removal of one or more arcs must also leave the process reorder closed.

The behavior of an assembly in which process P is one of the components may be described as a function F from processes (with port structure $\mathbf{p}P$) to processes. If P can be refined into Q, it is only reasonable to require that $F \cdot Q$ is at least as good as $F \cdot P$. In other words, F must then be *order preserving*.

Definition 2.31 (order preserving)

Function F from handshake structures to handshake structures is order preserving (alternatively called *monotone* or *isotone*), if it preserves refinement ordering, i.e. if

$$S \sqsubseteq T \;\Rightarrow\; F \cdot S \sqsubseteq F \cdot T$$

□

Property 2.32

0. Let $\preceq$ be a preorder on traces. Then the $\preceq$-closure on handshake structures is order preserving.
1. Corollary: prefix closure, reorder closure and input-extension closure are order preserving.

□

A property of handshake processes that proves useful in the implementation of handshake components and handshake circuits is the following.

Definition 2.33 (initial when closed)

A handshake process P is *initial-when-closed* if for all *closed* traces $t \in \mathbf{t}P^{\leq}$ we have

$$P \sqsubseteq \mathit{after} \cdot (t, P)$$

□

When a closed trace of such a process has been observed, we may assume that the process is in an initial state. All processes of Example 2.23 are initial-when-closed. None of the processes of Example 2.22 are.

The complete partial order

The greatest lower bound of a set of handshake processes is obtained by taking the union of the respective trace sets.

Definition 2.34 (union)

The union of handshake structures S and T, denoted by $S \sqcap T$, is defined as $\langle A, \mathbf{t}S \cup \mathbf{t}T \rangle$.

□

Property 2.35

If P and Q are handshake processes, then $P \sqcap Q$ is also a handshake process.

□

In Section 2.5 we shall interpret $P \sqcap Q$ as the nondeterministic composition of processes P and Q. The least upper bound of a set of handshake processes does

in general not exist in $\prod \cdot A$. If it exists, it is obtained by intersecting the trace sets.

Definition 2.36 (intersection)

The intersection of handshake structures S and T, denoted by $S \sqcup T$ is defined as $\langle A, \mathbf{t}S \cap \mathbf{t}T \rangle$.

□

For a chain of processes, it will be shown that the continued $\sqcup$ is a process.

Definition 2.37 (chain, limit)

0. An (ascending) *chain* is defined as an infinite sequence $(i : 0 \leq i : S_i)$ of handshake structures, such that $S_i \sqsubseteq S_{i+1}$.
1. For chain $(i : 0 \leq i : S_i)$ the *limit* is defined as the the continued intersection of the handshake structures in the chain, viz.

$$(\sqcup\, i : 0 \leq i : S_i)$$

which equals

$$\langle A, \{t : (\forall i : 0 \leq i : t \in \mathbf{t}S_i) : t\} \rangle$$

□

Definition 2.38 (continuous)

Let F be a function from handshake structures to handshake structures and let $(i : 0 \leq i : S_i)$ be a chain of handshake structures. F is (upward) continuous if it satisfies:

$$(\sqcup\, i : 0 \leq i : F \cdot S_i) \;=\; F \cdot (\sqcup\, i : 0 \leq i : S_i)$$

□

Continuity of a function from handshake structures to any CPO is defined similarly.

Property 2.39

0. The containment of the passive restriction is continuous:

$$\begin{array}{ll} & (\sqcup\, i : 0 \leq i : \mathbf{t}Pas \cdot S_i \subseteq \mathbf{t}S_i) \\ = & (\mathbf{t}Pas \cdot (\sqcup\, i : 0 \leq i : S_i) \subseteq (\sqcup\, i : 0 \leq i : S_i)) \end{array}$$

1. Let $\preceq$ be a preorder on traces. Then the $\preceq$-closedness is continuous.
2. Corollary: prefix closedness, reorder closedness and input-extension closedness are continuous.

Proof of 1 (i ranges over the natural numbers). We derive:

$$
\begin{array}{ll}
 & t \in (\sqcup i :: S_i) \\
= & \quad \{ \text{ definition of limit } \} \\
 & (\forall i :: t \in S_i) \\
= & \quad \{ \ S_i \text{ are closed under } \preceq \ \} \\
 & (\forall i :: (\forall u : u \preceq t : u \in S_i)) \\
= & \quad \{ \text{ calculus } \} \\
 & (\forall u : u \preceq t : (\forall i :: u \in S_i)) \\
= & \quad \{ \text{ definition of limit } \} \\
 & (\forall u : u \preceq t : u \in (\sqcup i :: S_i))
\end{array}
$$

□

Example 2.40

The $\preceq$-closure is *not* continuous in general. Consider for instance the chain $(i : 0 \leq i : S_i)$, with $S_i = \langle a^\circ, \{j : i \leq j : (a_0 a_1)^j\}\rangle$. Then $(\sqcup i :: S_i^{\leq}) = \langle a^\circ, (a^\circ)^H\rangle$ and $(\sqcup i :: S_i)^{\leq} = \langle a^\circ, \emptyset\rangle$.

□

The following property claims continuity for two specific preorders on traces.

Property 2.41

0. **r**-closure is continuous.
1. **x**-closure is continuous.

Proof of 0. Let $(i : 0 \leq i : S_i)$ be a chain and let t be a trace of $(\sqcup i :: S_i^{\mathbf{r}})$. Furthermore, let U_i be the set $\{u : u \in S_i \wedge t \ \mathbf{r} \ u : u\}$. Note that U_i depends on t. Clearly, $(i : 0 \leq i : U_i)$ is also a chain. We derive:

$t \in (\sqcup i :: S_i^{\mathbf{r}})$

= { definitions of **r**-closure and of limit }

$(\forall i :: (\exists u : t\ \mathbf{r}\ u : u \in S_i))$

= { definition of U_i }

$(\forall i :: (\exists u :: u \in U_i))$

= { $(i : 0 \leq i : U_i)$ is chain; $\mid U_i \mid$ is finite }

$(\exists j : j \geq 0 : (\forall i : i \geq j : U_i = U_j))$

= { $(i : 0 \leq i : U_i)$ is chain }

$(\exists u :: (\forall i : i \geq 0 : u \in S_i))$

= { definition of U_i }

$(\exists u : t\ \mathbf{r}\ u : (\forall i :: u \in S_i))$

= { definitions of limit and of **r**-closure }

$t \in (\sqcup i :: S_i)^{\mathbf{r}}$

□

Theorem 2.42

The limit of a chain of processes is a process.

Proof Follows from the continuity of prefix closedness, reorder closedness, input-extension closedness and containment of the passive restriction.

□

Definition 2.43 (complete partial order, CPO)

Partial order $\langle Z, \preceq \rangle$ is complete if

0. Z contains a least element, and

1. every chain in Z has a limit.

□

Corollary 2.44

Partial order $(\prod \cdot A, \sqsubseteq)$ is a complete partial order with *CHAOS*$\cdot A$ as least element and $(\sqcup\, i : 0 \leq i : P_i)$ as limit of chains $(i : 0 \leq i : P_i)$.

□

Property 2.45

For chain $(i : 0 \leq i : S_i)$ we have for all j: $P_j \sqsubseteq (\sqcup\, i : 0 \leq i : S_i)$.

□

One reason why we took all the trouble to prove that $(\prod \cdot A, \sqsubseteq)$ is a CPO, is that the least fixpoint for equations of the form $P = F \cdot P$, with F a continuous function, can be constructed straightforwardly within a CPO. This allows the definition of handshake processes by means of recursion. Recursive process definitions will be discussed in Chapter 4.

2.5 Nondeterminism

In contrast with CSP, the maximal elements of the partial order $\sqsubseteq$ are not necessarily deterministic. Nondeterministic behavior may exhibit itself in two forms:

0. a process may have the choice of doing an output or becoming quiescent;
1. a process may choose between two outputs, where the choice for one of the outputs disables the other.

This is formalized below.

Definition 2.46 (deterministic handshake process)

Handshake process $\langle A, T \rangle$ is deterministic if for all distinct output symbols a and b:

0. $ta \in T^{\leq} \;\Rightarrow\; t \notin T$
1. $ta \in T^{\leq} \wedge tb \in T^{\leq} \;\Rightarrow\; tab \in T^{\leq}$

□

Example 2.47

All processes presented so far are deterministic, except P_3 and P_4 in Example 2.22 and $OR{\cdot}(a^\circ, b^\bullet, c^\bullet)$, $MIX{\cdot}(a^\circ, b^\circ, c^\bullet)$, $NVAR_{Bool}(a^\circ, b^\circ)$ and $VAR_{Bool}(a^\circ, b^\circ)$ in Example 2.23, and, of course, $CHAOS{\cdot}\ A$. $RUN{\cdot}\ A$ is deterministic only if A consists exclusively of nonput ports and input ports. Note that $NMIX{\cdot}\ (a^\circ, b^\circ, c^\bullet)$ is deterministic, albeit not receptive. $NVAR_{Bool}{\cdot}\ (a^\circ, b^\circ)$ is nondeterministic, because after a trace b_0 the variable may reply with either $b_0 : false$ or $b_0 : true$.

□

An interesting and useful classification of *non*-deterministic asynchronous processes has been suggested by Tom Verhoeff [Ver89], on which we base the following definition.

Definition 2.48 (static and dynamic nondeterminism)

Nondeterministic process P is *statically* nondeterministic if it can be refined into a deterministic process, and *dynamically* nondeterministic otherwise.

□

Example 2.49

With reference to Examples 2.22 and 2.23:

0. P_4 and $OR{\cdot}\ (a^\circ, b^\bullet, c^\bullet)$ are statically nondeterministic; the former can be refined into (for instance!) P_2 and the latter to $CON{\cdot}\ (a^\circ, b^\bullet)$ with $c^\bullet$ added to its port structure.

1. P_3 and $MIX{\cdot}\ (a^\circ, b^\circ, c^\bullet)$ are dynamically nondeterministic, in spite of their maximality under $\sqsubseteq$.

2. Note that the statically nondeterministic P_4 can also be refined into the dynamically nondeterministic P_3 .

□

Dynamically nondeterministic processes require arbiters for their implementation.

Nondeterministic composition

Process $P \sqcap Q$ ("P or Q") behaves exactly like P *or* like Q. The choice between P and Q is nondeterministic.

Property 2.50

0. Nondeterministic composition is idempotent, order preserving, commutative, associative, distributive and continuous.
1. $P \sqcap Q \sqsubseteq P$.

□

If a process is specified by $P \sqcap Q$, the implementor is free to select either P or Q as implementation. For order preserving F we obviously have

$$F \cdot (P \sqcap Q) \sqsubseteq F \cdot P \sqcap F \cdot Q$$

A stronger property of functions on handshake processes is [BHR84] is *distributivity*:

Definition 2.51 (distributivity)

0. A function F from handshake processes to handshake processes is *distributive* if
$$F \cdot (P \sqcap Q) = F \cdot P \sqcap F \cdot Q$$
1. A function of two or more arguments is called distributive if it is distributive in each argument separately.

□

Distributive functions are clearly order preserving. Nondeterministic composition is distributive, since

$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R)$$

Example 2.52

0. Let $\preceq$ be a preorder on traces. Then the $\preceq$-closure on handshake structures is distributive.
1. Corollary: prefix closure, reorder closure and input-extension closure are distributive.

□

Chapter 3

Handshake circuits

3.0 Introduction

The most interesting operation on handshake processes is parallel composition. Parallel composition is defined only for *connectable* processes. Connectability of handshake processes captures the idea that ports form the unit of connection (as opposed to individual port symbols), and that a passive port can only be connected to a single active port and vice versa. A precise definition will be given later.

The communication between connectable handshake processes is asynchronous: the sending of a signal by one process and the reception of that signal by another process are two distinct events. Asynchronous communication is more complicated than synchronous communication, because of the possible occurrence of *interference*. The concept of interference with respect to voltage transitions has been mentioned in Section 0.1. Interference with respect to symbols occurs when one process sends a symbol and the other process is not ready to receive it. The receptiveness of handshake processes and the imposed handshake protocol exclude the possibility of interference, thus yielding a relatively simple definition for parallel composition.

Another complication is, however, the possibility of *divergence*: an unbounded amount of internal communication, which cannot be distinguished externally from deadlock. From an implementation viewpoint divergence is undesirable: it forms a drain on the power source, without being productive.

The external behavior of the parallel composition of connectable P and Q will be denoted by $P \mid\mid Q$, which is again a handshake process. Both internal and external behavior of the parallel composition of two processes will be analyzed

in detail in Section 3.1.

Section 3.2 introduces handshake circuits. A handshake circuit is a finite set of pairwise connectable handshake processes. The external behavior of a handshake circuit is again a handshake process, and is uniquely defined by $||$, due to the associativity and commutativity of $||$.

Handshake processes form a special class of so-called *delay-insensitive* processes. Delay-insensitive processes and their parallel (de-)compositions have been studied extensively (references will be given later). Some facts about handshake processes and their compositions are stated in terms of the existing theory on delay-insensitivity in Appendix A, including:

- handshake processes are delay-insensitive;
- ports of handshake processes are *independent*;
- handshake circuits are free of interference.

3.1 Parallel composition

First we must agree on how to specify connectivity between two handshake processes, say P and Q. A convenient way to specify connectivity is by identity of ports, that is, port a of P is connected to port b of Q if a and b consist of the same sets of symbols. In order to exclude various forms of "partial connections", we require that ports of P and Q are either identical, or have disjoint symbol sets: the port structures of P and Q must be compatible (cf. Definition 2.3). Furthermore, we exclude connections between two passive ports or two active ports, because this would imply the connection of outputs to outputs. In short, P and Q must be *connectable*. This notion is defined together with the *reflection* of a port structure next.

Definition 3.0 (connectability, reflection)

0. Port structures A and B are *connectable*, denoted by $A \bowtie B$, if
   (a) A and B are compatible, and
   (b) $A^\circ \cap B^\circ = \emptyset$ and $A^\bullet \cap B^\bullet = \emptyset$
1. Two handshake structures are connectable if their respective port structures are.

2. The reflection of port structures A, denoted by $\overline{A}$, is defined by:

$$\overline{\langle A^\circ, A^\bullet \rangle} = \langle A^\bullet, A^\circ \rangle$$

□

Connectability and reflection enjoy the following obvious properties.

Property 3.1

0. $A \bowtie \emptyset$
1. $A \bowtie B \;=\; B \bowtie A$
2. $\mathbf{a}A \cap \mathbf{a}B = \emptyset \;\Rightarrow\; A \bowtie B$
3. $A \bowtie \overline{A}$
4. $A = \overline{\overline{A}}$

□

Example 3.2

0. $a^\circ \;\bowtie\; a^\bullet$
1. $a^\circ?\mathbf{bool} \;\bowtie\; a^\bullet!\mathbf{bool}$
2. $P \;\bowtie\; CHAOS \cdot \overline{\mathbf{p}P}$
3. $SEQ \cdot (a^\circ, b^\bullet, c^\bullet) \;\bowtie\; MIX \cdot (b^\circ, c^\circ, d^\bullet)$
4. $MIX \cdot (a^\circ, b^\circ, c^\bullet) \;\bowtie\; SEQ \cdot (c^\circ, d^\bullet, e^\bullet)$

□

In the sequel P and Q are connectable handshake processes. Now we are ready to analyze the interaction between P and Q. Let C be the set of internal ports, viz. $(\mathbf{p}^\circ P \cap \mathbf{p}^\bullet Q) \cup (\mathbf{p}^\bullet P \cap \mathbf{p}^\circ Q)$, let port $p \in C$, and let a be an element of $\mathbf{a}p$, such that $a \in (\mathbf{i}P \cap \mathbf{o}Q)$. Furthermore, let $t \in \mathbf{t}P^{\leq}$ and $u \in \mathbf{t}Q^{\leq}$, and let this pair of traces specify the current state.

In general, $t\lceil C \neq u\lceil C$, because symbols sent by one process need not have arrived at the other process yet. Even if all sent symbols have arrived, t and u may differ due to reordering. Assume that event a may occur next as an output

of Q, i.e. trace $ua \in \mathbf{t}Q^{\leq}$. By outputting a, process Q either starts a handshake (if $a \in \mathbf{0}p$), or acknowledges a handshake (if $a \in \mathbf{1}p$). Because at most one symbol can be on its way in a channel, we may conclude that $u\lceil p = t\lceil p$. The question now is: is P ready to input a? I.e., is ta in $\mathbf{t}P^{\leq}$?

The following simple reasoning shows that this is so. Since Q is a handshake process, trace ua must be a handshake trace, and therefore $ua\lceil p$ and $ta\lceil p$ are also handshake traces. With t a handshake trace, ta must be a handshake trace as well. Because P is receptive, it must be prepared to extend t with a. Hence, $ta \in \mathbf{t}P^{\leq}$: the arrival of a at P does not cause interference. Similar reasoning holds for $a \in (\mathbf{o}P \cap \mathbf{i}Q)$, because of symmetry. Absence of interference for sets of processes is defined in Appendix A.

The interaction through common ports restricts the behavior of P and Q; not all traces of P can occur in the presence of Q and vice versa. Let the prefix-closed trace sets P' and Q' denote the respective restricted behaviors. The above reasoning suggests that $P'\lceil C = Q'\lceil C$. It is because of this that the *weave* [vdS85] of P and Q is useful in the definition of $P \parallel Q$.

Definition 3.3 (weave)

For connectable handshake structures R and S, we define the *weave* of R and S, denoted by $R \mathbf{w} S$ as

$$\langle A, \{t : t \in A^H \wedge t\lceil \mathbf{p}R \in \mathbf{t}R \wedge t\lceil \mathbf{p}S \in \mathbf{t}S : t\}\rangle$$

where $A = \mathbf{p}R \cup \mathbf{p}S$.

□

The following properties will be used. Cf. [vdS85] for many of the proofs.

Property 3.4

Let R, S and T be three mutually connectable handshake structures.

0. $R \mathbf{w} CHAOS \cdot \langle \emptyset, \emptyset \rangle = R$
1. $R \mathbf{w} S = S \mathbf{w} R$
2. $(R \mathbf{w} S) \mathbf{w} T = R \mathbf{w} (S \mathbf{w} T)$

□

Some properties of handshake structures are preserved with weaving.

Property 3.5

Let R and S be connectable handshake structures.

0. $(\leq)\cdot R \wedge (\leq)\cdot S \;\Rightarrow\; (\leq)\cdot (R \mathbf{w} S)$
1. $(\mathbf{r})\cdot R \wedge (\mathbf{r})\cdot S \;\Rightarrow\; (\mathbf{r})\cdot (R \mathbf{w} S)$
2. $(\mathbf{x})\cdot R \wedge (\mathbf{x})\cdot S \;\Rightarrow\; (\mathbf{x})\cdot (R \mathbf{w} S)$

Proof

0. cf. Property 1.17 in [vdS85].
1. Let $\mathbf{r}_R$, $\mathbf{r}_S$ and $\mathbf{r}_{RS}$ denote $\mathbf{r}\ \mathbf{p}_R$, $\mathbf{r}\ \mathbf{p}_S$ and $\mathbf{r}\ \mathbf{p}_{R \cup \mathbf{p}S}$ respectively. We derive:

$$
\begin{array}{ll}
 & t \in \mathbf{t}(R \mathbf{w} S) \wedge s\ \mathbf{r}_{RS}\ t \\
= & \quad \{\ \text{Definition 3.3 (weaving)}\ \} \\
 & t\lceil \mathbf{a}R \in \mathbf{t}R \wedge t\lceil \mathbf{a}S \in \mathbf{t}S \wedge s\ \mathbf{r}_{RS}\ t \\
\Rightarrow & \quad \{\ \text{Property 2.15.2 (twice)}\ \} \\
 & t\lceil \mathbf{a}R \in \mathbf{t}R \wedge s\lceil \mathbf{a}R\ \mathbf{r}_{RS}\ t\lceil \mathbf{a}R \wedge t\lceil \mathbf{a}S \in \mathbf{t}S \wedge s\lceil \mathbf{a}S\ \mathbf{r}_{RS}\ t\lceil \mathbf{a}S \\
\Rightarrow & \quad \{\ \text{Property 2.15.1 (twice)}\ \} \\
 & t\lceil \mathbf{a}R \in \mathbf{t}R \wedge s\lceil \mathbf{a}R\ \mathbf{r}_R\ t\lceil \mathbf{a}R \wedge t\lceil \mathbf{a}S \in \mathbf{t}S \wedge s\lceil \mathbf{a}S\ \mathbf{r}_S\ t\lceil \mathbf{a}S \\
\Rightarrow & \quad \{\ R \text{ and } S \text{ are reorder closed}\ \} \\
 & s\lceil \mathbf{a}R \in \mathbf{t}R \wedge s\lceil \mathbf{a}S \in \mathbf{t}S \\
= & \quad \{\ \text{definition of weaving}\ \} \\
 & s \in \mathbf{t}(R \mathbf{w} S)
\end{array}
$$

2. Similar to 1.

□

Property 3.6

Let R and S be connectable handshake structures.

0. $\mathbf{t}(R \mathbf{w} S)^{\leq} \subseteq \mathbf{t}(R^{\leq} \mathbf{w} S^{\leq})$

1. $\mathbf{t}Pas\cdot(R\ \mathbf{w}\ S) \subseteq \mathbf{t}(Pas\cdot R\ \mathbf{w}\ Pas\cdot S)$

Proof

0. cf. Property 1.18 in [vdS85].

1. Similar to Property 3.5.1.

□

The following property of weaving will be used as a lemma in Theorem 3.12.

Property 3.7

Let P and Q be handshake processes. Then

$$\begin{array}{l} t \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq}) \\ \Rightarrow\ \ t \in \mathbf{t}(P\ \mathbf{w}\ Q)^{\leq} \ \vee\ (\exists u : u \in (\mathbf{o}P \cup \mathbf{o}Q)^* \ \wedge\ tu \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq}) : u \neq \varepsilon) \end{array}$$

Proof Let $t \in (\mathbf{p}P \cup \mathbf{p}Q)^H$. We derive:

$$\begin{array}{ll} & t \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq}) \\ = & \quad \{\ \text{definition of weaving}\ \} \\ & t\lceil \mathbf{a}P \in \mathbf{t}P^{\leq} \ \wedge\ t\lceil \mathbf{a}Q \in \mathbf{t}Q^{\leq} \\ \Rightarrow & \quad \{\ \text{Property 2.13 (twice)}\ \} \\ & (\exists v, w : v \in (\mathbf{o}P)^* \wedge w \in (\mathbf{o}Q)^* : (t\lceil \mathbf{p}P)v \in \mathbf{t}P \ \wedge\ (t\lceil \mathbf{p}Q)w \in \mathbf{t}Q) \\ \Rightarrow & \quad \{\ P \text{ and } Q \text{ are receptive; } Q \text{ is reorder closed}\ \} \\ & (\ \exists v, w : v \in (\mathbf{o}P)^* \wedge w \in (\mathbf{o}Q)^* \\ & :\ (tvw\lceil \mathbf{p}P \in \mathbf{t}P \ \wedge\ tvw\lceil \mathbf{p}Q \in \mathbf{t}Q) \\ & \quad \vee (tvw\lceil \mathbf{p}P \in \mathbf{t}P^{\leq} \ \wedge\ tvw\lceil \mathbf{p}Q \in \mathbf{t}Q^{\leq} \ \wedge\ (v \neq \varepsilon \ \vee\ w \neq \varepsilon)) \\ &) \\ \Rightarrow & \quad \{\ u = vw\text{; definition of weaving (twice); calculus; trading}\ \} \\ & t \in \mathbf{t}(P\ \mathbf{w}\ Q)^{\leq} \ \vee\ (\exists u : u \in (\mathbf{o}P \cup \mathbf{o}Q)^* \ \wedge\ tu \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq}) : u \neq \varepsilon) \end{array}$$

□

So far we have ignored quiescence in analyzing the parallel composition of two processes, by looking at the prefix closures of the trace sets only. Fortunately, the weave also captures quiescence, because a parallel composition is

quiescent if and only if both components are. This makes the weave an attractive composition operator. However, the weave of two processes is *not* a handshake process, because of its internal ports. By concealing the internal ports and projection of the quiescent traces of the weave on the external ports, we obtain a handshake structure that represents the externally observable behavior of a parallel composition, in most cases. This form of parallel composition is known as *blending* [vdS85].

Definition 3.8 (blending; external port structure)

0. The blend of handshake processes P and Q, denoted by P **b** Q, i s defined as

$$(P \mathbf{w} Q)\lceil \mathbf{e}(\mathbf{p}P \cup \mathbf{p}Q)$$

where $\mathbf{e}(\mathbf{p}P \cup \mathbf{p}Q$ is the external port structure, defined next.

1. The *external port structure* of port structure A, denoted by **e** A , is the port structure $A \setminus \overline{A}$, which is equivalent to $\langle A^\circ \setminus A^\bullet, A^\bullet \setminus A^\circ \rangle$ (cf. Definition 2.3).

□

Unfortunately, the blend of two handshake processes is not a handshake process in general, as shown by the following example.

Example 3.9

Consider the parallel composition of $REP \cdot (a^\circ, b^\bullet)$ and $RUN \cdot b^\circ$. The former includes the traces $\varepsilon, a_0 b_0$ and $a_0 b_0 b_1 b_0$, the latter includes $\varepsilon, b_0 b_1$ and $b_0 b_1 b_0 b_1$.

The handshake structure $REP \cdot (a^\circ, b^\bullet)$ **w** $RUN \cdot b^\circ$ contains exactly one trace, viz. ε. No other trace is quiescent: after trace a_0 processes $REP \cdot (a^\circ, b^\bullet)$ and $RUN \cdot b^\circ$ "play ping pong" indefinitely. Concealment of b has of course no effect on this trace set. However, the resulting blend $\langle a^\circ, \{\varepsilon\} \rangle$ is *not* a handshake process, because it is not receptive. Trace a_0 does occur, and is quiescent as far as the environment is concerned. If we ignore handshakes along b, we apparently must accept $\langle a^\circ, \{\varepsilon, a_0\} \rangle$ as the behavior of $REP \cdot (a^\circ, b^\bullet) \,||\, RUN \cdot b^\circ$.

□

The occurrence of an unbounded sequence of internal events is known as *infinite chatter* (cf. [vdS85] page 52) or *infinite overtaking* (cf. [Hoa85] page 80). The traces that lead to such a bothersome state of affairs are called *divergences*.

Definition 3.10 (divergences)

For handshake structure $\langle A, T\rangle$ we define the *divergences* of $\langle A, T\rangle$, denoted by $div\cdot\langle A, T\rangle$, as the trace set

$$\{t : (\forall n : 0 \leq n : (\exists u : u \in (A^\circ \cap A^\bullet)^* \wedge tu \in T : n < len\cdot u)) : t\}$$

□

Recall that $A^\circ \cap A^\bullet$ is the set of internal ports of A. Note that a handshake structure without internal ports cannot have divergences. The set of divergences of a reorder-closed handshake structure is also closed under reordering, as we shall prove next.

Property 3.11

0. Let R be a prefix closed handshake structure. Then $div\cdot R \subseteq \mathbf{t}R$.
1. Let R be a reorder-closed handshake structure. Then $div\cdot R$ is also reorder closed.

Proof

0. Follows immediately from the definition of *div*.
1. Let $R = \langle A, T\rangle$. We derive:

$$\begin{array}{ll} & t \in \mathbf{t}div\cdot\langle A, T\rangle \wedge s\ \mathbf{r}\ t \\ = & \quad \{\ \text{Definition 3.10 (divergences)}\ \} \\ & (\forall n : 0 \leq n : (\exists u : u \in X^* \wedge tu \in T : n < len\cdot u)) \wedge s\ \mathbf{r}\ t \\ = & \quad \{\ \text{calculus; Definition 2.14.7}\ \} \\ & (\forall n : 0 \leq n : (\exists u : u \in X^* \wedge tu \in T \wedge su\ \mathbf{r}\ tu : n < len\cdot u)) \\ = & \quad \{\ T \text{ is reorder closed}\ \} \\ & (\forall n : 0 \leq n : (\exists u : u \in X^* \wedge su \in T : n < len\cdot u)) \\ = & \quad \{\ \text{definition of } div\ \} \\ & s \in \mathbf{t}div\cdot\langle A, T\rangle \end{array}$$

□

Theorem 3.12

Let P and Q be handshake processes. Then

$$\mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq}) \;=\; (\mathbf{t}(P\ \mathbf{w}\ Q) \cup \mathit{div}(P^{\leq}\ \mathbf{w}\ Q^{\leq}))^{\leq}$$

Proof by mutual set inclusion.

Case $(\mathbf{t}(P\ \mathbf{w}\ Q) \cup \mathit{div}(P^{\leq}\ \mathbf{w}\ Q^{\leq}))^{\leq} \;\subseteq\; \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq})$.

$t \in (\mathbf{t}(P\ \mathbf{w}\ Q) \cup \mathit{div}\cdot (P^{\leq}\ \mathbf{w}\ Q^{\leq}))^{\leq}$

$\Rightarrow$ { prefix closure distributes over $\cup$; Property 3.11.0 }

$t \in (\mathbf{t}(P\ \mathbf{w}\ Q)^{\leq} \cup \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq})^{\leq})$

$=$ { $\mathbf{t}S \subseteq \mathbf{t}S^{\leq}$; weaving is monotonic }

$t \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq})^{\leq}$

$=$ { Property 3.5.0 }

$t \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq})$

Case $\mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq}) \;\subseteq\; (\mathbf{t}(P\ \mathbf{w}\ Q) \cup \mathit{div}(P^{\leq}\ \mathbf{w}\ Q^{\leq}))^{\leq}$.

Let trace $t \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq})$ and let predicate X_n be defined as

$$t \in \mathbf{t}(P\ \mathbf{w}\ Q)^{\leq} \;\vee\; (\exists u : u \in (\mathbf{o}P \cup \mathbf{o}Q)^{*} \;\wedge\; tu \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq}) : n \leq \mathit{len}\cdot u)$$

Predicate X_0 holds trivially. Using Property 3.7 it follows that $(\forall n : 0 \leq n : X_n \Rightarrow X_{n+1})$ Hence, by induction, we have $(\forall n : 0 \leq n : X_n)$ Equivalently:

$$t \in \mathbf{t}(P\ \mathbf{w}\ Q)^{\leq}$$
$$\vee\ (\forall n : 0 \leq n : (\exists u : u \in (\mathbf{o}P \cup \mathbf{o}Q)^{*} \;\wedge\; tu \in \mathbf{t}(P^{\leq}\ \mathbf{w}\ Q^{\leq}) : n \leq \mathit{len}\cdot u))$$

The second term brings us very close to the definition of *div* (cf. Definition 3.10). However, u ranges over all outputs and not exclusively over internal symbols. Fortunately, the number of external outputs in u is finite, because of handshaking and the finite number of ports involved. Hence, as far as the second term is considered, t is a prefix of a divergence of $P^{\leq}\ \mathbf{w}\ Q^{\leq}$. Q.e.d.

□

Corollary 3.13

Let P and Q be connectable handshake processes such that $div \cdot (P^{\leq} \mathbf{w} Q^{\leq}) = \emptyset$.

0. $(P \mathbf{w} Q)^{\leq} = P^{\leq} \mathbf{w} Q^{\leq}$
1. $(P \mathbf{b} Q)^{\leq} = P^{\leq} \mathbf{b} Q^{\leq}$

□

Having dealt with interference, quiescence, concealment, and divergence, we have done all the groundwork needed for the definition of $P \,||\, Q$.

Definition 3.14 (parallel composition)

The parallel composition of connectable handshake processes P and Q is denoted by $P \,||\, Q$ and defined as

$$\langle A, (\mathbf{t}(P \mathbf{w} Q) \cup div \cdot (P^{\leq} \mathbf{w} Q^{\leq}))\lceil A \rangle$$

with $A = \mathbf{e}(\mathbf{p}P \cup \mathbf{p}Q)$.

□

As a corollary to Theorem 3.12 we may conclude:

Property 3.15

$(P \,||\, Q)^{\leq} = P^{\leq} \mathbf{b} Q^{\leq}$

□

Theorem 3.16

The parallel composition of connectable handshake processes P and Q, as denoted by $P \,||\, Q$, is a handshake process.

Proof Since $P \,||\, Q$ is clearly a handshake structure, it remains to prove that $P \,||\, Q$ satisfies the five conditions of Definition 2.19. The proof is structured accordingly.

0. According to Definition 3.14 we have $\mathbf{p}(P \,||\, Q) = \mathbf{e}(\mathbf{p}P \cup \mathbf{p}Q)$. From Definition 3.8.1 it can directly be seen that $\mathbf{e}(\mathbf{p}P \cup \mathbf{p}Q)$ has no internal ports.

1. We derive:

$$
\begin{array}{ll}
 & \mathit{true} \\
\Rightarrow & \quad \{\ \mathbf{t}P \text{ and } \mathbf{t}Q \text{ are non empty}\ \} \\
 & \varepsilon \in \mathbf{t}P^{\leq} \ \wedge\ \varepsilon \in \mathbf{t}Q^{\leq} \\
\Rightarrow & \quad \{\ \text{definition of blend}\ \} \\
 & \varepsilon \in \mathbf{t}(P^{\leq}\ \mathbf{b}\ P^{\leq}) \\
= & \quad \{\ \text{Property 3.15}\ \} \\
 & \varepsilon \in \mathbf{t}(P \parallel Q)^{\leq} \\
\Rightarrow & \quad \{\ \text{calculus}\ \} \\
 & \mathbf{t}(P \parallel Q) \neq \emptyset
\end{array}
$$

2. We derive:

$$
\begin{array}{ll}
 & \mathbf{t}\mathit{Pas}\cdot(P \parallel Q) \\
= & \quad \{\ \text{Property 2.12}\ \} \\
 & \mathbf{t}\mathit{Pas}\cdot(P \parallel Q)^{\leq} \\
= & \quad \{\ \text{Property 3.15}\ \} \\
 & \mathbf{t}\mathit{Pas}\cdot(P^{\leq}\ \mathbf{b}\ Q^{\leq}) \\
\subseteq & \quad \{\ \text{Property 3.6.1}\ \} \\
 & \mathbf{t}(\mathit{Pas}\cdot P^{\leq}\ \mathbf{b}\ \mathit{Pas}\cdot Q^{\leq}) \\
= & \quad \{\ \text{Property 2.12 (twice)}\ \} \\
 & \mathbf{t}(\mathit{Pas}\cdot P\ \ \mathbf{b}\ \ \mathit{Pas}\cdot Q) \\
\subseteq & \quad \{\ P \text{ and } Q \text{ are quiescent for passive traces; property of } \mathbf{b}\ \} \\
 & \mathbf{t}(P\ \mathbf{b}\ Q) \\
\subseteq & \quad \{\ \text{property of } \parallel\ \} \\
 & \mathbf{t}(P \parallel Q)
\end{array}
$$

Hence, $P \parallel Q$ is quiescent for passive traces.

3. We derive:

$$
\begin{array}{ll}
 & \mathit{true} \\
\Rightarrow & \quad \{\ P \text{ and } Q \text{ are reorder closed}\ \} \\
 & (\mathbf{r})\cdot P \ \wedge\ (\mathbf{r})\cdot Q \\
\Rightarrow & \quad \{\ \text{Properties 2.2.1 (twice) and 3.5.1 (twice)}\ \} \\
 & (\mathbf{r})\cdot (P\ \mathbf{w}\ Q) \ \wedge\ (\mathbf{r})\cdot (P^{\leq}\ \mathbf{w}\ Q^{\leq}) \\
\Rightarrow & \quad \{\ \text{Property 3.11.1}\ \} \\
 & (\mathbf{r})\cdot (P\ \mathbf{w}\ Q) \ \wedge\ (\mathbf{r})\cdot \mathit{div}\cdot (P^{\leq}\ \mathbf{w}\ Q^{\leq}) \\
\Rightarrow & \quad \{\ \text{Property 2.2.1}\ \} \\
 & (\mathbf{r})\cdot ((P\ \mathbf{w}\ Q \cup \mathit{div}\cdot (P^{\leq}\ \mathbf{w}\ Q^{\leq}))\lceil \mathbf{e}(\mathbf{A}P \cup \mathbf{A}Q)) \\
= & \quad \{\ \text{definition of } P \parallel Q\ \} \\
 & (\mathbf{r})\cdot (P \parallel Q)
\end{array}
$$

4. We derive:

$$
\begin{array}{ll}
 & \mathit{true} \\
\Rightarrow & \quad \{\ P \text{ and } Q \text{ are receptive}\ \} \\
 & (\mathbf{x})\cdot P^{\leq} \ \wedge\ (\mathbf{x})\cdot Q^{\leq} \\
\Rightarrow & \quad \{\ \text{Property 3.5.2}\ \} \\
 & (\mathbf{x})\cdot (P^{\leq}\ \mathbf{w}\ Q^{\leq}) \\
\Rightarrow & \quad \{\ \text{Property 2.18.3}\ \} \\
 & (\mathbf{x})\cdot (P^{\leq}\ \mathbf{b}\ Q^{\leq}) \\
= & \quad \{\ \text{Property 3.15}\ \} \\
 & (\mathbf{x})\cdot (P \parallel Q)^{\leq}
\end{array}
$$

□

Property 3.17

0. Parallel composition is commutative, associative, distributive and continuous.

1. $CHAOS \cdot \langle \emptyset, \emptyset \rangle \parallel P = P.$

□

In the remainder of this thesis, examples involving parallel composition are free of infinite overtaking. Parallel composition then reduces to the conceptually simpler blending.

The way two processes are connected can be pictured by means of a *connection diagram*. These diagrams are also a convenient means to display the connectivity pattern of handshake circuits (see e.g. the circuits of Section 1.2). In a connection diagram, processes are drawn as circles with their ports drawn as small circles attached to their periphery. Passive ports are represented by open circles, active ports by filled ones. A channel is represented by a line connecting *exactly* one passive port to one active port of two *distinct* processes. The direction of a channel is represented by an arrow indicating the direction of data transport (when applicable).

Example 3.18

The parallel compositions below refer to handshake processes of Example 2.23.

0. The parallel composition of two connectors connected "tail to head" is again a connector:

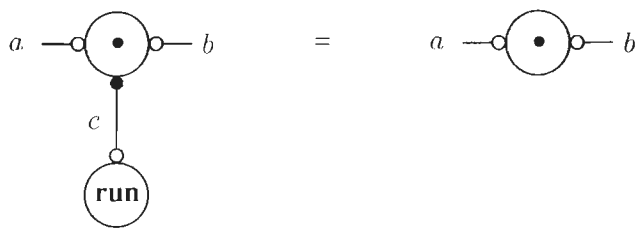
$$CON \cdot (a^\circ, b^\bullet) \parallel CON \cdot (b^\circ, c^\bullet) = CON \cdot (a^\circ, c^\bullet)$$

1. Connecting a connector to a process has the effect of renaming the port it connects to:

$$REP \cdot (a^\circ, b^\bullet) \parallel CON \cdot (b^\circ, c^\bullet) = REP \cdot (a^\circ, c^\bullet)$$

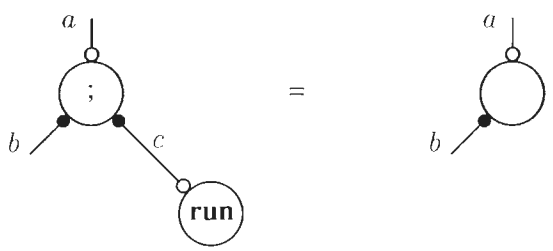
2. A port of a handshake process can effectively be concealed by connecting it to a *RUN* component:

$$JOIN \cdot (a^\circ, b^\circ, c^\bullet) \,||\, RUN \cdot (c^\circ) = PAS \cdot (a^\circ, b^\circ)$$



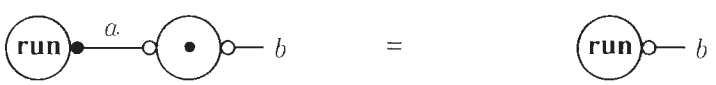
Also:

$$SEQ \cdot (a^\circ, b^\bullet, c^\bullet) \,|_{\cdot}\, RUN \cdot (c^\circ) = CON \cdot (a^\circ, b^\bullet)$$



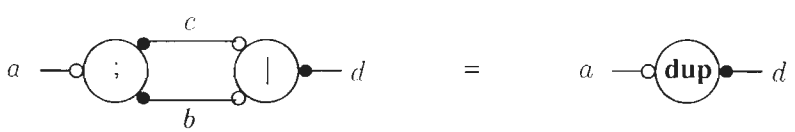
3. An active port can be turned into a passive port by connecting it to a passivator:

$$RUN \cdot a^\bullet \,||\, PAS \cdot (a^\circ, b^\circ) = RUN \cdot b^\circ$$



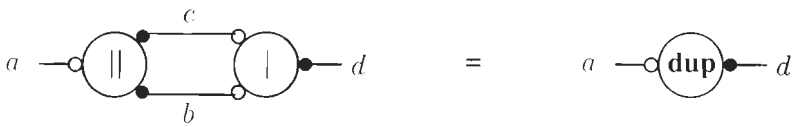
4. A duplicator can be constructed from a sequencer and a mixer:

$$SEQ \cdot (a^\circ, b^\bullet, c^\bullet) \,||\, MIX \cdot (b^\circ, c^\circ, d^\bullet) = DUP \cdot (a^\circ, d^\bullet)$$



A duplicator can also be constructed from a *PAR* component and a mixer:

$$PAR \cdot (a^\circ, b^\bullet, c^\bullet) \,||\, MIX \cdot (b^\circ, c^\circ, d^\bullet) = DUP \cdot (a^\circ, d^\bullet)$$

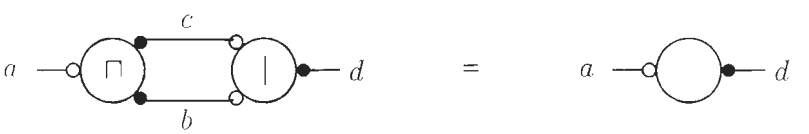


It is interesting to compare the respective weaves. Trace $a_0b_0c_0d_0$ is a trace of $PAR \cdot (a^\circ, b^\bullet, c^\bullet)$ **w** $MIX \cdot (b^\circ, c^\circ, d^\bullet)$ but not of $SEQ \cdot (a^\circ, b^\bullet, c^\bullet)$ **w** $MIX \cdot (b^\circ, c^\circ, d^\bullet)$. For trace $a_0b_0d_0$ the converse is true: it is quiescent only in the sequencer based duplicator.

However, these differences in internal behavior are concealed to the external observer.

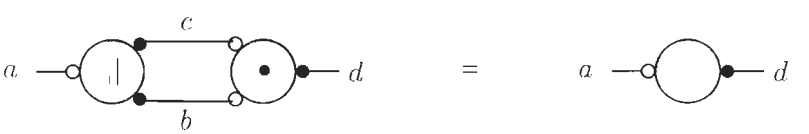
5. Nondeterminism is not preserved under parallel composition:

$$OR \cdot (a^\circ, b^\bullet, c^\bullet) \,||\, MIX \cdot (b^\circ, c^\circ, d^\bullet) = CON \cdot (a^\circ, d^\bullet)$$



6. Another realization of $CON \cdot (a^\circ, d^\bullet)$ is suggested by:

$$PAR \cdot (a^\circ, b^\bullet, c^\bullet) \,||\, JOIN \cdot (b^\circ, c^\circ, d^\bullet) = CON \cdot (a^\circ, d^\bullet)$$



In Chapter 6 we shall recognize examples 0, 1, 4 and 6 as instances of property 6.23. Each of the above examples can also be viewed as a substitution or rewrite rule: the composition at the left-hand side of an equality may be replaced by the component at the right-hand side. These substitutions therefore also suggest optimizations of handshake circuits (cf. Section 7.1).

□

We conclude this section with two properties of parallel composition that prove useful for the initialization of handshake circuits (Section 7.6).

Property 3.19

0. The property "being passive" is preserved under parallel composition, that is, $P \mid\mid Q$ is passive if P and Q are passive.

1. The property initial-when-closed is preserved under parallel composition.

□

3.2 Handshake circuits

Handshake circuits at last!

Definition 3.20 (handshake circuit)

0. A *handshake circuit* is a finite *connectable* set of handshake processes.

1. Let H be a finite set of handshake structures. H is *connectable*, denoted by $\underline{\bowtie} H$, if all handshake structures are pairwise connectable, that is:

$$\underline{\bowtie} H \quad = \quad (\forall S, T : S \in H \wedge T \in H \wedge S \neq T : S \bowtie T)$$

□

In particular, the empty set and the singleton set are handshake circuits. Note that the required connectability excludes "broadcast" among handshake processes: a port may occur in the port structures of at most two processes of a given handshake circuit. Consistent with the terminology of Chapter 1 we shall refer to the handshake processes of a handshake circuit as its *handshake components*.

Most of the operators of the previous section can be generalized to handshake circuits in a straightforward fashion.

Definition 3.21

0. The *external port structure* of handshake circuit H, denoted by $\mathbf{e}H$, is the port structure

$$\mathbf{e}(\cup P : P \in H : \mathbf{p}P)$$

1. $H^{\leq} = \{P : P \in H : P^{\leq}\}$

2. $\mathbf{W} \cdot H = (\mathbf{w}\ P : P \in H : P)$

3. $\mathbf{B}\cdot H = (\ \mathbf{b}\ \ P : P \in H : P)$

4. $||\cdot H = (||\ \ P : P \in H : P)$

□

The first definition relies on

$$A \bowtie B \ \wedge\ B \bowtie C \ \wedge\ C \bowtie A \ \Rightarrow\ \mathbf{e}(A \cup B) \bowtie C$$

The last three definitions rely on the associativity and commutativity of weaving, blending, and parallel composition, as well as on the existence of a null element $CHAOS\cdot \langle \emptyset, \emptyset \rangle$ for all three operators. Many of the properties of the previous section generalize similarly:

Corollary 3.22

For handshake circuit H we have:

0. $div\cdot(\mathbf{W}\cdot H^{\leq}) = \emptyset \ \Rightarrow\ (\mathbf{W}\cdot H)^{\leq} = \mathbf{W}\cdot(H^{\leq})$

1. $div\cdot(\mathbf{W}\cdot H^{\leq}) = \emptyset \ \Rightarrow\ ||\cdot H = \mathbf{B}\cdot H$

2. $\mathbf{e}H = \mathbf{p}(||\cdot H)$

3. $||\cdot H$ is a handshake process.

4. $(||\cdot H)^{\leq} = ||\cdot H^{\leq}$

□

The following properties relate to the set nature of a handshake circuit.

Property 3.23

0. $\emptyset$ is a handshake circuit. Since $STOP\cdot\langle\emptyset,\emptyset\rangle$ is the unit of parallel composition of handshake processes, we have $||\cdot\emptyset = STOP\cdot\langle\emptyset,\emptyset\rangle$.

1. Let H and I be handshake circuits, such that $\underline{\bowtie}(H \cup I)$. Then $H \cup I$ is a handshake circuit, and

$$\begin{aligned} \mathbf{e}(H \cup I) &= \mathbf{e}(\mathbf{e}H \cup \mathbf{e}I) \\ ||\cdot(H \cup I) &= (||\cdot H)\ \ ||\ \ (||\cdot I) \end{aligned}$$

□

As a corollary to Property 3.19, we may conclude that the behavior of a handshake circuit constructed from passive components is also passive. The same holds for the property initial-when-closed.

Example 3.24

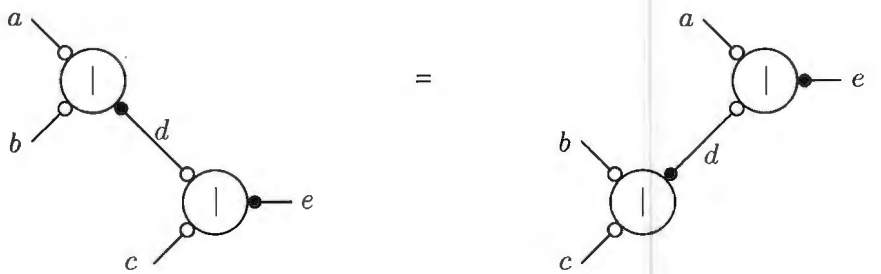
0. A three-way mixer is the natural generalization of the two-way mixer *MIX*. A three-way mixer with a°, b° and c° as passive ports and $d^\bullet$ as active port can be realized by

$$MIX\cdot(a^\circ, b^\circ, d^\bullet) \;||\; MIX\cdot(c^\circ, d^\circ, e^\bullet)$$

which is equivalent to

$$MIX\cdot(b^\circ, c^\circ, d^\bullet) \;||\; MIX\cdot(a^\circ, d^\circ, e^\bullet)$$

Pictorially this can be expressed as:



An N-way mixer ($N \geq 2$) can be realized as a tree of $N-1$ two-way mixers. Although all trees exhibit the same behavior (i.e. define the same N-way mixer process), response times to requests through different passive ports may differ considerably. The degenerated case, with all $N-1$ mixers linked into a list, is an extreme in this respect.

1. Similar to the N-way mixer, we may consider N-way generalizations of *SEQ*, *PAR* and *OR*. These three handshake circuits have one passive port and N active ports, and can be constructed from $N-1$ instances of their two-way counterparts.

2. $DUP_N\cdot(a^\circ, b^\bullet)$ is one way to generalize the duplicator, with $DUP_1\cdot(a^\circ, b^\bullet) = DUP\cdot(a^\circ, b^\bullet)$, and for $N > 1$:

$$DUP_N\cdot(a^\circ, b^\bullet) = DUP_{N-1}\cdot(a^\circ, b^\bullet_{N-1}) \;||\; DUP\cdot(b^\circ_{N-1}, b^\bullet)$$

For each handshake through a we may expect 2^N handshakes through port b. With $N = 90$ and a rate of one handshake through b per nanosecond, it takes approximately 10^{18} seconds to complete a single handshake through a°. This is about the estimated life time of the universe, and may present a slight problem for the testing of a VLSI circuit that implements this chain (see Section 7.7).

□

Chapter 4

Sequential handshake processes

4.0 Introduction

So far the quiescent trace set of a handshake process was specified in one of the following forms: by enumeration, by a predicate, by a state graph, or by parallel composition of other handshake processes.

For many handshake processes neither of the above forms may be convenient. An example of such a process is the process that first behaves like P and then, "after successful termination of P", behaves like Q. Of course, such *sequential composition* of the handshake processes P and Q requires a notion of successful termination of a process. A *sequential handshake process* is a handshake process in which that notion is incorporated.

The aim of this chapter is to develop a model for sequential handshake processes and a calculus for these processes. An important application of this calculus is the description of the handshake components required for the compilation of Tangram. Another application is the semantics of Tangram itself.

4.1 Sequential handshake processes

A sequential handshake process is a handshake process, of which a subset of the quiescent traces is designated as traces that lead to successful termination. In a sequential composition these so-called *terminal* traces can act as antecedents to traces of the subsequent sequential handshake process.

Let T denote the set of quiescent traces and let U denote the set of terminal traces of sequential handshake process P. Sets T and U must satisfy a number of conditions, which will be introduced informally first.

For terminal handshake traces u, we require that for any handshake trace v, trace uv is a handshake trace as well. Therefore, we require that all terminal traces are closed.

Traces that are both terminal and quiescent form a special class of traces. After such a trace, a non-deterministic choice is made whether to terminate successfully or not. Hence, $T \cap U$ does not need to be empty.

Proper input extensions of terminal trace u, and reorderings thereof form another special class of traces. Since they are not closed, they cannot be terminal. They are not necessarily quiescent either. We choose not to record these traces explicitely, but to require that $\langle A, T \cup U^{\mathbf{x}\,\mathbf{r}} \rangle$ is a handshake process.

Reordering should of course not have an effect on termination. Hence, both T and U must be closed under reordering. Combining the above leads to the following definition.

Definition 4.0 (sequential handshake process)

Let P be a triple $\langle A, T, U \rangle$, in which A is a port structure, and T and U are subsets of A^H. Furthermore, let V denote $T \cup U^{\mathbf{x}\,\mathbf{r}}$. Triple P is a *sequential handshake process* if the following conditions are satisfied (cf. Definition 2.19):

0. $A^\circ \cap A^\bullet = \emptyset$
1. $V \neq \emptyset$
2. $\mathbf{t}(Pas \cdot \langle A, V \rangle) \subseteq V$
3. $(\mathbf{r}) \cdot T \wedge (\mathbf{r}) \cdot U$
4. $(\mathbf{x}) \cdot V^{\leq}$ and
5. $(\forall u : u \in U : closed \cdot u)$

Trace set U is the set of *terminal* traces and T is the set of quiescent traces. Sequential handshake process P may be written as $\langle \mathbf{p}P, \mathbf{t}P, \mathbf{u}P \rangle$. The set of all sequential handshake processes with port structure A is denoted by $\sum \cdot A$.

□

Note that Conditions 0 to 4 closely mirror the corresponding conditions of the definition of handshake processes. For brevity's sake, the word handshake may

be omitted in "sequential handshake process". In the remainder of this section P and Q denote sequential handshake processes. The following property shows their relation to (non-sequential) handshake processes.

Property 4.1

0. If $\langle A, T, U \rangle$ is a sequential handshake process then $\langle A, T \cup U^{\mathbf{x}\,\mathbf{r}} \rangle$ is a handshake process.

1. Corollary: if $\langle A, T, \emptyset \rangle$ is a sequential handshake process then $\langle A, T \rangle$ is a handshake process.

□

These properties inspire the following definition.

Definition 4.2 (permanent sequential process)

0. A sequential handshake process is permanent if its set of terminal traces is empty.

1. A handshake process $\langle A, T \rangle$ is said to *correspond to* the permanent sequential process $\langle A, T, \emptyset \rangle$, and vice versa.

□

When P is a permanent sequential process, and no confusion can arise, we will sometimes use P as if it is a handshake process and omit the phrase "the handshake process corresponding to". In particular, permanent sequential processes will be used to define the behavior of handshake components and handshake circuits.

In [Hoa85] terminal traces are appended with a symbol $\surd$, indicating successful termination. A clear advantage of such an encoding is the absence of the need to introduce another process model. To some extent, this advantage is eroded when the extra rules that govern the use of $\surd$ have to be taken into account. Moreover, the recording of the terminal traces in a separate set will pay off in the definitions of the various operators on sequential processes.

For a port structure A the following generic sequential processes are defined.

Definition 4.3

0. *CHAOS*· A is the least predictable sequential handshake process. It can engage in a handshake through any port at any time, it can become quiescent at any time, and it may terminate successfully after any closed trace:

$$CHAOS \cdot A = \langle A, A^H, \{t : t \in A^H \wedge closed \cdot t : t\}\rangle$$

1. *RUN*· A is always willing to engage in a handshake through any of its ports. However, it never terminates:

$$RUN \cdot A = \langle A, \mathbf{t}Pas \cdot \langle A, A^H\rangle, \emptyset\rangle$$

2. *STOP*· A never engages in a handshake communication through any of its ports. Neither does it ever terminate successfully:

$$STOP \cdot A = \langle A, \{\varepsilon\}^{\mathbf{x}}, \emptyset\rangle$$

3. *SKIP*· A never engages in a handshake communication through any of its ports. All it does is terminate successfully:

$$SKIP \cdot A = \langle A, \emptyset, \{\varepsilon\}\rangle$$

□

CHAOS· A, *RUN*· A, and *STOP*· A have also been defined as handshake processes and have the same set of quiescent traces as their non-sequential counterparts (cf. Definition 2.20). In future reference to these processes the context will indicate which variant is intended. *RUN*· A and *STOP*· A are permanent sequential processes. Also note that:

- $CHAOS \cdot \langle\emptyset, \emptyset\rangle = \langle\langle\emptyset, \emptyset\rangle, \{\varepsilon\}, \{\varepsilon\}\rangle$
- $RUN \cdot \langle\emptyset, \emptyset\rangle = \langle\langle\emptyset, \emptyset\rangle, \{\varepsilon\}, \emptyset\rangle$
- $STOP \cdot \langle\emptyset, \emptyset\rangle = \langle\langle\emptyset, \emptyset\rangle, \{\varepsilon\}, \emptyset\rangle$
- $SKIP \cdot \langle\emptyset, \emptyset\rangle = \langle\langle\emptyset, \emptyset\rangle, \emptyset, \{\varepsilon\}\rangle$

Indeed, there are three sequential processes with the empty port structure.

The CPO of sequential handshake processes

The set of sequential handshake processes with port structure A, denoted by $\sum\cdot A$, can be analyzed in a way similar to our analysis of $\prod\cdot A$. The respective definitions, properties and theorems then bear close resemblance. In this subsection we rephrase the more significant results of Section 2.4 in terms of sequential processes. All sequential processes in this subsection have port structure A. First we introduce a partial order relation $\sqsubseteq$ among sequential processes.

Definition 4.4 (refinement)

Let P and Q be sequential processes with the same port structure. P refines to Q, denoted by $P \sqsubseteq Q$, if

$$\mathbf{t}P \supseteq \mathbf{t}Q \quad \text{and} \quad \mathbf{u}P \supseteq \mathbf{u}Q$$

□

Again, $P \sqsubseteq Q$ may be read as P refines to Q, P specifies Q, or Q implements P. The least element in this order is $CHAOS\cdot A$. A function from $\sum\cdot A$ to $\sum\cdot A$ is *order preserving* if it preserves refinement ordering.

Definition 4.5 (nondeterministic composition)

The nondeterministic composition of P and Q, denoted by $P \sqcap Q$, is defined as $\langle A, \mathbf{t}P \cup \mathbf{t}Q, \mathbf{u}P \cup \mathbf{u}Q\rangle$.

□

$P \sqcap Q$ is the greatest lower bound of P and Q in the partial order $\langle \sum\cdot A, \sqsubseteq\rangle$. Later $\sqcap$ will be generalized to sequential processes with unequal port structures. A function from $\sum\cdot A$ to $\sum\cdot A$ is *distributive* if it distributes over nondeterministic composition.

Definition 4.6 (intersection)

The intersection of P and Q, denoted by $P \sqcup Q$, is defined as

$$\langle A, \mathbf{t}P \cap \mathbf{t}Q, \mathbf{u}P \cap \mathbf{u}Q\rangle$$

□

The intersection of two sequential processes is generally not a sequential process.

However, with limit and chain defined as in Section 2.4 we arrive at the following, hardly surprising, theorem.

Theorem 4.7

Partial order $(\sum \cdot A, \sqsubseteq)$ is a CPO with $CHAOS \cdot A$ as least element and $(\sqcup\, i : 0 \leq i : P_i)$ as limit of chain $(i : 0 \leq i : P_i)$.

□

In accordance with Definition 2.38, function F from $\sum \cdot A$ to $\sum \cdot A$ is *continuous* if it commutes with the limit operation.

Recursion

One way to define a sequential handshake process is by recursion, for example as a fixpoint of a given function F, i.e. a solution of the equation $P = F \cdot P$. We conclude this section by instantiating the well-known fixpoint construction in CPO's [BHR84,DP90].

Theorem 4.8

Let F be a continuous function from $\sum \cdot A$ to $\sum \cdot A$, and let the n-fold composition of F be denoted by F^n. Then

$$(\sqcup\, n : 0 \leq n : F^n \cdot CHAOS)$$

is the *least fixpoint* of F.

Proof First we prove that the above limit is a fixpoint of F:

$$\begin{aligned}
& F \cdot (\sqcup n : 0 \leq n : F^n \cdot CHAOS) \\
= & \quad \{ \text{ Continuity of } F \ \} \\
& (\sqcup n : 0 \leq n : F \cdot (F^n \cdot CHAOS)) \\
= & \quad \{ \text{ calculus } \} \\
& (\sqcup n : 1 \leq n : F^n \cdot CHAOS) \\
= & \quad \{ \ CHAOS \sqsubseteq F^n \cdot CHAOS \ \} \\
& (\sqcup n : 0 \leq n : F^n \cdot CHAOS)
\end{aligned}$$

It is also the least fixpoint, since (let Q be a fixpoint):

$(\sqcup n : 0 \leq n : F^n \cdot \mathit{CHAOS})$

$\sqsubseteq$ { $\mathit{CHAOS} \sqsubseteq Q$ and F is order preserving }

$(\sqcup n : 0 \leq n : F^n \cdot Q)$

$=$ { Q is a fixpoint }

$(\sqcup n : 0 \leq n : Q)$

$=$ { calculus }

Q

□

An application of this fixpoint theorem is given in the next section.

4.2 Process calculus

This section develops a calculus for sequential handshake processes. It is restricted to sequential processes with undirected ports, thereby excluding input and output of data. Extensions to this calculus, including data communication and assignments, are described informally in Section 4.4. The calculus includes the following operations: parallel composition, extension, concealment, nondeterministic composition, sequential composition, N-fold repetition, infinite repetition, enclosure, and choice. The choice of the operators is inspired by the syntax of Tangram.

Basic sequential processes

The following definition introduces four basic sequential processes.

Definition 4.9 **(stop, skip, a°, and $a^\bullet$)**

Let a be a name.

0. **stop** $= \mathit{STOP} \cdot \langle \emptyset, \emptyset \rangle = \langle \langle \emptyset, \emptyset \rangle, \{\varepsilon\}, \emptyset \rangle$
1. **skip** $= \mathit{SKIP} \cdot \langle \emptyset, \emptyset \rangle = \langle \langle \emptyset, \emptyset \rangle, \emptyset, \{\varepsilon\} \rangle$
2. $a^\circ = \langle a^\circ, \{\varepsilon\}, \{a_0 a_1\} \rangle$
3. $a^\bullet = \langle a^\bullet, \{a_0\}, \{a_0 a_1\} \rangle$

□

Note that a° may both denote a port structure and a sequential process; the same holds for $a^\bullet$. Generally, the context indicates which denotation is intended.

Parallel composition

Definition 4.10 (parallel composition)

Let P and Q be two connectable sequential processes, and let $A = \mathbf{e}(\mathbf{p}P \cup \mathbf{p}Q)$. The parallel composition of P and Q is denoted by $P \,||\, Q$ and defined by

0. $\mathbf{p}(P \,||\, Q) = A$

1. $\mathbf{t}(P \,||\, Q) =$
   $(\mathbf{t}P \;\mathbf{w}\; \mathbf{t}Q \;\cup\; (\mathbf{u}P)^{\mathbf{X}} \;\mathbf{w}\; \mathbf{t}Q \;\cup\; \mathbf{t}P \;\mathbf{w}\; (\mathbf{u}Q)^{\mathbf{X}} \;\cup\; div \cdot (\mathbf{t}P^{\leq} \;\mathbf{w}\; \mathbf{t}Q^{\leq}))\lceil A$

2. $\mathbf{u}(P \,||\, Q) = (\mathbf{u}P \;\mathbf{w}\; \mathbf{u}Q)\lceil A$

where the weave of trace sets V and W in the context of $\mathbf{p}P$ and $\mathbf{p}Q$ is shorthand for

$$\{t : t \in (\mathbf{p}P \cup \mathbf{p}Q)^H \;\wedge\; t\lceil \mathbf{p}P \in V \;\wedge\; t\lceil \mathbf{p}Q \in W : t\}$$

□

This definition closely resembles Definition 3.14. The main addition is the requirement that both P and Q must agree on successful termination.

Property 4.11

0. $P \,||\, Q$ is a sequential process.

1. Parallel composition is commutative, associative, distributive, and continuous.

2. $\mathbf{skip} \,||\, P \;=\; P$.

3. $a^\circ \,||\, a^\bullet = \mathbf{skip}$.

□

Conformant port structures

Connectability of port structures is a requirement for the parallel composition of (sequential) handshake processes. In such a composition, two processes may only share the opposite side of a channel. If sequential processes share the same side of a channel, a different requirement is imposed on the respective port structures: they must be *conformant*.

Definition 4.12 (conformance)

Port structures A and B are *conformant* denoted by $A \mathbin{\triangle\!\!\triangle} B$, if

0. A and B are *compatible*, and
1. $A^\circ \cap B^\bullet = \emptyset$ and $A^\circ \cap B^\bullet = \emptyset$.

Two handshake structures are conformant if their respective port structures are. Conformance of sequential processes is defined similarly.

□

Conformance is related to connectability by the following property.

Property 4.13

$$A \mathbin{\triangle\!\!\triangle} B \;=\; A \bowtie \overline{B}$$

□

Conformance enjoys the following obvious properties.

Property 4.14

0. $A \mathbin{\triangle\!\!\triangle} B \;=\; B \mathbin{\triangle\!\!\triangle} A$.
1. $B \subseteq A \;\Rightarrow\; B \mathbin{\triangle\!\!\triangle} A$. Consequently, $A \mathbin{\triangle\!\!\triangle} \emptyset$ and $A \mathbin{\triangle\!\!\triangle} A$.
2. $\mathbf{p}A \cap \mathbf{p}B = \emptyset \;\wedge\; A$ and B are compatible $\;\Rightarrow\; A \mathbin{\triangle\!\!\triangle} B$.

□

Extension

Some relations and operations on processes are defined only for processes with equal port structures. Under such circumstances, the extension of the port structure of a process may be useful. The extension of P with conformant port

structure A is a sequential process that has port structure $\mathbf{p}P \cup A$, and behaves like P. The following definition relies on the fact that disjoint port structures are both conformant and connectable.

Definition 4.15 (extension)

Let $A \mathrel{\triangle\!\!\triangle} \mathbf{p}P$. The extension of P by A is denoted by $(A)\cdot P$ and defined as

$$P \,||\, SKIP \cdot (A \setminus \mathbf{p}P)$$

□

Property 4.16

Let A, B and $\mathbf{p}P$ be mutually conformant. Then

0. $(A)\cdot P$ is a sequential process.
1. $(\langle \emptyset, \emptyset \rangle)\cdot P = P$
2. $(A)\cdot P = (A \setminus \mathbf{p}P)\cdot P$
3. $(\mathbf{p}P)\cdot P = P$
4. $(B)\cdot(A)\cdot P = (A)\cdot(B)\cdot P = (A \cup B)\cdot P$
5. $(A)\cdot\mathbf{stop} = STOP \cdot A$
6. $(A)\cdot\mathbf{skip} = SKIP \cdot A$

□

The following property is helpful in translating Tangram programs into handshake circuits.

Property 4.17

Let A be a port structure and P a permanent sequential process. Then

$$(A)\cdot P = P \,||\, STOP \cdot (A \setminus \mathbf{p}P)$$

Of course, $(A)\cdot P$ is then also permanent.

□

Concealment

Concealment of a subset of the ports of sequential process P has the effect that handshakes through these concealed ports occur without participation of the environment, and are even invisible to the environment. This concealment may have the effect of hiding unbounded sequences of handshakes through the concealed ports. These possible divergences are taken into account in the following definition.

Definition 4.18 (concealment)

Let $A \mathrel{\triangle\!\!\triangle} \mathbf{p}P$. The behavior of P with A concealed, denoted by $|[A \mid P]|$, is defined as

$$\langle B, (\mathbf{t}P \cup div \cdot \langle \mathbf{p}P \cup \overline{A}, \mathbf{t}P^{\leq} \rangle)\lceil B, \mathbf{u}P\lceil B \rangle$$

where $B = \mathbf{p}P \setminus A$.

□

Concealment enjoys the following properties.

Property 4.19

Let A and B be port structures and P a sequential process such that A, B and $\mathbf{p}P$ are mutually conformant. Then

0. $|[A \mid P]|$ is a sequential process.
1. $|[A \mid P]| = |[A \cap \mathbf{p}P \mid P]|$
2. $|[A \mid |[B \mid P]|]| = |[A \cup B \mid P]|$
3. $|[\langle \emptyset, \emptyset \rangle \mid P]| = P$

□

The following property is helpful in translating Tangram programs into handshake circuits.

Property 4.20

Let A be a port structure and P a permanent sequential process, such that $A \mathrel{\triangle\!\!\triangle} \mathbf{p}P$. Then

$$|[A \mid P]| = P \mid\mid RUN \cdot (\overline{A \cap \mathbf{p}P})$$

Sequential process $|[A \mid P]|$ is then also permanent.

□

Nondeterministic composition

This subsection generalizes nondeterministic composition of sequential processes with equal port structures to sequential processes with different, yet conformant, port structures.

Definition 4.21 (nondeterministic composition)

Let $P \mathrel{\triangle\!\!\triangle} Q$. The nondeterministic composition of P and Q is denoted by $P \sqcap Q$, and defined as

$$(\mathbf{p}Q)\cdot P \sqcap (\mathbf{p}P)\cdot Q$$

□

Property 4.22

Let P and Q be conformant sequential processes.

0. $P \sqcap Q$ is a sequential process.
1. Nondeterministic composition is idempotent, commutative, associative, distributive, and continuous (cf. continuity of sequential composition below).
2. $P \sqcap CHAOS\cdot \mathbf{p}P = CHAOS\cdot \mathbf{p}P$

□

Sequential composition

The sequential composition of P and Q first behaves like P and, upon successful termination of P, continues to behave like Q. The definition of sequential composition starts with the sequential composition of sequential processes with equal port structures.

Definition 4.23 (sequential composition)

0. Let P and Q be handshake processes with port structure A. The sequential composition of P and Q is denoted by $P;_A Q$, and defined as

$$\langle A, \mathbf{t}P \cup (\mathbf{u}P;\mathbf{t}Q)^{\mathbf{r}}, (\mathbf{u}P;\mathbf{u}Q)^{\mathbf{r}} \rangle$$

where the sequential composition of trace sets V and W is defined by

$$V;W = \{v,w : v \in V \wedge w \in W : vw\}$$

1. Let P and Q be handshake processes with conformant port structure. The sequential composition of P and Q is denoted by $P;Q$, and defined as

$$(\mathbf{p}Q)\cdot P \ ;_A \ (\mathbf{p}P)\cdot Q$$

where $A = \mathbf{p}P \cup \mathbf{p}Q$.

□

Note that if $\mathbf{p}P = \mathbf{p}Q$ we have $P ;_{\mathbf{p}P} Q \ = \ P;Q$.

Property 4.24

0. $P;Q$ is a sequential process.
1. Sequential composition is associative and distributive.
2. **skip**; $P \ = \ P \ = \ P;$ **skip**
3. **stop**; $P \ =$ **stop**
4. $P;$ **stop** $= \langle \mathbf{p}P, \mathbf{t}P \cup \mathbf{u}P^{\mathbf{x}\,\mathbf{r}}, \emptyset \rangle$, which is clearly permanent.

□

The next property finds application in the definition of infinite repetition.

Property 4.25

Sequential composition is continuous in both operands, that is

$$(\sqcup i : 0 \leq i : P_i;Q) \ = \ (\sqcup i : 0 \leq i : P_i);Q$$

and

$$(\sqcup i : 0 \leq i : P;Q_i) \ = \ P;(\sqcup i : 0 \leq i : Q_i)$$

We prove the latter.

Proof Continuity is proven for the terminal traces; the proof for the quiescent traces is similar.

$$
\begin{array}{ll}
 & t \in \mathbf{u}(\sqcup i :: P; Q_i) \\
= & \quad \{ \text{ definitions of } \sqcup \text{ and sequential composition } \} \\
 & t \in (\cap i :: (\mathbf{u}P; \mathbf{u}Q_i)^{\mathbf{r}}) \\
= & \quad \{ \text{ continuity of } \mathbf{r}\text{-closure (Property 2.41) } \} \\
 & t \in (\cap i :: (\mathbf{u}P; \mathbf{u}Q_i))^{\mathbf{r}} \\
= & \quad \{ \text{ definition of } \mathbf{r}\text{-closure } \} \\
 & (\exists u : t \ \mathbf{r} \ u : u \in (\cap i :: \mathbf{u}P; \mathbf{u}Q_i)) \\
= & \quad \{ \text{ definitions of chain and sequential composition } \} \\
 & (\exists u : t \ \mathbf{r} \ u : (\forall i :: (\exists r, s : u = rs : r \in \mathbf{u}P \wedge s \in \mathbf{u}Q_i)) \\
= & \quad \{ \text{ finite number of } u \text{ and hence of } s \text{ (cf. Property 2.41) } \} \\
 & (\exists u, r, s : t \ \mathbf{r} \ u \wedge u = rs \wedge r \in \mathbf{u}P : (\forall i :: (s \in \mathbf{u}Q_i))) \\
= & \quad \{ \text{ definition of limit; calculus } \} \\
 & (\exists r, s : t \ \mathbf{r} \ rs \wedge r \in \mathbf{u}P : s \in (\cap i :: \mathbf{u}Q_i)) \\
= & \quad \{ \text{ definitions of sequential composition, } \mathbf{r}\text{-closure, and limit } \} \\
 & t \in (\mathbf{u}P; \mathbf{u}(\sqcup i :: Q_i))^{\mathbf{r}} \\
= & \quad \{ \text{ definition of sequential composition } \} \\
 & t \in \mathbf{u}(P; (\sqcup i :: Q_i))
\end{array}
$$

□

N-fold repetition

The 0-fold repetition of P behaves like **skip**. For positive N, the N-fold repetition of P behaves like P, and after successful termination of P behaves like the $(N-1)$-fold repetition of P.

Definition 4.26 (N-fold repetition)

Let N be a natural number. The N-fold repetition of P is denoted by $\#N[P]$, and defined as

$$\begin{array}{lll} \textbf{if} & N = 0 & \rightarrow \ \mathit{SKIP} \cdot \mathbf{p}P \\ [] & N > 0 & \rightarrow \ P; \#(N-1)[P] \\ \textbf{fi} & & \end{array}$$

□

Property 4.27

Let P be a sequential processes and let N be a natural number.

0. $\#N[P]$ is a sequential process.
1. Finite repetition is continuous.
2. $\#N[\textbf{stop}] = \textbf{stop}$, for $N > 0$.
3. $\#N[\textbf{skip}] = \textbf{skip}$.

□

Finite repetition is not distributive, i.e. in general we do *not* have

$$\#N[P \sqcap Q] \ = \ \#N[P] \sqcap \#N[Q]$$

Sequential process $\#N[P \sqcap Q]$ may choose between P and Q at *every* step of the iteration; in the case of $\#N[P] \sqcap \#N[Q]$ this choice is made only once. The latter is a refinement of the former.

Infinite repetition

The infinite repetition of P behaves like an infinite sequential composition of sequential process P, schematically suggested by $P; P; P; \ldots$.

Definition 4.28 (infinite repetition)

The infinite repetition of P is denoted by $\#[P]$ and defined as the least fixpoint of F, where F is defined by $F \cdot X = P; X$.

□

This fixpoint is the limit of the chain $(i : 0 \leq i : \#i[P]; \mathit{CHAOS} \cdot \mathbf{p}P)$ as explained in Section 4.1.

Property 4.29

0. $\#[P]$ is a sequential process; it is also permanent.
1. Infinite repetition is continuous.
2. $P;\#[P] = \#[P]$
3. $\#[\mathbf{stop}] = \mathbf{stop}$
4. $\#[\mathbf{skip}] = \mathit{CHAOS} \cdot \emptyset$

□

It is interesting to compare these properties with Property 4.27. The last property shows that infinite repetition of P cannot be regarded as the limit of the chain $(i : 0 \leq i : \#i[P])$. Infinite repetition is not distributive, for the same reason as finite repetition.

Enclosure

The *enclosure* of a sequential process P by a passive handshake a° (assume $a^\circ \notin \mathbf{p}P$) first behaves like $\mathit{STOP} \cdot \mathbf{p}P$. After event a_0 it behaves like P, and if P terminates successfully, the enclosure terminates successfully with event a_1.

Definition 4.30 (**enclosure**)

Let a° be a port structure, such that a° is not contained in $\mathbf{p}P$. The enclosure of P by a° is denoted by $a^\circ : P$ and defined as

$$\langle a^\circ \cup \mathbf{p}P,\ \ \mathbf{t}(\mathit{STOP} \cdot \mathbf{p}P) \cup (\{a_0\}; \mathbf{t}P)^{\mathbf{r}},\ \ (\{a_0\}; \mathbf{u}P; \{a_1\})^{\mathbf{r}} \rangle$$

□

Property 4.31

Let P be a sequential process, and let a and b be distinct names, such that a° and b° are not contained in $\mathbf{p}P$.

0. $a^\circ : P$ is a sequential process.
1. Enclosure is distributive and continuous.
2. $a^\circ : \mathbf{skip} = a^\circ$

3. $a^\circ : \mathbf{stop} = STOP \cdot a^\circ$

4. $a^\circ : b^\circ : P = b^\circ : a^\circ : P$. In particular, $a^\circ : b^\circ = b^\circ : a^\circ$.

□

The last property, which may come somewhat as a surprise, is a direct consequence of the reordering in Definition 4.30.

Choice

Consider the conformant sequential processes $a^\circ; P$ and $b^\circ; Q$. Sequential processes of this form are called *guarded* processes. We are interested in a sequential process that either behaves like $a^\circ; P$ or like $b^\circ; Q$, such that the environment may choose between the two sequential processes by either offering a_0 or b_0. This is quite different from $a^\circ; P \sqcap b^\circ; Q$, in which the choice between the operands of $\sqcap$ is made nondeterministically.

The above choice is denoted by $[a^\circ; P \mid b^\circ; Q]$, and behaves like the sequential process $a^\circ; P \sqcap b^\circ; Q$, except that traces a_0 and b_0 are not quiescent. The choice construct can be generalized by allowing an enclosures as guarded processes. Recall that an enclosure is a sequential process of the form $a^\circ : P$, This brings us to the following definitions.

Definition 4.32 (guarded process)

A guarded process is a sequential process that can be written as $b^\circ; P$ or as $b^\circ : P$. Port b° is called the *guard* of the guarded process.

□

Recall that $b^\circ = b^\circ; \mathbf{skip} = b^\circ : \mathbf{skip}$.

Definition 4.33 (choice)

Let P and Q be two conformant guarded processes, with disjoint guards p and q. If P is an enclosure we require $p \notin \mathbf{p}^\circ P$ and similarly for Q and q. The choice between P and Q is denoted by $[P \mid Q]$, and defined as

$$\langle A,\ \mathbf{t}((A)\cdot P) \setminus \{q_0\} \cup \mathbf{t}((A)\cdot Q) \setminus \{p_0\},\ \mathbf{u}P \cup \mathbf{u}Q \rangle$$

where $A = \mathbf{p}P \cup \mathbf{p}Q$.

□

The difference with the nondeterministic composition of P and Q is rather subtle. In contrast with nondeterministic composition trace p_0 is not a quiescent trace, *unless* it is a quiescent trace of P; similarly for trace q_0 and Q.

Property 4.34

Let P and Q be two conformant guarded processes.

0. $[P \mid Q]$ is a sequential process.
1. Choice is commutative, distributive and continuous.
2. $[a^\circ : \mathbf{stop} \mid b^\circ : \mathbf{stop}] = STOP \cdot (a^\circ \cup b^\circ)$
3. $[a^\circ; b^\circ \mid b^\circ; a^\circ] = a^\circ \parallel b^\circ$

□

The sequential process denoted in the last property is

$$\langle a^\circ \cup b^\circ, \{\varepsilon, a_0a_1, b_0b_1\}, \{a_0a_1b_0b_1\}^{\mathbf{r}} \cup \{b_0b_1a_0a_1\}^{\mathbf{r}} \rangle$$

The choice construct can readily be generalized to provide a choice among N, $N > 0$, guarded processes.

4.3 Examples

Since handshake processes correspond to permanent sequential handshake processes, the calculus of Section 4.2 can be used to specify handshake processes.

Example 4.35

Most handshake processes of Example 2.23 are repeated below, but now represented by expressions in the handshake calculus.

0. $STOP \cdot (a^\circ) = (a^\circ) \cdot \mathbf{stop}$
1. $RUN \cdot (a^\circ) = \#[a^\circ]$
2. $CON \cdot (a^\circ, b^\bullet) = \#[a^\circ : b^\bullet]$
3. $OR \cdot (a^\circ, b^\bullet, c^\bullet) = \#[a^\circ : (b^\bullet \sqcap c^\bullet)]$
4. $SEQ \cdot (a^\circ, b^\bullet, c^\bullet) = \#[a^\circ : (b^\bullet; c^\bullet)]$

5. $DUP \cdot (a^\circ, b^\bullet) = \#[a^\circ : (b^\bullet; b^\bullet)]$
6. $REP \cdot (a^\circ, b^\bullet) = (a^\circ : \#[b^\bullet])$
7. $PAR \cdot (a^\circ, b^\bullet, c^\bullet) = \#[a^\circ : (b^\bullet \,||\, c^\bullet)]$
8. $MIX \cdot (a^\circ, b^\circ, c^\bullet) = \#[[a^\circ : c^\bullet \mid b^\circ : c^\bullet]]$
9. $PAS \cdot (a^\circ, b^\circ) = \#[a^\circ : b^\circ]$
10. $JOIN \cdot (a^\circ, b^\circ, c^\bullet) = \#[a^\circ : b^\circ : c^\bullet]$
11. $COUNT_N \cdot (a^\circ, b^\bullet) = \#[a^\circ : \#N[b^\bullet]]$

□

In Chapter 2 we have established the above components to be *initial-when-closed*, meaning that for each closed trace t in $P^{\leq}$ the process $after \cdot (t, P)$ is a refinement of P itself. The property initial-when-closed can be checked syntactically, as shown by the next property.

Property 4.36

A sequential handshake process that can be written in the form $\#[a^\circ : P]$ or in the form $\#[a^\circ : P \mid b^\circ : Q]$ is permanent. The corresponding (non-sequential) handshake process is both passive and initial-when-closed.

□

Note that $(a^\circ) \cdot$**stop** can also be written as $\#[a^\circ :$ **stop**$]$. As a matter of fact, all handshake components required for the compilation of Tangram can be written in one of the two forms of Property 4.36.

4.4 Directed communications

With the calculus introduced so far we can only specify sequential processes with undirected ports. The handshake circuits obtained by the translation of Tangram programs also require handshake components with input and output ports, such as $VAR_{Bool} \cdot (a^\circ, b^\circ)$ in Example 2.23. This section extends the calculus of the previous section to sequential handshake processes that communicate data. These extensions are introduced informally and applied to the specification of handshake components.

Declarations

The first operand of the concealment construct $|[A \mid P]|$ may also contain declarations of variables. For instance, x : **var** T declares variable x of type T. The scope of such a declaration is delineated by the enclosing bracket pair.

Input

Assume port definition $a°?T$, where T is a finite set of values, such as *Bool*. The sequential process $a°?x$ has the above port structure, and responds to any input $a_0{:}v$, with $v \in T$, with an acknowledgement a_1.

The x in $a°?x$ is a variable that denotes the incoming value (cf. Tangram), and may be referenced elsewhere. In the sequential composition $a°?x; F \cdot x$ the second operand $F \cdot x$ denotes a sequential process whose behavior depends on the value of x. In the enclosure $a°?x : F \cdot x$ the acknowledgement to an incoming value through $a°$ is postponed until after the successful termination of $F \cdot x$. Both $a°?x; F \cdot x$ and $a°?x : F \cdot x$ are guarded processes and may therefore occur as alternatives in choice processes.

With active port $a•?T$, sequential process $a•?x$ requests an input by outputting a_0. Then it is receptive to all inputs $a_1 : v$ with $v \in T$. In the sequential composition $a•?x; F \cdot x$ the behavior of sequential process $F \cdot x$ depends on the value of x, being the most recent value input through $a•$.

Output

An example of a sequential process whose behavior depends on the value of x is the output process $a°!E(x)$, where $E(x)$ is an expression in which x occurs as a free variable. The enclosure $a°!E(x) \ : P$ is a sequential process, that behaves like P after communication a_0, and, after successful termination of P *and* successful evaluation of E, concludes with communication $a_1 : E(x)$. The value of $E(x)$ may depend on P. For instance, the terminal traces of $b°!x : a•?x$ are $b_0\ a_0\ a_1{:}v\ b_1{:}v$, with v ranging over T. The active counterpart of $a°!E(x)$ is $a•!E(x)$.

Guarded selection

The equivalent of guarded commands (see Chapter 1) can be introduced as well. With B a Boolean expression, **if** $B \rightarrow P$ **fi** behaves as P if B evaluates to *true*, and is left unspecified otherwise. This generalizes to selections with multiple guards in the well-known way.

Sequential process **do** $B \rightarrow P$ **od** repeatedly behaves as P, as long as B evaluates to *true*. In particular **do false** $\rightarrow P$ **od** behaves like $SKIP \cdot (\mathbf{p}P)$ and **do true** $\rightarrow P$ **od** behaves like $\#[P]$.

Examples

The handshake processes below illustrate the above extensions. Together with the handshake processes of Example 2.23 they form a complete list of all handshake components required for the compilation of Tangram programs.

Example 4.37

0. $STOP_{(?,T)} \cdot (a^\circ)$ accepts any input $a:v$, with $v \in T$, but does not respond to it:

   $STOP_{(?,T)} \cdot (a^\circ)$
   $=$
   $STOP \cdot (a^\circ ?T)$

1. $STOP_{(!,T)} \cdot (a^\circ)$ does not respond to a request for a value in T.

   $STOP_{(!,T)} \cdot (a^\circ)$
   $=$
   $STOP \cdot (a^\circ !T)$

2. $RUN_{(?,T)} \cdot (a^\circ)$ repeatedly responds to any input through a of type T with the acknowledgement a_1:

   $RUN_{(?,T)} \cdot (a^\circ)$
   $=$
   $RUN \cdot (a^\circ ?T)$

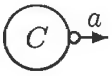
3. $RUN_{(!,T)} \cdot (a^\circ)$ repeatedly responds to a_0 by an output of the form $a_1 : v$, with $v \in T$, and is clearly static nondeterministic:

   $RUN_{(!,T)} \cdot (a^\circ)$
   $=$
   $RUN \cdot (a^\circ !T)$

4. With C a constant, $CST_{\cdot C}\,(a^\circ)$ repeatedly responds to input a_0 with output $a_1 : C$. Process $CST_{\cdot C}\,(a^\circ)$ may be regarded as a deterministic refinement of $RUN_{(!,T)} \cdot (a^\circ)$, provided that the value of C is in T.

$CST{\cdot}_C\,(a^\circ)$
$=$
$(a^\circ!\{C\}){\cdot}\#[a^\circ!C]$



5. *Connector* $CON_{(?,T)}{\cdot}\,(a^\circ, b^\bullet)$ repeatedly passes a value of set T arriving at passive port a through active port b.

$CON_{(?,T)}{\cdot}\,(a^\circ, b^\bullet)$
$=$
$(a^\circ?T, b^\bullet!T){\cdot}\#[|[x :\ \mathbf{var}\ T \mid a^\circ?x : b^\bullet!x]|]$

6. *Connector* $CON_{(!,T)}{\cdot}\,(a^\circ, b^\bullet)$ is similar to $CON_{(?,T)}{\cdot}\,(a^\circ, b^\bullet)$, except that it is "demand driven", whereas the latter is "data driven".

$CON_{(!,T)}{\cdot}\,(a^\circ, b^\bullet)$
$=$
$(a^\circ!T, b^\bullet?T){\cdot}\#[|[x :\ \mathbf{var}\ T \mid a^\circ!x : b^\bullet?x]|]$

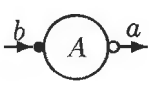
7. $UN_{(\Box,T)}{\cdot}\,(a^\circ, b^\bullet)$ behaves rather similar to $CON_{(!,T)}{\cdot}\,(a^\circ, b^\bullet)$. The main difference is that the value output through a° is $\Box v$, where v is the most recent value input through $b^\bullet$, and $\Box$ is a unary operator. If $\Box v$ is not defined, the subsequent behavior is left unspecified. The type of port a° is $\Box T$.

$UN_{(\Box,T)}{\cdot}\,(a^\circ, b^\bullet)$
$=$
$(a^\circ!\Box T, b^\bullet?T){\cdot}\#[|[x :\ \mathbf{var}\ T \mid a^\circ!(\Box x) : b^\bullet?x]|]$

Examples of unary operators are '$\neg$' and '$-$'.

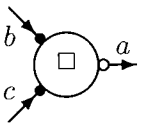
8. $ADAPT_{(T,U)}{\cdot}(a^\circ, b^\bullet)$ is a specialization of $UN_{(\Box,T)}{\cdot}(a^\circ, b^\bullet)$. For input values in T it behaves as a connector. After reception of a value in $U \setminus T$ its subsequent behavior is left unspecified.

$ADAPT_{(T,U)}{\cdot}\,(a^\circ, b^\bullet)$
$=$
$(a^\circ!T, b^\bullet?U){\cdot}$
$\#[|[x :\ \mathbf{var}\ T \cup U \mid a^\circ!x : (b^\bullet?x;\ \mathbf{if}\ x \in T \rightarrow \mathbf{skip}\ \mathbf{fi}\ )]|]$



9. $BIN_{(\Box,U,V)}{\cdot}\,(a^\circ, b^\bullet, c^\bullet)$ is the generalization of $UN_{(\Box,T)}{\cdot}\,(a^\circ, b^\bullet)$ to binary operators. The type of port a° is $U \Box V$.

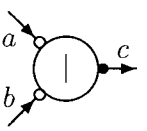
$BIN_{(\Box,U,V)}\cdot(a^\circ, b^\bullet, c^\bullet)$
=
$(a^\circ!(U\Box V), b^\bullet?U, c^\bullet?V)\cdot$
#[|[x : **var** U & y : **var** V
| $a^\circ!(x\Box y) : (b^\bullet?x \parallel c^\bullet?y)$
]|
]



Examples of binary operators are '$\vee$', '$\wedge$', '=', '$<$', '+', '$-$', and '$*$'.

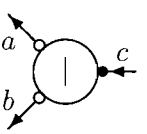
10. $MIX_{(!,T)}\cdot(a^\circ, b^\circ, c^\bullet)$ is one of the two generalizations of $MIX\cdot(a^\circ, b^\circ, c^\bullet)$ that will be considered. Incoming values of type T through ports $a^\circ?T$ and $b^\circ?T$ are passed through $c^\bullet!T$. The subsequent acknowledgement through $c^\bullet!T$ is routed to the origin of the last message. $MIX_{(!,T)}\cdot(a^\circ, b^\circ, c^\bullet)$ may be called a *multiplexer*.

$MIX_{(!,T)}\cdot(a^\circ, b^\circ, c^\bullet)$
=
$(a^\circ?T, b^\circ?T, c^\bullet!T)\cdot$
#[|[x : **var** T | [$a^\circ?x : c^\bullet!x$ | $b^\circ?x : c^\bullet!x$]]|]



11. $MIX_{(?,T)}\cdot(a^\circ, b^\circ, c^\bullet)$ is the other generalization of $MIX\cdot(a^\circ, b^\circ, c^\bullet)$. Requests through $a^\circ!T$ and $b^\circ!T$ are passed through $c^\bullet?T$. The subsequent value input through $c^\bullet?T$ is passed through the output port where the request came from. An appropriate name for this process is *demultiplexer*.

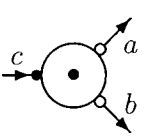
$MIX_{(?,T)}\cdot(a^\circ, b^\circ, c^\bullet)$
=
$(a^\circ!T, b^\circ!T, c^\bullet?T)\cdot$
#[|[x : **var** T | [$a^\circ!x : c^\bullet?x$ | $b^\circ!x : c^\bullet?x$]]|]



If a multiplexer is considered as the data driven generalization of the mixer, the demultiplexer is to be considered as its demand driven generalization.

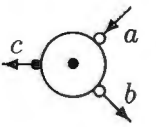
12. $JOIN_{(?,T)}\cdot(a^\circ, b^\circ, c^\bullet)$ generalizes the $JOIN\cdot(a^\circ, b^\circ, c^\bullet)$ in a demand driven way. Requests through a and b are joined before the request is passed through c. The incoming data through c is forked through a and b.

$JOIN_{(?,T)}\cdot(a^\circ, b^\circ, c^\bullet)$
=
$(a^\circ!T, b^\circ!T, c^\bullet?T)\cdot$#[|[$x$: **var** T | $a^\circ!x : b^\circ!x : c^\bullet?x$]|]



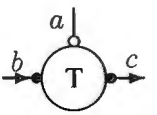
13. $JOIN_{(!,T)}\cdot(a^\circ, b^\circ, c^\bullet)$ generalizes the $JOIN\cdot(a^\circ, b^\circ, c^\bullet)$ in a data driven way, though in a rather subtle way. The incoming data through a is joined with a request through b before the value is passed through c. An acknowledgement through c leads to an acknowledgement through a and an output of the value through b.

$JOIN_{(!,T)}\cdot(a^\circ, b^\circ, c^\bullet)$
=
$(a^\circ?T, b^\circ!T, c^\bullet!T)\cdot\#[|[x:\ \mathbf{var}\ T \mid a^\circ?x : b^\circ!x : c^\bullet!x]|]$



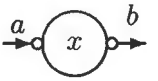
14. $TRF_T\cdot(a^\circ, b^\bullet, c^\bullet)$ is a *transferrer*. Repeatedly, after activation through a° it actively requests for an input through port $b^\bullet?T$ and actively passes the message through port $c^\bullet!T$, before it acknowledges through a°. It is a key component in the translation of Tangram input, output and assignment commands.

$TRF_T\cdot(a^\circ, b^\bullet, c^\bullet)$
=
$(a^\circ, b^\bullet?T, c^\bullet!T)\cdot\#[a^\circ : |[x:\ \mathbf{var}\ T \mid b^\bullet?x; c^\bullet!x]|]$



15. The Boolean variable of Example 2.23 is generalized to a variable of arbitrary type below.

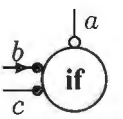
$VAR_T\cdot(a^\circ, b^\circ)$
=
$(a^\circ?T, b^\circ!T)\cdot|[x:\ \mathbf{var}\ T \mid \#[[a^\circ?x \mid b^\circ!x]]]|$



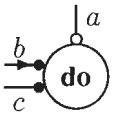
Variable x is declared outside the infinite repetition. This ensures that the value output with $b^\circ!x$ is the most recent value input through a°.

16. The last three components are needed for the translation of Tangram's guarded commands. Component $IF\cdot(a^\circ, b^\bullet, c^\bullet)$ responds to an a_0 by an active input of a Boolean value through $b^\bullet?Bool$. If this value equals *false*, its subsequent behavior is left unspecified. If this value is *true*, an active handshake through $c^\bullet$ follows and after a subsequent a_1 the component returns to its initial state.

$IF\cdot(a^\circ, b^\bullet, c^\bullet)$
=
$(a^\circ, b^\bullet?\mathbf{bool}, c^\bullet)\cdot$
$\#[a^\circ : |[x:\ \mathbf{var\ bool} \mid b^\bullet?x;\ \mathbf{if}\ x \rightarrow c^\bullet\ \mathbf{fi}\ ]|]$



17. Component $DO{\cdot}(a^\circ, b^\bullet, c^\bullet)$ responds to an a_0 by an active input of a Boolean value through $b^\bullet?Bool$. If this value equals *true*, an active $c^\bullet$ comes next, followed by another active input through $b^\bullet?Bool$. This repeats until the value *false* arrives. When the value *false* arrives, the component returns to its initial state after an a_1.

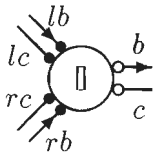


$DO{\cdot}(a^\circ, b^\bullet, c^\bullet)$
=
$(a^\circ, b^\bullet?\mathbf{bool}, c^\bullet){\cdot}$
$\#[a^\circ : |[x : \mathbf{var\ bool} \mid b^\bullet?x;\ \mathbf{do}\ x \rightarrow c^\bullet; b^\bullet?x\ \mathbf{od}\ ]|]$

18. $BAR{\cdot}(b^\circ, c^\circ, lb^\bullet, lc^\bullet, rb^\bullet, rc^\bullet)$ is the most complex component for more than one reason. Firstly, it has as many as 6 ports, organized in three pairs (b, c), (lb, lc) and (rb, rc). Secondly, it combines two more or less independent behaviors, from which the environment can choose. Its behavior is best explained by the restricted form in which it will be used in compiled Tangram programs: as a 2-phase behavior.

Phase 0 starts with a request for a Boolean output through b. This request is forked through lb and rb. The disjunction of the incoming Boolean values is then returned through b. Let x and y denote these incoming Boolean values.

Phase 1 starts with a request through c°. Depending on the values of x and y, the component responds with an active handshake through lc (if x = *true*), or with an active handshake through rc (if y = *true*). If both x and y are *true*, the choice between lc and rc is nondeterministic. If both values are *false* the subsequent behavior is left unspecified.



$BAR{\cdot}(b^\circ, c^\circ, lb^\bullet, lc^\bullet, rb^\bullet, rc^\bullet)$
=
$(b^\circ!\mathbf{bool}, c^\circ, lb^\bullet?\mathbf{bool}, lc^\bullet, rb^\bullet?\mathbf{bool}, rc^\bullet){\cdot}$
$|[\ x, y : \mathbf{var\ bool}$
$|\ \ \#[\ \ [b^\circ!(x \vee y)\ \ : (lb^\bullet?x \parallel rb^\bullet?y)$
$\qquad\quad |\ c^\circ\ \ \ \ \ \ \ \ : \mathbf{if}\ x \rightarrow lc^\bullet\ [\!]\ y \rightarrow rc^\bullet\ \mathbf{fi}$
$\qquad\quad]$
$\ \ \ ]$
$\ ]|$

□

Chapter 5

Tangram

5.0 Introduction

Tangram is a VLSI-programming language based on CSP. The main construct of Tangram is the *command*. Commands are either primitive commands, such as $a?x$ and $x := x + 1$, or composite commands, such as $R; S$ and $R \,||\, S$, where R and S are commands themselves.

Execution of a command may result in a number of communications with the environment through external ports. Another form of interaction with the environment is the reading from and writing into external variables. A Tangram *program* is a command without external variables, prefixed by an explicit definition of its external ports.

Not all compositions of commands are valid in Tangram. For instance, in a sequential composition the two constituent commands must agree on the input/output direction of their common ports. Also, two commands composed in parallel may not write concurrently into a common variable. Section 5.1 defines the syntax of Tangram, including these composition rules. The meaning of each command is described informally.

For a subset of the Tangram commands the *handshake-process* denotations are given in Section 5.3. This subset is referred to as *Core Tangram*.

5.1 Tangram

The main syntactic constructs of Tangram are program, command, guarded-command set, and expression. With each construct we will associate a so-called alphabet structure: a set of typed ports and variables.

Alphabet structures

Let *Val* denote the set of all values. *Val* includes the Boolean values *Bool*, $Bool = \{false, true\}$, and the integer numbers. *Val* also contains the special *null* value ˜. A *type* is a finite subset of '*Val*'. The set of all types is denoted by $\mathcal{P} \cdot Val$, viz. the power set of *Val*. Ports and variables are typed. The type information of ports and variables is recorded in an *alphabet structure*.

Definition 5.0 (alphabet structure)

0. An alphabet structure A is a 5-tuple $\langle \mathbf{p}?A, \mathbf{p}!A, \mathbf{v}?A, \mathbf{v}!A, \tau_A \rangle$, where $\mathbf{p}?A$, $\mathbf{p}!A$, $\mathbf{v}?A$, $\mathbf{v}!A$ are sets of names.

1. $\mathbf{p}?A$ is the set of *input ports*, and $\mathbf{p}!A$ the set of *output ports*. $\mathbf{p}?A$ and $\mathbf{p}!A$ must be disjoint; their union is denoted by $\mathbf{p}A$.

2. $\mathbf{v}?A$ is the set of *read ports* and $\mathbf{v}!A$ the set of *write ports*. Sets $\mathbf{v}?A$ and $\mathbf{v}!A$ need not be disjoint, because a process may read from and write into the same variable. The union of the two sets is denoted by $\mathbf{v}A$.

3. $\mathbf{p}A$ and $\mathbf{v}A$ must be disjoint. The set of all ports, i.e. the union of $\mathbf{p}A$ and $\mathbf{v}A$, is denoted by $\mathbf{c}A$.

4. τ_A is the *type function* of alphabet structure A:

$$\tau_A : \mathbf{c}A \rightarrow \mathcal{P} \cdot Val$$

   It assigns a type to each name in $\mathbf{c}A$.

5. The empty alphabet structure $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ is abbreviated to $\emptyset$.

□

Symbols **p**?, **p**!, **p**, **v**?, **v**!, **v** and **c** are considered as operators on alphabet structures. A concise notation for alphabet structures is based on so-called *port definitions*. A port definition may have one of the four forms below.

Definition 5.1 (port definition)

Let a and x be names and let T be a finite subset of *Val*.

0. $a^{\tilde{}} = \langle \{a\}, \emptyset, \emptyset, \emptyset, \{(a, \tilde{})\} \rangle$, a synchronization port.

1. $a?T = \langle \{a\}, \emptyset, \emptyset, \emptyset, \{(a, T)\} \rangle$, an input port of type T.

2. $a!T = \langle \emptyset, \{a\}, \emptyset, \emptyset, \{(a, T)\}\rangle$, an output port of type T.
3. $x : T = \langle \emptyset, \emptyset, \{x\}, \{x\}, \{(x, T)\}\rangle$, a variable of type T.

□

The following definition introduces a few notions that are useful when composing Tangram commands and expressions. Let, for the remainder of this chapter, A and B be alphabet structures.

Definition 5.2 (relations and operations on alphabet structures)

0. A and B are *type compatible* if common names are either ports or variables in both alphabet structures, and if these common names are of the same type, i.e. if

$$\begin{aligned} & (\mathbf{p}A \cap \mathbf{v}B = \emptyset) \ \wedge \ (\mathbf{p}B \cap \mathbf{v}A = \emptyset) \\ \wedge \ & (\mathbf{A}a : a \in \mathbf{c}A \cap \mathbf{c}B : \tau_A \cdot a = \tau_B \cdot a) \end{aligned}$$

1. A and B are *conformant*, denoted by $A \mathbin{\triangle\!\!\triangle} B$, if they are type compatible and if their common ports agree in direction:

$$\begin{aligned} A \mathbin{\triangle\!\!\triangle} B = \quad & A \text{ and } B \text{ are type compatible} \\ \wedge \ & (\mathbf{p}?A \cap \mathbf{p}!B = \emptyset) \ \wedge \ (\mathbf{p}?B \cap \mathbf{p}!A = \emptyset) \end{aligned}$$

2. The *conformant union* of two conformant alphabet structures A and B is denoted by $A \cup_{\triangle\!\!\triangle} B$, and is defined as the componentwise union of A and B.

3. The *conformant difference* of two conformant alphabet structures A and B is denoted by $A \setminus_{\triangle\!\!\triangle} B$, and is defined as componentwise set difference of A and B.

4. A and B are *connectable*, denoted by $A \bowtie B$, if they are type compatible, have no common output ports, and variables with write access of one structure do not occur in the other structure:

$$\begin{aligned} A \bowtie B = \quad & A \text{ and } B \text{ are type compatible} \\ \wedge \ & (\mathbf{p}!A \cap \mathbf{p}!B = \emptyset) \ \wedge \ (\mathbf{v}A \cap \mathbf{v}!B = \emptyset) \ \wedge \ (\mathbf{v}!A \cap \mathbf{v}B = \emptyset) \end{aligned}$$

5. The *connectable union* of two conformant alphabet structures A and B is denoted by $A \cup_{\bowtie} B$, and is defined as componentwise union of A and B, *except* for $\mathbf{p}?(A \cup_{\bowtie} B)$:

$$\mathbf{p}?(A \cup_{\bowtie} B) = (\mathbf{p}?A \cup \mathbf{p}?B) \setminus (\mathbf{p}!A \cup \mathbf{p}!B)$$

i.e. the output ports dominate.

6. Let D denote a list of port definitions that define mutually conformant alphabet structures. Then D defines the alphabet structure formed by the conformant union of the alphabet structures of the individual port definitions. This alphabet structure is denoted by $\mathbf{A}D$.

□

Alphabet structures will be defined for Tangram programs, commands, guarded-command sets and expressions. In many instances, these alphabet structures are expressed in terms of the alphabet structures of the constituent constructs, with an associated composition rule.

Programs

Let S be a command and let D be a list of port definitions. Then the following table defines valid Tangram programs. (Let $\mathbf{A}S$ denote the alphabet structure of command S.)

construct	Alphabet structure	rule
Program		
$(D)\cdot S$	$\mathbf{A}D$	$\mathbf{A}S \subseteq \mathbf{A}D \ \wedge \ \mathbf{v}\mathbf{A}S = \emptyset$

The (composition) rule states that S has no external variables, and that all externals ports must be defined in D. The behavior of program $(D)\cdot S$ is that of command S. The program does not participate in communications though ports in $\mathbf{A}D \setminus \mathbf{A}S$.

Primitive commands

The primitive commands of Tangram are listed in the table below. Let a be declared[0] as a port of type U, let x be declared as a variable of type V, and let E be an expression with alphabet structure $\mathbf{A}E$.

[0]Declarations are discussed with the block command.

construct	Alphabet structure	rule
Commands		
stop	$\emptyset$	
skip	$\emptyset$	
a	$a^{\sim}$	
$a?x$	$a?U \cup_{\triangle\triangle} \langle \emptyset, \emptyset, \emptyset, \{x\}, \{(x, V)\} \rangle$	$a \neq x$
$a!E$	$a!U \cup_{\triangle\triangle} \mathbf{A}E$	$a \notin \mathbf{v?A}E$
$x := E$	$\langle \emptyset, \emptyset, \emptyset, \{x\}, \{(x, V)\} \rangle \cup_{\triangle\triangle} \mathbf{A}E$	

- **stop** does not engage in any action; it corresponds to an unconditional deadlock.

- **skip** does not engage in any action either, but it does terminate successfully.

- a is a *synchronization command.* The execution of a amounts to a synchronization action through port a.

- $a?x$ is an *input command.* The execution of $a?x$ involves the reception of a value through port a and the subsequent storage of that value in variable x. If the received value is not in V the effect of $a?x$ is left unspecified.

- $a!E$ is an *output command.* The execution of $a!E$ starts with the evaluation of E. If this evaluation terminates and the result is in U, this value is sent through a. Otherwise the behavior of $a!E$ is left unspecified: it may for instance result in deadlock, or in sending an unspecified value ($\in U$) through a.

- $x := E$ is an assignment command. If the evaluation of E terminates and the result value is in V, the execution of $x := E$ assigns the result value of this evaluation to x. Otherwise the behavior of $x := E$ is left unspecified.

Composite commands

All composite commands, except the selection and repetition commands, are listed in a table below. Let R and S be commands, N a natural number, and let D be a list of port definitions.

construct	Alphabet structure	rule
Commands		
$R \sqcap S$	$\mathbf{A}R \cup_{\triangle\!\triangle} \mathbf{A}S$	$\mathbf{A}R \mathbin{\triangle\!\triangle} \mathbf{A}S$
$R; S$	$\mathbf{A}R \cup_{\triangle\!\triangle} \mathbf{A}S$	$\mathbf{A}R \mathbin{\triangle\!\triangle} \mathbf{A}S$
$\#N[S]$	$\mathbf{A}S$	
$\#[S]$	$\mathbf{A}S$	
$R \mid\mid S$	$\mathbf{A}R \cup_{\bowtie} \mathbf{A}S$	$\mathbf{A}R \bowtie \mathbf{A}S$
$\lvert[D \mid S]\rvert$	$\mathbf{A}S \setminus_{\triangle\!\triangle} \mathbf{A}D$	$\mathbf{A}D \mathbin{\triangle\!\triangle} \mathbf{A}S$
$(D)\cdot S$	$\mathbf{A}D \cup_{\triangle\!\triangle} \mathbf{A}S$	$\mathbf{A}D \mathbin{\triangle\!\triangle} \mathbf{A}S$

- $R \sqcap S$ is the *nondeterministic composition* of R and S. This composition behaves either like R or like S, where the selection between them is nondeterministic. This command is included mainly for theoretical interest.

- $R; S$ is the *sequential composition* of R and S. It first behaves like R and, when R terminates successfully, continues by behaving like S. If R does not terminate successfully, neither does $R; S$.

- $\#N[S]$ is the *N-fold repetition* of S. Command $\#0[S]$ behaves like **skip**, and for $N > 0$ the behavior of $\#N[S]$ is that of $S;\ \#(N-1)[S]$.

- $\#[S]$ is the *infinite repetition* of S. It never terminates successfully.

- $R \mid\mid S$ is the *parallel composition* of R and S. Note that R and S are not allowed to have write access to common variables. However, R and S may share input ports and variables with read access.

  The behavior of this composition must agree with the behavior of both R and S. Its execution involves the parallel execution of both R and S. Communications on common ports must occur simultaneously in R and S. The synchronized execution of $a!E$ in one process and $a?x$ in the other has also the effect of $x := E$.

- $\lvert[D \mid S]\rvert$ is a *block (command)*. A port definition in D has two roles in this construct. Firstly, the type function of the alphabet structure of D applies to the ports of S. Secondly, the names declared in D are hidden (concealed) for the environment of the block. In other words, D declares port and variable names, whose scope is bound by the enclosing scope brackets. The behavior of $\lvert[D \mid S]\rvert$ is that of S, with all interaction on ports and variables declared in D concealed for the environment.

- $(D)\cdot S$ extends the alphabet structure of S by the port definitions of D. $(D)\cdot S$ is not prepared to engage in any communication through the ports $\mathbf{A}D \setminus_{\triangle\!\triangle} \mathbf{A}S$; otherwise it behaves like S.

Of the binary command operators, the semicolon binds the strongest, followed by '||' and then '⊓'. As usual, the bracket pair '(' and ')' may be used to overrule this priority rule.

Guarded commands

The selection and repetition commands are listed in the table below. They introduce so-called *guarded-command sets* [Dij75], a third syntactic category next to programs and commands. Let B be a Boolean expression, S a command, and let G and H be guarded-command sets.

construct	Alphabet structure	rule
Commands		
if G **fi** **do** G **od**	$\mathbf{A}G$ $\mathbf{A}G$	
Guarded-commands sets		
$B \rightarrow S$ $G \,[]\, H$	$\mathbf{A}B \cup_{\triangle\!\triangle} \mathbf{A}S$ $\mathbf{A}G \cup_{\triangle\!\triangle} \mathbf{A}H$	$\mathbf{A}B \mathbin{\triangle\!\triangle} \mathbf{A}S$ $\mathbf{A}G \mathbin{\triangle\!\triangle} \mathbf{A}H$

- The execution of a selection command with a guarded-command set G depends on the value of $BB \cdot G$, the disjunction of the guards of G:

$$\begin{aligned} BB \cdot \emptyset &= \mathit{false} \\ BB \cdot (B \rightarrow S) &= \text{value of } B \\ BB \cdot (G \,[]\, H) &= BB \cdot G \vee BB \cdot H \end{aligned}$$

  If $BB \cdot G$ evaluates to *false*, the behavior of the selection command is left unspecified: for instance, it may stop. Otherwise, the selection command behaves like one of the commands of the guarded-command set for which the guard evaluates to *true*.

- **do** G **od** is Tangram's guarded repetition command. As long as $BB \cdot G$ evaluates to *true*, one of the commands for which the guard evaluates to *true* is selected for execution. When $BB \cdot G$ evaluates to *false*, **do** G **od** terminates successfully. Accordingly, **do od** is equivalent to **skip**.

Expressions

Expressions form a fourth syntactic category in Tangram. They occur in assignments, output commands and guards. The four forms of expressions are listed in the table below. Let E and F be expressions, x a variable declared of type V, and let C be a constant.

construct	Alphabet structure	rule
Expressions		
C	$\emptyset$	
x	$\langle \emptyset, \emptyset, \{x\}, \emptyset, \{(x, V)\} \rangle$	
$\Box E$	$\mathbf{A}E$	
$E \Box F$	$\mathbf{A}E \cup_{\triangle\triangle} \mathbf{A}F$	$\mathbf{A}E \triangle\triangle \mathbf{A}F$

- The value of expression C is simply the value of constant C; its type is $\{C\}$. The evaluation of expression C always terminates successfully.

- If x is declared as a variable, then x is also an expression with type V. The value of x is the value of the variable. The initial value of x is in V, but otherwise unspecified. A program may therefore start with $b!x$ as in shift registers of Section 1.3. Successful termination is guaranteed.

- $\Box E$ is an expression constructed from expression E and a unary operator '$\Box$'. The type of $\Box E$ is the set of values obtained by applying $\Box$ to all elements of the type of E. The evaluation of $\Box E$ terminates successfully if that of E does *and* the operator '$\Box$' is defined for the value of E. If the evaluation of E terminates successfully the value of $\Box E$ is obtained by applying $\Box$ to the value of E.

- $E \Box F$ is the natural generalization of $\Box E$ to binary operators.

5.2 Tangram semantics

In Chapter 6 we develop a mapping from Tangram programs to handshake circuits. The (external) behavior of such a handshake circuit has been defined as the handshake process obtained by the parallel composition of its constituent handshake components (cf. Chapter 3). In order to relate a compiled handshake circuit to the original Tangram program, a handshake-process denotation of that Tangram program is required. Given such a denotation, it is sensible to require that the external behavior of the compiled handshake circuit is a refinement of

that handshake-process denotation. In this section we investigate the semantics of Tangram in terms of (sequential) handshake processes.

There are two viable approaches to obtain a handshake-process denotation of a Tangram program:

0. the *direct approach* in which a sequential handshake processes is associated with each Tangram command;

1. the *indirect approach* comprising a denotation in terms of an existing process model (such as the well-known synchronous failures/divergences model of CSP) and a mapping from that model to handshake processes.

The direct approach is pursued in this thesis, and is discussed next. The indirect approach is discussed in Appendix B.

Direct approach

We want to associate a sequential handshake process with each Tangram command. An issue with far-reaching consequences is the choice between a passive or an active implementation for each Tangram port. Tangram itself does not give much of a clue to this, except that choice favors passive ports (see Section 4.2), and read/write accesses to external variables must be through active ports (cf. *VAR* in Example 2.23).

Also connectability requirements in the case of parallel composition have to be considered in the choice between passive or active port implementations. Note e.g. that a Tangram command without output ports is connectable to itself, which is obviously not true for a (sequential) handshake process. Another complication is that broadcast in Tangram (i.e. common ports are not concealed with parallel composition) has no counterpart in sequential handshake processes.

We shall ignore the latter complications for a while and first consider two simple strategies:

- *directed* mappings, viz. inputs passive and outputs active, or vice versa;

- *uniform* mappings, viz. the *all-passive* mapping (all ports implemented passively) or the *all-active* mapping (all ports implemented actively).

Both strategies are viable. Directed mappings have the advantage that directed point-point channels do not give rise to connectability violations. However, a provision has to be made for undirected channels and broadcast in Tangram. The directed mapping "inputs active and outputs passive" results in cheaper circuits than the other directed mapping [Mar89].

A uniform mapping leads to a simple translation strategy, as is shown in Chapter 6. After some simple local optimizations (cf. Section 7.1) circuits are obtained that are comparable in cost and performance with the circuits obtained by the directed mappings. In the sequel we shall consider uniform mappings only.

For both uniform mappings the notions of conformance of (Tangram) alphabet structures and conformance of (handshake) port structures fully agree. However, also for both choices, parallel composition of sequential handshake processes requires redefinition, since connectability is satisfied only in trivial cases.

Consider connectable Tangram commands S and T, and let the corresponding compiled handshake circuits be denoted by $\mathcal{C} \cdot S$ and $\mathcal{C} \cdot T$. In general, these handshake circuits cannot be connected to form a larger handshake circuit, because of connectability violations: some form of "glue" handshake components is needed. We first compare the all-passive and the all-active mappings:

- All-passive tends to result in expensive handshake circuits: here we pay the price for receptiveness, especially for passive input ports (see also Section 7.2). More importantly, glue components that synchronize passive handshakes cannot be realized, because there is not any way to enforce synchronization between two passive two-phase handshakes.

- All-active is relatively straightforward, and cheap. Moreover, passive glue can be realized: e.g. a passivator synchronizes two active handshakes. Moreover, with a *JOIN* as glue component, the common ports of a parallel composition remain accessible for the environment, thus providing the equivalent of broadcast. All-active is also consistent with the requirement for active read/write access to external variables. However, all-active excludes the choice construct, at least with two-phase handshaking.

The all-active approach is clearly favored by the above analysis, and is therefore adopted for the semantics of Tangram. For Core Tangram such a denotation will be given in the next subsection. We expect that this approach can be extended to full Tangram. The price we pay is that a choice construct in Tangram is not accommodated for.

5.3 Core Tangram

Core Tangram is obtained by reducing *Val* to $\{\tilde{}\}$, and by subsequent weeding out all constructs that have become meaningless or redundant. The resulting language

is then defined in terms of sequential handshake processes. With ~ as only value, the Tangram distinction between input and output disappears. Also, the concept of storage is no longer meaningful. Alphabet structures in Core Tangram have the form:

$$\langle \mathbf{p}?A, \emptyset, \emptyset, \emptyset, \{\tilde{}\}\rangle \ .$$

Consequently, an alphabet structures does not contain more information than a set of port names. Also, the notions type compatibility, conformance and connectability of alphabet structures become void: all alphabet structures are both conformant and connectable. For the remainder of this section alphabet structures contain input ports only, all of type $\{\tilde{}\}$. Note also that the syntactic categories *expression* and *guarded-command set* are meaningless when $Val = \{\tilde{}\}$.

Definition 5.3 (port structure of an alphabet structure)

Let A be a alphabet structure of the form $\langle \mathbf{p}?A, \emptyset, \emptyset, \emptyset, \{\tilde{}\}\rangle$. The *handshake expansion* of A, denoted by $\mathcal{H}\cdot A$, is defined as

$$(\cup a : a \in \mathbf{p}?A : a^{\bullet})$$

□

Clearly, $(\mathcal{H}\cdot A)^{\circ} = \emptyset$. The next definition gives the sequential handshake-process denotations for Core Tangram commands.

Definition 5.4 (Core Tangram commands)

Let a be a name, D a list of port definitions, N a natural number, and S and T Core Tangram commands. The ten commands of Core Tangram and their sequential handshake-processes denotations are enumerated below.

0. $\mathcal{H}\cdot \mathbf{skip} = \mathbf{skip}$
1. $\mathcal{H}\cdot \mathbf{stop} = \mathbf{stop}$
2. $\mathcal{H}\cdot a = a^{\bullet}$
3. $\mathcal{H}\cdot((D)\cdot S) = (\mathcal{H}\cdot D)\cdot(\mathcal{H}\cdot S)$
4. $\mathcal{H}\cdot(S \sqcap T) = \mathcal{H}\cdot S \sqcap \mathcal{H}\cdot T$
5. $\mathcal{H}\cdot(S;T) = \mathcal{H}\cdot S \ ; \ \mathcal{H}\cdot T$
6. $\mathcal{H}\cdot(\#N[S]) = \#N[\mathcal{H}\cdot S]$

7. $\mathcal{H}\cdot(\#[S]) = \#[\mathcal{H}\cdot S]$

8. $\mathcal{H}\cdot(S \,||\, T) = \mathcal{H}\cdot S \,||\, \mathcal{H}\cdot T$, provided $\mathbf{A}S \cap \mathbf{A}T = \emptyset_A$.
   A general parallel composition is defined below.

9. $\mathcal{H}\cdot|[D \mid S]| = |[\mathcal{H}\cdot D \mid \mathcal{H}\cdot S]|$

□

General parallel composition in Core Tangram in the above context requires an alternative form of parallel composition of sequential handshake processes. This alternative must deal with common active ports and broadcast. A consequence of the latter is that we do not need to worry about divergences! It is presumably possible to base the definition of this form of parallel composition on the existing definition by introducing glue handshake processes. (For permanent processes this is shown in Chapter 6.) However, this seems to result in a relatively ugly definition, and the successful termination aspect is hard to deal with. A simpler and more direct alternative is developed next.

Consider sequential handshake processes P and Q. Let a be a common (active) port of P and Q, and let trace t satisfy $t\lceil\mathbf{p}P \in \mathbf{t}P^{\leq}$ and similarly for Q: trace t may be observed when P and Q operate in parallel. Assume furthermore that a_0 would lead P subsequently into a quiescent state (i.e. $ta_0 \in \mathbf{t}P$), but that Q is not prepared to participate in a_0 (i.e. $ta_0 \notin \mathbf{t}Q^{\leq}$). In such a situation, the parallel composition of P and Q is quiescent after trace t, even if $t\lceil\mathbf{p}P \notin \mathbf{t}P$ or $t\lceil\mathbf{p}Q \notin \mathbf{t}Q$. This analysis shows that the weave of P and Q is insufficient to describe the effect of parallel composition. In other words: outputs cannot be synchronized.

The following definition of parallel composition is based on the observation that the inability of one process to participate in a common output will force the composite into a quiescent state.

Definition 5.5 (parallel composition in Core Tangram)

Let S and T be connectable Tangram commands. Let $P = \mathcal{H}\cdot S$ and $Q = \mathcal{H}\cdot T$.

0. $\mathcal{H}\cdot(S \,||\, T) = \mathcal{H}\cdot S \overset{\bullet\bullet}{||} \mathcal{H}\cdot T$

1. where $P \overset{\bullet\bullet}{||} Q$ denotes the parallel composition of all-active processes, defined as

$$\langle \mathbf{p}P \cup \mathbf{p}Q, \mathbf{t}P^{\mathbf{S}} \,\mathbf{w}\, \mathbf{t}Q \;\cup\; \mathbf{t}P \,\mathbf{w}\, \mathbf{t}Q^{\mathbf{S}}, \mathbf{u}P \,\mathbf{w}\, \mathbf{u}Q \rangle$$

where weaving of trace sets is used as in Definition 3.14.

2. where $P^{\mathbf{s}}$ is the $\mathbf{s}$ *closure* of P, based on the preorder $\mathbf{s}$ on handshake traces:

$$s \,\mathbf{s}\, t \;=\; (\exists u : u \in (\mathbf{o}P)^* : su = t)$$

s is a prefix of t that can be extended to t with outputs only.

□

Now that we have defined the parallel composition of connectable Tangram commands, we can complete the definition of the semantics of Core Tangram.

Definition 5.6 (Core Tangram program)

A Core Tangram program is an extension command of the form $(D)\cdot S$, such that $\mathbf{A}S \subseteq \mathbf{A}D$.

□

In Chapter 6 we also use the "repeatable go" of $\mathcal{H}\cdot T$.

Definition 5.7 (repeatable go)

Let P be a sequential handshake process. The *repeatable go* of P, denoted by $\triangleright^*\cdot P$, is the sequential handshake process

$$\#[\,\triangleright^\circ : P]$$

where port name $\triangleright$ is pronounced as "go".

□

Consider $\triangleright^*\cdot\mathcal{H}\cdot T$. The environment may start the execution of T by sending a $\triangleright_0$. If T terminates successfully $\triangleright^*\cdot\mathcal{H}\cdot T$ will reply with a $\triangleright_1$. After event $\triangleright_1$ the handshake process is ready for another execution of T. Note also that $\triangleright^*\cdot\mathcal{H}\cdot T$ is passive and initial-when-closed.

Function $\mathcal{H}$ defines a semantics for Tangram. We shall refer to this semantics as *handshake semantics* of Tangram. Extending $\mathcal{H}$ to full Tangram relies on the extension of the calculus in Chapter 4. In Chapter 6 we assume that $\mathcal{H}$ has been extended to cover all Tangram.

Chapter 6

Tangram → handshake circuits

6.0 Introduction

The topic of this chapter is the translation of Tangram programs into handshake circuits. Let T be a Tangram program. In Chapter 5 we have defined the meaning of T as the handshake process $\mathcal{H} \cdot T$. The translation to handshake circuits is presented as a mathematical function $\mathcal{C}$, from the set of Tangram programs to the set of handshake circuits. Thus, $\mathcal{C} \cdot T$ is a handshake circuit, and handshake process $||\cdot \mathcal{C} \cdot T$ is the behavior of that circuit. Function $\mathcal{C}$ will be designed such that

$$\triangleright^* \cdot \mathcal{H} \cdot T \quad = \quad ||\cdot \mathcal{C} \cdot T$$

where $\triangleright^* \cdot P$ was defined as $\#[\, \triangleright^\circ : P]$ (cf. Definition 5.7). That is, the translation preserves all the nondeterminism of the program. From a practical viewpoint it is sufficient to realize

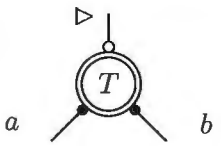
$$\triangleright^* \cdot \mathcal{H} \cdot T \quad \sqsubseteq \quad ||\cdot \mathcal{C} \cdot T$$

in which the behavior of the handshake circuit is a refinement of the handshake behavior of the Tangram program. It may be expected that this relaxed form results in cheaper handshake circuits. The advantage of defining the most nondeterministic handshake circuit of T is that many alternative translation functions that synthesize more deterministic circuits can readily be derived from it. Some of these alternatives will be indicated.

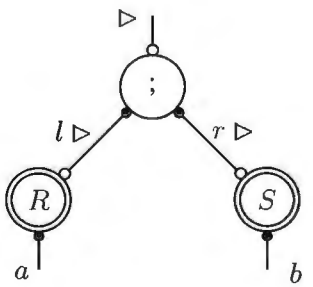
The translation function $\mathcal{C}$ has been described briefly and incompletely in [vBKR*91]. A predecessor of the translation method [vBRS88] is organized

quite differently, but yields essentially the same handshake circuits. Similar syntax-directed translation methods have been presented in [BM88,BS89,Bro90]. A major difference, however, is that these methods translate directly into some form of asynchronous gate-level circuits with a specific timing discipline.

The translation of Tangram programs into handshake circuits is *syntax directed*, that is, the compilation function $\mathcal{C}$ is structured according to the syntax of Tangram. This technique is conveniently introduced by means of an example, in which we apply $\mathcal{C}$ to *command* T. This example will also be used to explain a graphical representation of the compilation function.

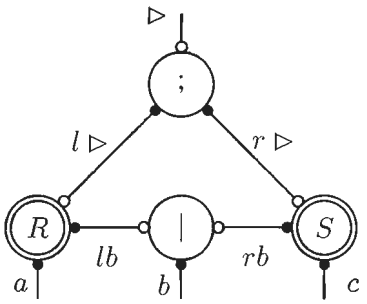


The concentric circles enclosing T denote the application of function $\mathcal{C}$ to the Tangram command T. The peripheral open circle represents the passive port $\triangleright$ and the peripheral filled circles represent the active ports of the compiled circuit. Now, suppose that T is of the form $R;S$, such that the alphabet structures of R and S are disjoint. Syntax-directed translation suggests to construct the handshake circuit for T from the two handshake circuits obtained by the translation of R and S. These subcircuits behave like $\#[l\triangleright^\circ : \mathcal{H}\cdot R]$ and $\#[r\triangleright^\circ : \mathcal{H}\cdot S]$ respectively (with activation ports $l\triangleright$ and $r\triangleright$). The sequential activation of these two circuits can be enforced by connecting them to a sequencer as in



After the circuit is activated through $\triangleright$, by $\triangleright_0$, the sequencer will activate the circuit corresponding to R with $l\triangleright_0$. Successful termination of R, is acknowledged by $l\triangleright_1$, to which sequencer responds with $r\triangleright_0$. If S also terminates successfully it will indicate so by $r\triangleright_1$, and the execution of $R;S$ is completed by $\triangleright_1$. Then the circuit is in its initial state, available for another execution of $R;S$.

When R and S have ports in common, the translation of $R;S$ is only slightly more complicated. Given the disjoint nature of the compiled subcircuits, a "glue" component is required to give both R and S access to a single external port. A mixer for each common port, together with proper renaming of the involved ports, results in the desired handshake circuit. In the circuit below, it is assumed that command R has ports a and b and that command S has ports b and c.



The general translation of Tangram commands of the form $R;S$ is described later. The significance of the above suggested approach is that the compilation function can now be applied recursively to R and S *independently*. The required port renaming will be made more precise later.

Section 6.1 presents the translation of Tangram programs in a semiformal manner. For each Tangram production rule (cf. Section 5.1) a corresponding compilation rule is defined, supported by a graphical version of it and an operational interpretation. The required handshake components have been introduced in Examples 2.23, 4.35, and 4.37.

The aim of Section 6.2 is to formalize the discussed equivalence between Tangram programs and the corresponding handshake circuits in the Compilation Theorem. The scope of the Compilation Theorem is restricted to Core Tangram, as a consequence of similar restrictions in $\mathcal{H}$ and the handshake calculus in Section 4.2.

6.1 Compilation function

The translation of Tangram into handshake circuits is defined by means of compilation function $\mathcal{C}$. The syntax-directed organization of $\mathcal{C}$ makes it necessary to include all syntactic categories of Tangram in the domain of $\mathcal{C}$, viz. program, command, guarded-command set, and expression. The application of $\mathcal{C}$ to an element of each of these categories results in a handshake circuit. The port struc-

tures and behaviors of the handshake circuits corresponding to these syntactic categories are introduced informally below.

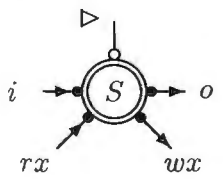
- Let S be a Tangram command. The port structure of $\mathcal{C} \cdot S$ consists of the passive activation port $\triangleright^\circ$ and an active port corresponding to each port of alphabet structure $\mathbf{A}S$. A handshake through $\triangleright^\circ$ results in the execution of S, according to the handshake semantics of Tangram discussed in Chapter 5.

- Let T be a Tangram program. The port structure and behavior are those of *command* T.

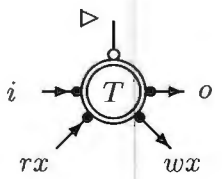
- Let G be a non-empty guarded-command set in Tangram. The handshake circuit $\mathcal{C} \cdot G$ has, in addition to the handshake ports that stem from its alphabet structure, two passive ports, viz. b° and $\triangleright^\circ$. Port b° is a Boolean output port through which the environment may collect the *disjunction of the guards*. Port $\triangleright^\circ$ is the activation port through which the appropriate guarded command is selected for execution.

- Let E be a Tangram expression. The handshake circuit $\mathcal{C} \cdot E$ has a passive output port e° through which the value of E is output. For each variable x that occurs in E the circuit $\mathcal{C} \cdot E$ has a single active read port $rx^\bullet$. Multiple occurrences of x share the same read port.

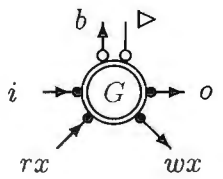
The handshake circuits for the four syntactic categories are depicted below. Here i and o denote an input and an output port respectively, rx and wx denote the read port and write port of a variable x, and the other ports are explained above.



$\mathcal{C} \cdot S$



$\mathcal{C} \cdot T$



$\mathcal{C} \cdot G$

$\mathcal{C} \cdot E$

In the presentation of the compilation function $\mathcal{C}$ the grouping of syntactic constructs of Chapter 5 is followed. The order of these groups is slightly different for didactical purposes. Before defining $\mathcal{C}$ for all of Tangram's production rules, the technical issue of renaming of ports must be dealt with.

Renaming

The translation of composite Tangram commands such as $R;S$ results in a handshake circuit consisting of the sub-circuits $\mathcal{C} \cdot R$ and $\mathcal{C} \cdot S$, and some "glue" circuitry. Part of this glue circuitry is required to deal with ports common to both $\mathcal{C} \cdot R$ and $\mathcal{C} \cdot S$. The introduction of glue components makes it necessary to introduce new names for specifying the interconnections. This requires a systematic way of renaming the activation ports and the common ports of R and S. The names introduced by such a renaming may not clash with existing names. A simple and effective renaming that avoids clashes is to modify *all* names in R and S by prefixing the name with a fixed character string.

Definition 6.0 (renaming)

0. Let n be a name. $\underline{l} \cdot n$ is the $\underline{l}$-renaming of n and equals ln, i.e. the character string n prefixed with the letter l.

1. Let A be an alphabet structure. $\underline{l} \cdot A$ is the alphabet structure A with all port and variable names $\underline{l}$-renamed.

2. Let T be a Tangram command. $\underline{l} \cdot T$ is the command T with all occurrences of port and variable names $\underline{l}$-renamed.

3. Let P be a handshake component. $\underline{l} \cdot P$ is the handshake component P with all occurrences of symbol names $\underline{l}$-renamed. A similar renaming also applies to handshake circuits.

4. The $\underline{r}$-renaming is defined similarly.

□

The following properties of renaming are frequently used.

Property 6.1

$\underline{l}$- and $\underline{r}$- renaming commute with

0. Tangram operators and $\mathcal{H}$ (when applied to commands),

1. parallel composition (when applied to handshake processes), and
2. ∪ (when applied to handshake circuits).

□

Furthermore $\mathcal{C}$ is designed to commute with renaming as well.

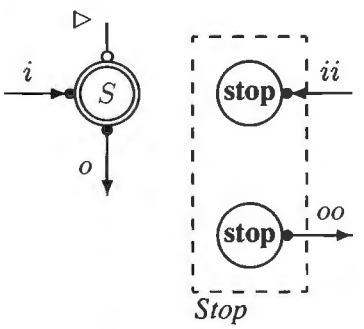
Tangram program

The translation of Tangram program $(B)\cdot S$ yields the same handshake circuit as the translation of the extension command $(B)\cdot S$, which is treated next.

Extension and concealment

Extension

The extension of a command S with an alphabet structure B behaves like S. The ports of B that were not already part of S are simply connected to *STOP* components (cf. Property 4.17).



$\mathcal{C}\cdot((B)\cdot S)$

The definition of this compilation rule is somewhat streamlined by introducing a *Stop* term.

Definition 6.2

0. $\mathcal{C}\cdot((B)\cdot S) = Stop\cdot(B \setminus \mathbf{a}S) \cup \mathcal{C}\cdot S$

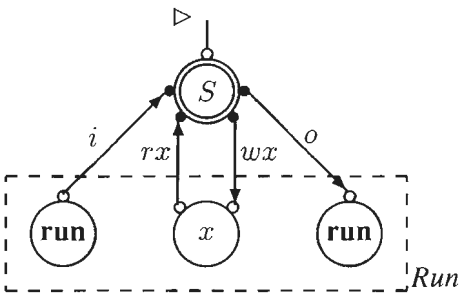
1. Let A be an alphabet structure. $Stop \cdot A$ is the handshake circuit

$$\begin{array}{rl} & \{c : c \in \mathbf{c}?A \quad : STOP_{(?,\tau_c)} \cdot (c^\circ)\} \\ \cup & \{c : c \in \mathbf{c}!A \quad : STOP_{(!,\tau_c)} \cdot (c^\circ)\} \end{array}$$

□

Concealment

In the translation of $|[B \mid S]|$, ports and variables of B have to be treated differently. Ports can simply be connected to appropriate *RUN* components (cf. Property 4.20). Variables to which S has both read and write access are implemented by *VAR* components of the appropriate type. Variables to which S has either read or write access are connected to appropriate *RUN* components as well, in order to avoid dangling write or read ports of *VAR* components.



$$\mathcal{C} \cdot |[B \mid S]|$$

The handshake components stemming from B are collected into a *Run* term.

Definition 6.3

0. $\mathcal{C} \cdot |[B \mid S]| = Run \cdot (B \cap \mathbf{A}S) \cup \mathcal{C} \cdot S$

1. Let A be an alphabet structure. $Run \cdot A$ is the handshake circuit

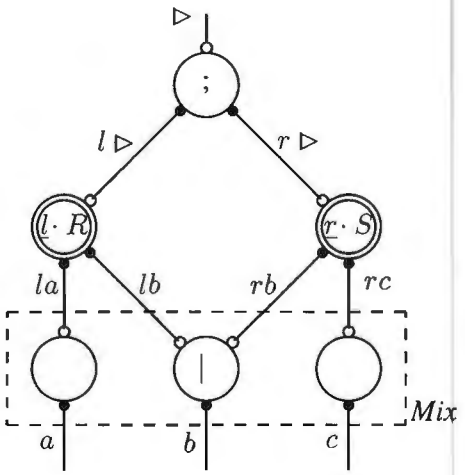
$$\begin{array}{rl} & \{x : x \in \mathbf{v}?A \cap \mathbf{v}!A \quad : VAR_{\tau_x} \cdot (wx^\circ, rx^\circ)\} \\ \cup & \{x : x \in \mathbf{v}?A \setminus \mathbf{v}!A \quad : RUN_{(!,\tau_x)} \cdot (wx^\circ)\} \\ \cup & \{x : x \in \mathbf{v}!A \setminus \mathbf{v}?A \quad : RUN_{(?,\tau_x)} \cdot (rx^\circ)\} \\ \cup & \{c : c \in \mathbf{p}?A \quad : RUN_{(!,\tau_c)} \cdot (c^\circ)\} \\ \cup & \{c : c \in \mathbf{p}!A \quad : RUN_{(?,\tau_c)} \cdot (c^\circ)\} \end{array}$$

□

Composite commands

Sequential composition

The translation of Tangram commands of the form $R; S$ is depicted by



$\mathcal{C} \cdot (R; S)$

It contains the handshake subcircuits $\underline{l} \cdot \mathcal{C} \cdot R$ and $\underline{r} \cdot \mathcal{C} \cdot S$ (in the picture we have used the commutation of renaming and application of $\mathcal{C}$).

The subcircuit contained in the dashed box is called a *Mix* term, and contains appropriate *MIX* components for ports common to R and S and *CON* components for other ports in R and S. These connectors are a byproduct of the renaming of *all* ports of R and S. The introduction of the connectors can be avoided, by using a more complex renaming scheme.

Definition 6.4

0. $$\mathcal{C} \cdot (R; S) = \{SEQ \cdot (\triangleright^\circ, l \triangleright^\bullet, r \triangleright^\bullet)\} \cup Mix \cdot (\mathbf{A}R, \mathbf{A}S) \cup \underline{l} \cdot \mathcal{C} \cdot R \cup \underline{r} \cdot \mathcal{C} \cdot S$$

1. Let A and B be conformant alphabet structures. $Mix \cdot (A, B)$ is the handshake circuit

$$Con_l \cdot (A \setminus B) \cup Con_r \cdot (B \setminus A) \cup \{c : c \in \mathbf{c}?A \cap \mathbf{c}?B : MIX_{(?,\tau_c)} \cdot (lc^\circ, rc^\circ, c^\bullet)\} \cup \{c : c \in \mathbf{c}!A \cap \mathbf{c}!B : MIX_{(!,\tau_c)} \cdot (lc^\circ, rc^\circ, c^\bullet)\}$$

2. Let A be an alphabet structure. $Con_l \cdot A$ is the handshake circuit

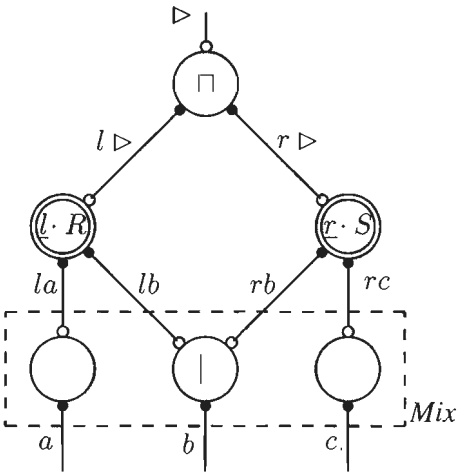
$$\begin{array}{rl} & \{c : c \in \mathbf{c}?A \quad : CON_{(?,\tau_c)} \cdot (lc^\circ, c^\bullet)\} \\ \cup & \{c : c \in \mathbf{c}!A \quad : CON_{(!,\tau_c)} \cdot (lc^\circ, c^\bullet)\} \end{array}$$

Similarly for $Con_r \cdot A$.

□

Nondeterministic choice

The translation for Tangram commands of the form $R \sqcap S$ closely resembles that of sequential commands. Since R and S are never activated concurrently, a *Mix* term takes care of the common ports.



$\mathcal{C} \cdot (R \sqcap S)$

Definition 6.5

$$\begin{array}{rl} \mathcal{C} \cdot (R \sqcap S) = & \{OR \cdot (\triangleright^\circ, l \triangleright^\bullet, r \triangleright^\bullet)\} \cup Mix \cdot (\mathbf{A}R, \mathbf{A}S) \\ \cup & \underline{l} \cdot \mathcal{C} \cdot R \cup \underline{r} \cdot \mathcal{C} \cdot S \end{array}$$

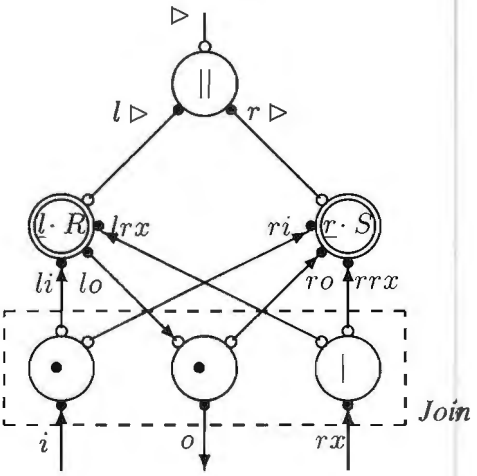
□

The *OR* component nondeterministically selects between the activation of the subcircuits $\underline{l} \cdot \mathcal{C} \cdot R$ and $\underline{r} \cdot \mathcal{C} \cdot S$. An alternative compilation rule, which reduces nondeterminism and avoids the costly *Mix*-term, is

$$\mathcal{C} \cdot (R \sqcap S) \quad = \quad \mathcal{C} \cdot R$$

Parallel composition

The compilation of commands of the form $R \parallel S$ is a little more complicated, because accesses to common ports and to common variables have to be treated differently. Communications through common ports have to be synchronized by *JOIN* components. Read access to common variables must be mixed by a *MIX* component. Recall that parallel commands do not have write access to common variables. In the circuit diagram below, i and o are common ports and rx provides read access to common variable x.



$\mathcal{C}\cdot(R \parallel S)$

The required "glue" for parallel composition is collected into a *Join* term.

Definition 6.6

0. $\mathcal{C}\cdot(R \parallel S) = \{PAR\cdot(\triangleright^\circ, l\triangleright^\bullet, r\triangleright^\bullet)\} \cup Join\cdot(\mathbf{A}R, \mathbf{A}S)$
$\cup\ \underline{l}\cdot\ \mathcal{C}\cdot R \cup \underline{r}\cdot\ \mathcal{C}\cdot S$

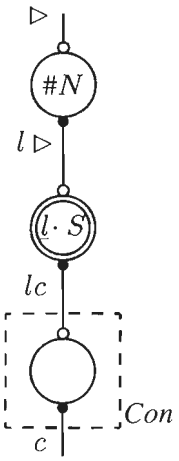
1. Let A and B be connectable alphabet structures. $Join\cdot(A, B)$ is the handshake circuit

$$
\begin{aligned}
& Con_l \cdot (A \setminus B) \cup Con_r \cdot (B \setminus A) \\
\cup\ & \{x : x \in \mathbf{v}?A \cap \mathbf{v}?B : MIX_{(?,\tau_c)}\cdot(lrx^\circ, rrx^\circ, rx^\bullet)\} \\
\cup\ & \{c : c \in \mathbf{p}?A \cap \mathbf{p}?B : JOIN_{(?,\tau_c)}\cdot(lc^\circ, rc^\circ, c^\bullet)\} \\
\cup\ & \{c : c \in \mathbf{p}!A \cap \mathbf{p}?B : JOIN_{(!,\tau_c)}\cdot(lc^\circ, rc^\circ, c^\bullet)\} \\
\cup\ & \{c : c \in \mathbf{p}?A \cap \mathbf{p}!B : JOIN_{(!,\tau_c)}\cdot(rc^\circ, lc^\circ, c^\bullet)\}
\end{aligned}
$$

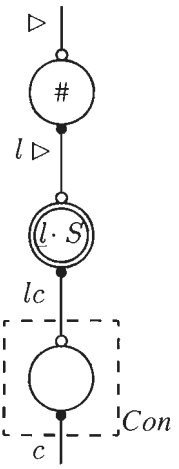
□

Repetition

The two forms of repetition in Tangram are simply included in the definition of $\mathcal{C}$. A *Con* term makes the renaming of the repeated command consistent with earlier commands.



$\mathcal{C} \cdot (\#N[S])$



$\mathcal{C} \cdot (\#[S])$

Definition 6.7

0. $\mathcal{C} \cdot (\#N[S]) = \{COUNT_N \cdot (\triangleright^\circ, l \triangleright^\bullet)\} \cup Con_l \cdot \mathbf{A}S \cup \underline{l} \cdot \mathcal{C} \cdot S$

1. $\mathcal{C} \cdot (\#[S]) = \{REP \cdot (\triangleright^\circ, l \triangleright^\bullet)\} \cup Con_l \cdot \mathbf{A}S \cup \underline{l} \cdot \mathcal{C} \cdot S$

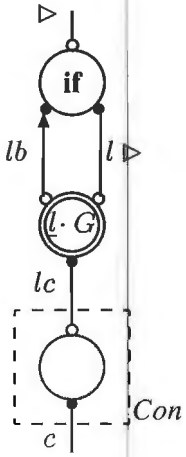
□

Guarded commands

Selection

Each selection or guarded repetition contains a set of guarded commands. Let G be a guarded-command set. Given $\mathcal{C} \cdot G$ the translation of a selection command is depicted by

$\mathcal{C}\cdot(\ \textbf{if}\ \ \textbf{fi}\ )$



$\mathcal{C}\cdot(\ \textbf{if}\ G\ \textbf{fi}\ )$

After activation through $\triangleright^\circ$, the **if** component collects the disjunction of the guards through $lb^\bullet$, as computed by $l\ \mathcal{C}\cdot G$. If this value is *true*, the subcircuit $l\ \mathcal{C}\cdot G$ is activated through $l\ \triangleright$; if *false* the subsequent behavior of circuit is left unspecified.

Definition 6.8

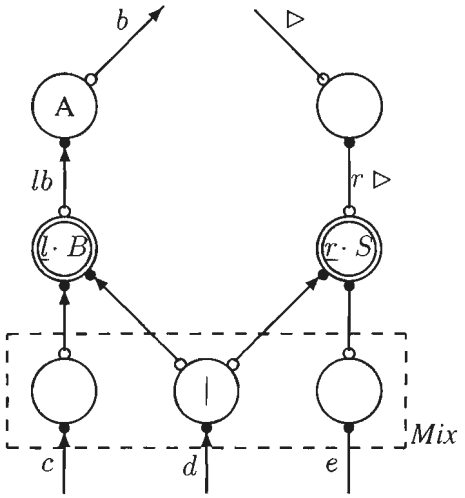
0. $\mathcal{C}\cdot(\ \textbf{if}\ \ \textbf{fi}\ ) = \{STOP\cdot(\triangleright^\circ)\}$
1. $\mathcal{C}\cdot(\ \textbf{if}\ G\ \textbf{fi}\ ) = \{IF\cdot(\triangleright^\circ, lb^\bullet, l\triangleright^\bullet)\} \cup Con_l\cdot \mathbf{A}G \cup \underline{l}\cdot\ \mathcal{C}\cdot G$

□

Note that the inclusion in Tangram's guarded commands of a default guard "otherwise" can be implemented straightforwardly by modifying the **if** component. An additional "otherwise" activation port is selected for handshaking if the value *false* is received through b.

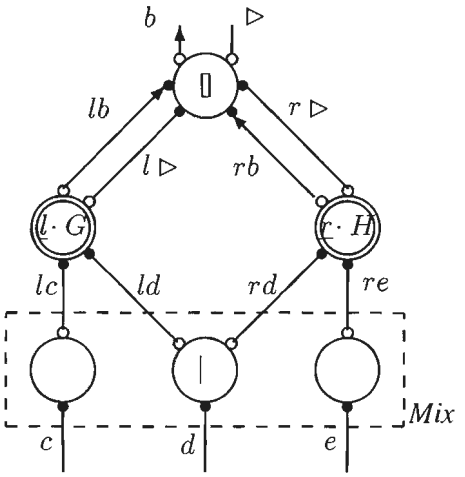
Guarded-command set

A singleton guarded-command set has the form $B \rightarrow S$. The corresponding handshake circuit consists of subcircuits $\underline{l}\cdot\ \mathcal{C}\cdot B$ and $\underline{l}\cdot\ \mathcal{C}\cdot S$. The guard B and the command S may have read-access to common variables. Hence, renaming and a *Mix* term are required. The connector connected to $\triangleright^\circ$ is also a consequence of renaming. Tangram does not restrict guards to Boolean expressions. Hence, an adapter is required to guarantee that only Boolean values are passed along b.



$\mathcal{C} \cdot (B \rightarrow S)$

A guarded-command set with at least two elements can be decomposed into two nonempty guarded-command sets. In Tangram such a set is denoted by connecting the component sets with a ' [] '. The circuit $\mathcal{C} \cdot (G \,[]\, H)$ contains the subcircuits $\underline{l} \cdot \mathcal{C} \cdot G$ and $\underline{r} \cdot \mathcal{C} \cdot H$. A *BAR* component implements the Tangram ' [] '. Common ports and variables of guarded-command sets G and H are accessed through a *Mix* term.



$\mathcal{C} \cdot (G \,[]\, H)$

The two-phase operation of the circuit $\mathcal{C}\cdot(G \,[]\, H)$ can be understood as follows. Firstly, the environment requests a value through b°. This request is passed on to the subcircuits $\underline{l}\cdot\ \mathcal{C}\cdot G$ and $\underline{r}\cdot\ \mathcal{C}\cdot H$, and the disjunction of the Booleans that arrive through lb and rb is then output through b°.

If this value is *true*, the circuit is ready for the second phase, which starts with $\triangleright_0^\circ$. The *BAR* component nondeterministically selects either $l\,\triangleright^\bullet$ or $r\,\triangleright^\bullet$, *provided* that a *true* value arrived through the corresponding Boolean port in the first phase. After termination of the selected guarded-command subset, the second phase is completed with $\triangleright_1^\circ$.

Definition 6.9

0. $\mathcal{C}\cdot(B \rightarrow S) = \{ADAPT_{(\mathbf{bool},\tau_B)}\cdot(b^\circ, lb^\bullet)\} \cup \mathcal{C}\cdot B \cup CON\cdot(\triangleright^\circ, r\,\triangleright^\bullet) \cup \mathcal{C}\cdot S \cup Mix\cdot(\mathbf{v}B, \mathbf{v}S)$

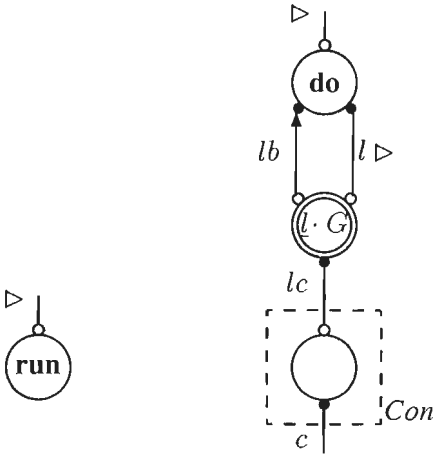
1. $\mathcal{C}\cdot(G \,[]\, H) = \{BAR\cdot(b^\circ, \triangleright^\circ, lb^\bullet, l\,\triangleright^\bullet, lb^\bullet, r\,\triangleright^\bullet)\} \cup Mix\cdot(\mathbf{A}G, \mathbf{A}H) \cup \underline{l}\cdot\ \mathcal{C}\cdot G \cup \underline{r}\cdot\ \mathcal{C}\cdot H$

□

Nondeterminism can be reduced by making the *BAR* component more deterministic. For instance, if both incoming Booleans are *true*, the component may favor $l\,\triangleright^\bullet$ after reactivation through $\triangleright^\circ$.

Guarded repetition

The handshake circuit for a guarded repetition closely resembles that of a selection command. The behavior of **do** **od** equals that of **skip**, which is simply implemented by a *RUN* component. If there is at least one guarded command, the resulting handshake circuit becomes:



$\mathcal{C}\cdot(\ \textbf{do}\ \ \textbf{od}\ )$ $\mathcal{C}\cdot(\ \textbf{do}\ G\ \textbf{od}\ )$

After activation through $\triangleright^\circ$, the **do** component inputs a Boolean through $lb^\bullet$ and, if *true*, handshakes through $l\triangleright^\bullet$. This is repeated until *false* arrives. Then the **do** component returns in its initial state after a $\triangleright^\circ_!$.

Definition 6.10

0. $\mathcal{C}\cdot(\ \textbf{do}\ \ \textbf{od}\ ) = \{RUN\cdot(\triangleright^\circ)\}$

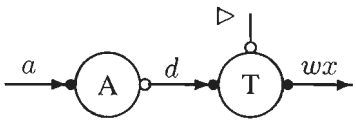
1. $\mathcal{C}\cdot(\ \textbf{do}\ G\ \textbf{od}\ ) = \{DO\cdot(\triangleright^\circ, lb^\bullet, l\triangleright^\bullet)\} \cup Con_l \cdot \mathbf{A}G \cup \underline{l}\cdot\ \mathcal{C}\cdot G$

□

Primitive commands

Input

The circuit of $\mathcal{C}\cdot(a?x)$ is depicted by



$\mathcal{C}\cdot(a?x)$

The adapter takes care of a possible mismatch between the types of a and x.

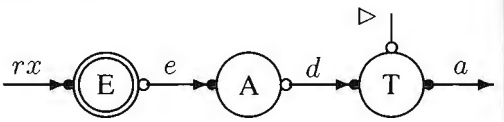
Definition 6.11

$\mathcal{C} \cdot (a?x) = \{TRF_{\tau_x} \cdot (\triangleright^\circ, d^\bullet, wx^\bullet)\ ,\ ADAPT_{(\tau_x,\tau_a)} \cdot (d^\circ, a^\bullet)\}$

□

Output

The circuit of $\mathcal{C} \cdot (a!E)$ contains the subcircuit $\mathcal{C} \cdot E$. Again, an adapter is introduced to resolve possible type mismatches. Port $rx^\bullet$ provides read access to variable x.



$\mathcal{C} \cdot (a!E)$

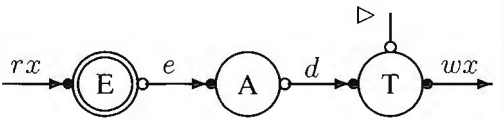
Definition 6.12

$\mathcal{C} \cdot (a!E) = \{TRF_{\tau_a} \cdot (\triangleright^\circ, d^\bullet, a^\bullet)\ ,\ ADAPT_{(\tau_a,\tau_E)} \cdot (d^\circ, e^\bullet)\} \cup \mathcal{C} \cdot E$

□

Assignment

The translation of the assignment is very similar to that of the output command.



$\mathcal{C} \cdot (x := E)$

Definition 6.13

$\mathcal{C} \cdot (x := E) = \{TRF_{\tau_x} \cdot (\triangleright^\circ, d^\bullet, wx^\bullet)\ ,\ ADAPT_{(\tau_x,\tau_E)} \cdot (d^\circ, e^\bullet)\} \cup \mathcal{C} \cdot E$

□

Synchronization

The synchronization command a is implemented by connecting the activation port $\triangleright^\circ$ to port $a^\bullet$.

$\mathcal{C} \cdot a$

Definition 6.14

$\mathcal{C} \cdot a = \{CON \cdot (\triangleright^\circ, a^\bullet)\}$

□

Skip and stop

The translations of **skip** and **stop** are self-evident.

$\mathcal{C} \cdot$ **skip**

$\mathcal{C} \cdot$ **stop**

Definition 6.15

0. $\mathcal{C} \cdot$ **skip** $= \{RUN \cdot (\triangleright^\circ)\}$
1. $\mathcal{C} \cdot$ **stop** $= \{STOP \cdot (\triangleright^\circ)\}$

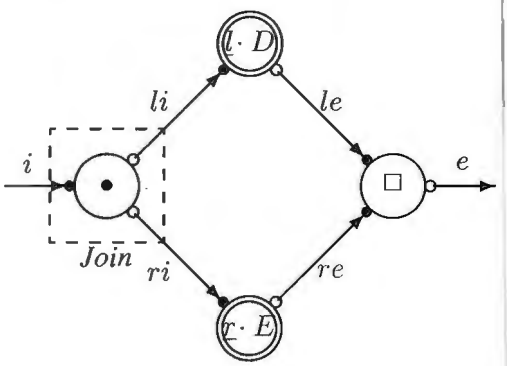
□

Expressions

Expressions form the last syntactic category left to consider for compilation to handshake circuits.

Binary operators

The circuit $\mathcal{C}\cdot(D\Box E)$ contains the subcircuits $\mathcal{C}\cdot D$ and $\mathcal{C}\cdot E$, appropriately renamed. If D and E both refer to the same variable, say x, a *JOIN* component can be used to combine the read accesses to x. A *Join* term is then needed to accommodate for an overlap $\mathbf{v}?D$ and $\mathbf{v}?E$.



$\mathcal{C}\cdot(D\Box E)$

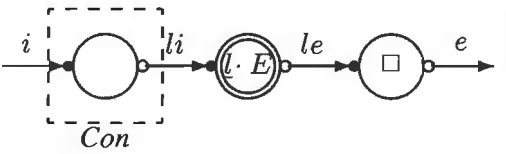
Definition 6.16

$$\mathcal{C}\cdot(D\Box E) = \{BIN_{(\Box,\tau_D,\tau_E)}\cdot(e^\circ, le^\bullet, re^\bullet)\} \cup Join\cdot(\mathbf{A}D, \mathbf{A}E)$$
$$\Box \qquad \cup \ \underline{l}\cdot\ \mathcal{C}\cdot D \cup \underline{r}\cdot\ \mathcal{C}\cdot E$$

Note that a *Mix* term would do the job as well. Whereas a *Join* term enforces synchronization of accesses to common variables, the *Mix* term enforces sequentialization. Cost and performance considerations favor the *Join* term in most cases.

Unary operators

Expressions of the form $\Box E$ are translated similarly.



$\mathcal{C}\cdot(\Box E)$

The *Join* term reduces to a *Con* term.

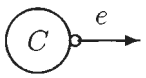
Definition 6.17

$\mathcal{C} \cdot (\square E) = \{UN_{(\square, \tau_E)} \cdot (e^\circ, le^\bullet)\} \cup Con_l \cdot \mathbf{A}E \cup \underline{l} \cdot \mathcal{C} \cdot E$

□

Constants

The handshake circuit for a constant expression is self-evident.



$\mathcal{C} \cdot C$

Definition 6.18

$\mathcal{C} \cdot C = \{CST \cdot_C (e^\circ)\}$

□

Variables

The expression x translates to a simple connector. Recall that the declaration of variable x yields a *VAR* component with read port rx°.

$\mathcal{C} \cdot x$

Definition 6.19

$\mathcal{C} \cdot x = \{CON_{(?, \tau_x)} \cdot (e^\circ, rx^\bullet)\}$

□

This concludes the translation of Tangram commands into handshake circuits. Examples of compiled handshake circuits can be found in Chapter 1. The circuits of Figures 1.1, 1.2, 1.3, 1.9, and 1.11 can be obtained by applying $\mathcal{C}$ to the corresponding Tangram programs or commands. The circuits of Figures 1.4 and

1.5 can be obtained by compilation and subsequent minor optimizations at the handshake-circuit level (cf. Chapter 8). The circuits of Figure 1.12 require non-trivial extensions of $\mathcal{C}$.

6.2 Compilation theorem

In this section we analyze the most important property of the $\mathcal{C}$ function, specifically that it yields handshake circuits that are *equivalent* to the corresponding Tangram programs in a precise sense. This analysis is restricted to Core Tangram. Recall that in Core Tangram all involved alphabet structures are of the form $\langle \mathbf{p}?A, \emptyset, \emptyset, \emptyset, \{\tilde{}\}\rangle$. For convenience we shall write $\mathbf{a}S$ to denote $\mathbf{p}?\mathbf{A}S$, where S is a Core Tangram command. Note that $\mathbf{a}S$ is a set of names.

Below the compilation function for Core Tangram is presented in a self-contained form. The function $\mathcal{C}$ is consistent with the more general compilation function of the previous section.

Definition 6.20

The compilation function $\mathcal{C}$ for Core Tangram is defined by:

0. $\mathcal{C}\cdot\mathbf{skip} = \{RUN\cdot(\triangleright^\circ)\}$

1. $\mathcal{C}\cdot\mathbf{stop} = \{STOP\cdot(\triangleright^\circ)\}$

2. $\mathcal{C}\cdot a = \{CON\cdot(\triangleright^\circ, a^\bullet)\}$

3. $\mathcal{C}\cdot(R;S) = \{SEQ\cdot(\triangleright^\circ, l\triangleright^\bullet, r\triangleright^\bullet)\} \cup Mix\cdot(\mathbf{a}R, \mathbf{a}S) \cup \underline{l}\cdot\mathcal{C}\cdot R \cup \underline{r}\cdot\mathcal{C}\cdot S$

4. $\mathcal{C}\cdot(R \sqcap S) = \{OR\cdot(\triangleright^\circ, l\triangleright^\bullet, r\triangleright^\bullet)\} \cup Mix\cdot(\mathbf{a}R, \mathbf{a}S) \cup \underline{l}\cdot\mathcal{C}\cdot R \cup \underline{r}\cdot\mathcal{C}\cdot S$

5. $\mathcal{C}\cdot(\#N[S]) = \{COUNT_N\cdot(\triangleright^\circ, l\triangleright^\bullet)\} \cup Con_l\cdot\mathbf{a}S \cup \underline{l}\cdot\mathcal{C}\cdot S$

6. $\mathcal{C}\cdot(\#[S]) = \{REP\cdot(\triangleright^\circ, l\triangleright^\bullet)\} \cup Con_l\cdot\mathbf{a}S \cup \underline{l}\cdot\mathcal{C}\cdot S$

7. $\mathcal{C}\cdot(R \parallel S) = \{PAR\cdot(\triangleright^\circ, l\triangleright^\bullet, r\triangleright^\bullet)\} \cup Join\cdot(\mathbf{p}R, \mathbf{p}S) \cup \underline{l}\cdot\mathcal{C}\cdot R \cup \underline{r}\cdot\mathcal{C}\cdot S$

8. $\mathcal{C}\cdot(|[B \mid S]|) = Run\cdot(B \cap \mathbf{a}S) \cup \mathcal{C}\cdot S$

9. $\mathcal{C}\cdot((B)\cdot S) = Stop\cdot(B \setminus \mathbf{a}S) \cup \mathcal{C}\cdot S$

where (in alphabetic order):

10. $CON\cdot(a^\circ, b^\bullet) = \#[a^\circ : b^\bullet]$
11. $COUNT_N\cdot(a^\circ, b^\bullet) = \#[a^\circ : \#N[b^\bullet]]$
12. $JOIN\cdot(a^\circ, b^\circ, c^\bullet) = \#[a^\circ : b^\circ : c^\bullet]$
13. $MIX\cdot(a^\circ, b^\circ, c^\bullet) = \#[[a^\circ : c^\bullet \mid b^\circ : c^\bullet]]$
14. $OR\cdot(a^\circ, b^\bullet, c^\bullet) = \#[a^\circ : (b^\bullet \sqcap c^\bullet)]$
15. $PAR\cdot(a^\circ, b^\bullet, c^\bullet) = \#[a^\circ : (b^\bullet \parallel c^\bullet)]$
16. $REP\cdot(a^\circ, b^\bullet) = (a^\circ : \#[b^\bullet])$
17. $RUN\cdot(a^\circ) = \#[a^\circ]$
18. $SEQ\cdot(a^\circ, b^\bullet, c^\bullet) = \#[a^\circ : (b^\bullet; c^\bullet)]$
19. $STOP\cdot(a^\circ) = (a^\circ)\cdot\mathbf{stop}$

and:

20. $Run\cdot A = \{a : a \in A : RUN\cdot(a^\circ)\}$
21. $Stop\cdot A = \{a : a \in A : STOP\cdot(a^\circ)\}$
22. $Con_l\cdot A = \{a : a \in A : CON\cdot(la^\circ, a^\bullet)\}$
23. $Mix\cdot(A, B) = Con_l\cdot(A \setminus B) \cup Con_r\cdot(B \setminus A) \cup \{a : a \in A \cap B : MIX\cdot(la^\circ, ra^\circ, a^\bullet)\}$
24. $Join\cdot(A, B) = Con_l\cdot(A \setminus B) \cup Con_r\cdot(B \setminus A) \cup \{a : a \in A \cap B : JOIN\cdot(la^\circ, ra^\circ, a^\bullet)\}$

□

One of the central theorems of this thesis is the compilation theorem.

Theorem 6.21 (compilation theorem)

Let T be a Core Tangram program. Then

$$\triangleright^*\cdot\mathcal{H}\cdot T = \parallel\cdot\mathcal{C}\cdot T$$

Proof by structural induction over Core Tangram later in this section.

□

The proof of the compilation theorem requires a little more ground work to be done.

Separation properties

The presented syntax-directed translation method is one of recursive decomposition. The circuit for a composite command is decomposed into circuits for the subcommands and some additional circuitry that is specific for the command operator and for the port alphabets of the subcommands. One could say that this additional circuitry is *separated* from the circuits of the subcommands during such a decomposition step. A systematic analysis of this form of separation is studied next.

The formulation of separation properties is based on so-called σ-functions. A σ-function is a function from sequential processes to sequential processes of a restricted form.

Definition 6.22 **(σ-function)**

Let a be a name, N a natural number, B a port structure, and P a sequential handshake process.

0. Let X be a sequential handshake process. Then the following expressions define σ-functions:
   (a) $(B)\cdot X$
   (b) $P \sqcap X$
   (c) $X; P$
   (d) $P; X$
   (e) $\#[X]$
   (f) $\#N[X]$
   (g) $P \,||\, X$
   (h) $a^\circ : X$

   Furthermore, a composition of σ-functions is also a σ-function.

1. σ-function F is said to *interfere* with port structure A if there is a port a, $a \in A$, for which $F \cdot a^\circ$ or $F \cdot a^\bullet$ is undefined.

2. σ-function F is *permanent* if its image consists exclusively of permanent sequential processes.

□

Function $\triangleright^*$ is an example of a σ-function. In general, a σ-function is partial. For instance, function F, $F \cdot X = X; P$, is defined only if X is conformant with P. The definition of σ-functions can be extended to include the complete handshake calculus. Such extensions are not relevant to our current purposes.

Property 6.23 (separation)

0. *Command separation.* Let P be a sequential process and let F be a permanent σ-function, such that F does *not* interfere with $\mathbf{p}P$ or a°. Then

$$F \cdot P = F \cdot a^\bullet \;||\; \#[a^\circ : P]$$

The non-interference requirement on F guarantees that $F \cdot a^\bullet$ and $\#[a^\circ : P]$ are connectable. This separation property is similar in intent to the "decomposition rule" of [Mar89].

1. *Con separation.* Let P be a sequential process such that $\mathbf{p}^\circ P = \emptyset$, and let $a \in \mathbf{p}^\bullet P$. Furthermore, let F be a σ-function, such that F does not interfere with $\mathbf{p}P$ or la. Then

$$F \cdot P = F \cdot (a := la) \cdot P \;||\; CON \cdot (la^\circ, a^\bullet)$$

where $(a := la) \cdot P$ denotes the sequential process P with all occurrences of symbols of port a l-renamed. Consequently,

$$F \cdot P = F \cdot \underline{l} \cdot P \;||\; Con_l \cdot \mathbf{p}P$$

Again, non-interference of F with $\mathbf{p}P$ or la assures connectability.

2. *Mix separation.* Let P and Q be conformant sequential processes, with $\mathbf{p}^\circ P = \mathbf{p}^\circ Q = \emptyset$ and let $a \in \mathbf{p}^\bullet P \cap \mathbf{p}^\bullet Q$. Furthermore, let F be a σ-function, such that F does not interfere with $\mathbf{p}P$, $\mathbf{p}Q$, la, or ra (to assure connectability in the decomposition below). Then

$$F \cdot (P; Q) = F \cdot ((a := la) \cdot P; (a := ra) \cdot Q) \;||\; MIX \cdot (la^\circ, ra^\circ, a^\bullet)$$

Consequently,

$$F \cdot (P; Q) = F \cdot (\underline{l} \cdot P; \underline{r} \cdot Q) \;||\; Mix \cdot (\mathbf{p}P, \mathbf{p}Q)$$

3. A similar *Mix* separation exist for $F \cdot (P \sqcap Q)$.

4. *Join separation.* Let P and Q be sequential processes with $\mathbf{p}^\circ P = \mathbf{p}^\circ Q = \emptyset$ and let $a \in \mathbf{p}^\bullet P \cap \mathbf{p}^\bullet Q$. Furthermore, let F be a σ-function, such that F does not interfere with $\mathbf{p}P$, $\mathbf{p}Q$, la or ra (to assure connectability in the decomposition below). Then

$$F \cdot (P \overset{\bullet\bullet}{\|} Q) = F \cdot ((a := la) \cdot P \overset{\bullet\bullet}{\|} (a := ra) \cdot Q) \;\|\; JOIN \cdot (la^\circ, ra^\circ, a^\bullet)$$

Consequently,

$$F \cdot (P \overset{\bullet\bullet}{\|} Q) = F \cdot (\underline{l} \cdot P \overset{\bullet\bullet}{\|} \underline{r} \cdot Q) \;\|\; Join \cdot (\mathbf{p}P, \mathbf{p}Q)$$

Since $\underline{l} \cdot P$ and $\underline{r} \cdot Q$ are obviously connectable, the $\overset{\bullet\bullet}{\|}$ in the last process expression may be replaced by $\|$.

□

The separation properties can be used to prove most of the parallel compositions in Example 3.18.

Example 6.24

Consider σ-function F defined by $F \cdot X = \#[a^\circ : \#2[X]]$. Clearly, a handshake through a° has the effect of executing X twice. Note that $F \cdot b^\bullet = DUP \cdot (a^\circ, b^\bullet)$, the duplicator of Example 2.23. Using the above separation properties, we obtain the following decomposition of $F \cdot P$:

$$\begin{aligned}
& F \cdot P \\
= \quad & \{ \text{ Separation property 0 } \} \\
& F \cdot b^\bullet \;\|\; \#[b^\circ : P] \\
= \quad & \{ \text{ Definition of } F \} \\
& \#[a^\circ : \#2[b^\bullet]] \;\|\; \#[b^\circ : P] \\
= \quad & \{ \text{ Separation property 2; Definition of } SEQ \} \\
& SEQ \cdot (a^\circ, lb^\bullet, rb^\bullet) \;\|\; MIX \cdot (lb^\circ, rb^\circ, b^\bullet) \;\|\; \#[b^\circ : P]
\end{aligned}$$

□

Proof of the compilation theorem

The proof of the compilation theorem is by structural induction over Core Tangram, and follows the command order of Definition 6.20. The proofs for the three primitive commands **skip**, **stop** and $a^\bullet$ are skipped. The syntactic category *program* is not treated separately, since a program is a just an extension command.

Most proof cases refer to one or more separation properties. Of course, it is then a part of the proof obligations to verify that a *proper* σ-function is involved. In particular, the non-interference of the σ-function with its argument or with the freshly introduced ports must be checked. From the simplicity of the applied renaming scheme, non-interference can be easily established, and hence the connectability of the sub-circuits introduced by the separation step is guaranteed.

Case command $S;T$:

$$
\begin{aligned}
& \rhd^* \cdot \mathcal{H} \cdot (S;T) \\
= \quad & \{\ \mathcal{H} \text{ distributes over } ;\ \} \\
& \rhd^* \cdot (\mathcal{H} \cdot S\ ;\ \mathcal{H} \cdot T) \\
= \quad & \{\ \mathit{Mix} \text{ separation } \} \\
& \rhd^* \cdot (\underline{l} \cdot \mathcal{H} \cdot S\ ;\ \underline{r} \cdot \mathcal{H} \cdot T)\ ||\ \mathit{Mix} \cdot (\mathbf{a}S, \mathbf{a}T) \\
= \quad & \{\ \text{command separation (twice) } \} \\
& \rhd^* \cdot (l \rhd^\bullet; r \rhd^\bullet)\ ||\ \underline{l} \cdot \rhd^* \cdot \mathcal{H} \cdot S\ ||\ \underline{r} \cdot \rhd^* \cdot \mathcal{H} \cdot T\ ||\ \mathit{Mix} \cdot (\mathbf{a}S, \mathbf{a}T) \\
= \quad & \{\ \text{induction hypothesis } \} \\
& \rhd^* \cdot (l \rhd^\bullet; r \rhd^\bullet)\ ||\ (\underline{l}\ ||\cdot\ \mathcal{C} \cdot S)\ ||\ (\underline{r}\ ||\cdot\ \mathcal{C} \cdot T)\ ||\ \mathit{Mix} \cdot (\mathbf{a}S, \mathbf{a}T) \\
= \quad & \{\ \text{rewrite; definition of } \mathit{SEQ} \ \} \\
& ||\cdot(\{\mathit{SEQ} \cdot (\rhd^\circ, l \rhd^\bullet, r \rhd^\bullet)\} \cup \mathit{Mix} \cdot (\mathbf{a}R, \mathbf{a}S) \cup \underline{l} \cdot \mathcal{C} \cdot S \cup \underline{r} \cdot \mathcal{C} \cdot T) \\
= \quad & \{\ \text{definition } \mathcal{C} \ \} \\
& ||\cdot\ \mathcal{C} \cdot (S;T)
\end{aligned}
$$

Case command $S \sqcap T$: similar to command $S;T$.

Case command #[S] :

$$
\begin{aligned}
& \rhd^* \cdot \mathcal{H} \cdot (\#[S])
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\ \mathcal{H} \text{ commutes with } \#\ \} \\
&\quad \rhd^* \cdot \#[\mathcal{H} \cdot S] \\
&= \quad \{\ \mathit{Con} \text{ separation}\ \} \\
&\quad \rhd^* \cdot \#[\underline{l} \cdot \mathcal{H} \cdot S] \ ||\ \mathit{Con}_l \cdot \mathbf{a}S \\
&= \quad \{\ \text{command separation}\ \} \\
&\quad \rhd^* \cdot \#[l \rhd^\bullet] \ ||\ \underline{l} \cdot \rhd^* \cdot \mathcal{H} \cdot S \ ||\ \mathit{Con}_l \cdot \mathbf{a}S \\
&= \quad \{\ \text{induction hypothesis}\ \} \\
&\quad \rhd^* \cdot \#[l \rhd^\bullet] \ ||\ (\underline{l} \ ||\cdot\, \mathcal{C} \cdot S) \ ||\ \mathit{Con}_l \cdot \mathbf{a}S \\
&= \quad \{\ \text{rewrite; definition of } \mathit{REP}\ \} \\
&\quad ||\cdot(\{\mathit{REP} \cdot (\rhd^\circ, l \rhd^\bullet)\} \cup \mathit{Con}_l \cdot \mathbf{a}R \cup \underline{l} \cdot \mathcal{C} \cdot S) \\
&= \quad \{\ \text{definition } \mathcal{C}\ \} \\
&\quad ||\cdot\, \mathcal{C} \cdot (\#[S])
\end{aligned}
$$

Case command $R = \#N[S]$: similar to command $\#[S]$.

Case command $S \,||\, T$:

$$
\begin{aligned}
&\quad \rhd^* \cdot \mathcal{H} \cdot (S \,||\, T) \\
&= \quad \{\ \mathcal{H} \text{ distributes over } ||\ \} \\
&\quad \rhd^* \cdot (\mathcal{H} \cdot S \ \overset{\bullet\bullet}{||}\ \mathcal{H} \cdot T) \\
&= \quad \{\ \mathit{Join} \text{ separation}\ \} \\
&\quad \rhd^* \cdot (\underline{l} \cdot \mathcal{H} \cdot S \ ||\ \underline{r} \cdot \mathcal{H} \cdot T) \ ||\ \mathit{Join} \cdot (\mathbf{a}S, \mathbf{a}T) \\
&= \quad \{\ \text{command separation (twice)}\ \} \\
&\quad \rhd^* \cdot (l \rhd^\bullet \,||\, r \rhd^\bullet) \ ||\ \underline{l} \cdot \rhd^* \cdot \mathcal{H} \cdot S \ ||\ \underline{r} \cdot \rhd^* \cdot \mathcal{H} \cdot T \ ||\ \mathit{Join} \cdot (\mathbf{a}S, \mathbf{a}T) \\
&= \quad \{\ \text{induction hypothesis}\ \} \\
&\quad \rhd^* \cdot (l \rhd^\bullet \,||\, r \rhd^\bullet) \ ||\ (\underline{l} \ ||\cdot\, \mathcal{C} \cdot S) \ ||\ (\underline{r} \ ||\cdot\, \mathcal{C} \cdot T) \ ||\ \mathit{Join} \cdot (\mathbf{a}S, \mathbf{a}T) \\
&= \quad \{\ \text{rewrite; definition of } \mathit{PAR}\ \} \\
&\quad ||\cdot(\{\mathit{PAR} \cdot (\rhd^\circ, l \rhd^\bullet, r \rhd^\bullet)\} \cup \mathit{Join} \cdot (\mathbf{a}R, \mathbf{a}S) \cup \underline{l} \cdot \mathcal{C} \cdot S \cup \underline{r} \cdot \mathcal{C} \cdot T) \\
&= \quad \{\ \text{definition } \mathcal{C}\ \} \\
&\quad ||\cdot\, \mathcal{C} \cdot (S \,||\, T)
\end{aligned}
$$

Case command $|[B \mid S]|$:

$\triangleright^* \cdot \mathcal{H} \cdot |[B \mid S]|$

$=$ { property of concealment }

$\triangleright^* \cdot \mathcal{H} \cdot |[B \cap \mathbf{a}S \mid S]|$

$=$ { $\mathcal{H}$ "distributes over" concealment }

$\triangleright^* \cdot |[\mathcal{H} \cdot (B \cap \mathbf{a}S) \mid \mathcal{H} \cdot S]|$

$=$ { concealment commutes with enclosure and repetition }

$|[\mathcal{H} \cdot (B \cap \mathbf{a}S) \mid \triangleright^* \cdot \mathcal{H} \cdot S]|$

$=$ { property 4.20 of non-terminating processes }

$\triangleright^* \cdot \mathcal{H} \cdot S \;||\; RUN \cdot \overline{\mathcal{H} \cdot (B \cap \mathbf{a}S)}$

$=$ { induction hypothesis }

$||\cdot \mathcal{C} \cdot S \;||\; RUN \cdot \overline{\mathcal{H} \cdot (B \cap \mathbf{a}S)}$

$=$ { rewrite; definition of *Run* }

$||\cdot(\, \mathcal{C} \cdot S \;\cup\; Run \cdot \overline{\mathcal{H} \cdot (B \cap \mathbf{a}S)})$

$=$ { definition $\mathcal{C}$ }

$||\cdot \mathcal{C} \cdot |[B \mid S]|$

Case command $(B) \cdot S$: similar to command $|[B \mid S]|$.

□

Chapter 7

Handshake circuits → VLSI circuits

7.0 Introduction

Handshake circuits are proposed as an intermediary between communicating processes (Tangram programs) and VLSI circuits. Chapter 6 describes the translation of Tangram programs into handshake circuits. This chapter is concerned with the realization of handshake circuits as *efficient* and *testable* VLSI circuits. First we observe that the fine-grained parallelism available in VLSI circuits matches the fine-grained concurrency in handshake circuits nicely. The mapping of handshake circuits to VLSI circuits can therefore be relatively direct.

A rather naive mapping is suggested by the following correspondence:

0. a channel corresponds to a set of wires, one per symbol;
1. an event with name a corresponds to a voltage transition along wire a;
2. each handshake component corresponds to a VLSI circuit that satisfies the specification at the transition level.

There is no doubt that the above mapping can result in functional circuits. In general, however, the resulting circuits will be prohibitive in size, poor in performance, probably hard to initialize, and impractical to test for fabrication faults. Concerns for circuit size, performance, initialization and testability will therefore be recurring themes in this chapter.

A full treatment of all relevant VLSI-realization issues is beyond the scope of this thesis. Issues that directly relate to (properties of) handshake circuits have

been selected for a relatively precise treatment; other topics are sketched more briefly. This chapter discusses:

- *peephole* optimization: the substitution of subcircuits by cheaper ones;
- relaxation of the receptiveness requirement of handshake processes;
- handshake signaling between handshake components;
- decomposition into VLSI operators and (isochronic) forks;
- initialization of the resulting VLSI circuits;
- fabrication testing of the VLSI circuit.

7.1 Peephole optimization

An obvious method of optimization of handshake circuits is substitution of subcircuits by cheaper subcircuits on the basis of "equals for equals". Example 3.18 lists a number of pairs of circuits with equal behavior. Substitution of one member of the pair by the other will not affect the functional behavior of the circuit. However, for given VLSI realizations of the handshake components, such a substitution will affect the circuit's cost and performance. At this point we shall not delve into cost models or metrics. In the examples below the advantage(s) of substitution in one way or the other will be hinted upon only.

Example 7.0

The following optimizations are the most interesting from a practical viewpoint:

0. Removal of connectors as suggested by Examples 3.18.0 and 3.18.1 has only advantages. However, since connectors can be realized by wires only, the expected advantages will evaporate during the layout phase.

1. Example 3.18.2 is a useful one. It allows the elimination of most *RUN* components from compiled handshake circuits. Since the proposed compilation function introduces a *RUN* for each internal channel in a Tangram program, this optimization yields interesting savings.

2. Example 3.24 discusses trees of *MIX*, *SEQ*, *PAR*, and *OR* components. Balancing of such trees does not affect the cost of a circuit, but generally improves the (average) performance. The same holds for trees of *BAR* components (cf. Example 4.37.18).

□

In many instances a subcircuit may be replaced by a cheaper subcircuit with an "almost equal" behavior. Such substitutions are allowed if their effect on the external behavior of the circuit cannot be observed by any possible environment. This form of optimization will be called *refinement in context.*

Definition 7.1 (refinement in context)

Let P, Q and R be handshake processes, such that $\mathbf{p}P = \mathbf{p}Q$ and $\mathbf{p}P \bowtie \mathbf{p}R$. Process P *refines to* Q *in the context of* R, denoted by $P \sqsubseteq_R Q$, if

$$P \parallel R \quad \sqsubseteq \quad Q \parallel R$$

□

The following properties are given without proof.

Property 7.2

0. $P \sqsubseteq_R P$
1. $P \sqsubseteq_R Q \ \wedge \ Q \sqsubseteq_R T \ \Rightarrow P \sqsubseteq_R T$ (Hence, $\sqsubseteq_R$ is a preorder.)
2. $P \sqsubseteq_{CHAOS_\emptyset} Q \ \equiv \ P \sqsubseteq Q$
3. If $\mathbf{p}R = \overline{\mathbf{p}P}$, the defining expression of refinement in context can be rewritten as

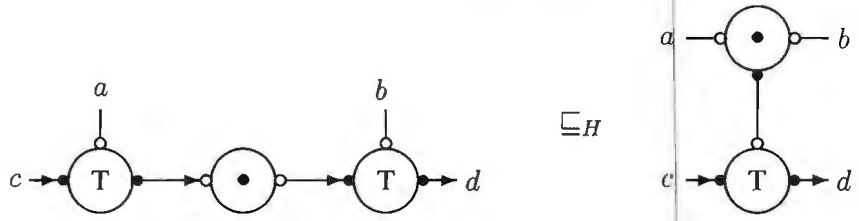
$$\mathbf{t}P \cap \mathbf{t}R \ \supseteq \ \mathbf{t}Q \cap \mathbf{t}R$$

□

Example 7.3

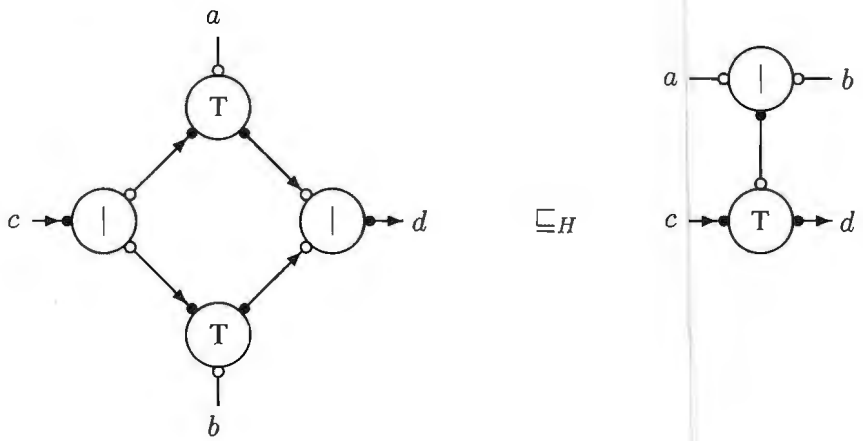
Three examples with practical interest are given below. We assume that the context H is a handshake circuit, such that for refinement $P \sqsubseteq_H Q$ the circuit $P \cup H$ can be obtained by compilation of a Tangram program. Several of these substitutions may be applied in succession.

0. The following refinement in context has been applied to the circuit of Figure 1.3 in order to obtain that of Figure 1.4.



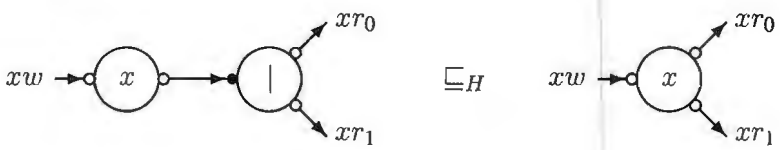
A difference between the two circuits is that trace a_0 is quiescent in the circuit of the right-hand side and not in the other one.

1. The following optimization has been applied after the translation of the wagging buffer of Section 1.1 in order to obtain the handshake circuit of Figure 1.5.



The substitution relies on the mutual exclusion of handshakes through ports a and b. Overlap of these handshakes may cause message overtaking via the parallel transfer paths.

2. Read access to a common variable in a Tangram program results in a tree of *MIX* components connected to the read port of the variable. This form of the *MIX* component turns out to be rather expensive. A variable with multiple read ports that can be served in parallel is a cheaper and faster alternative.



This optimization has been applied in order to obtain the handshake circuit of Figure 1.11.

□

The above form of local optimization will be called *peephole optimization* by analogy to optimizations in conventional compilers that generate machine code [McK65]. By scanning over a handshake circuit and looking at a local subcircuit (with a bounded diameter), as if through a peephole, one can find opportunities for improvements by substitution. It is characteristic of peephole optimization that an improvement may spawn opportunities for additional improvements.

7.2 Non-receptive handshake components

By Definition 2.19 handshake processes are required to be receptive. The main advantage of this requirement is the relatively simple definition of parallel composition of handshake processes (Definition 3.14), which guarantees absence of computation interference (Theorem A.12).

Unfortunately, the requirement of receptiveness tends to make the circuit realizations of handshake processes more costly. In particular, the constant readiness for inputs through passive ports requires gates or latches to "shield" or remember input transitions for later processing. Moreover, the choice construct requires an arbiter circuit to arbitrate between transitions through the guard inputs. A non-receptive implementation of choice is deterministic and arbitration can therefore be avoided (cf. the *MIX* and *NMIX* components of Example 2.23).

In this section we investigate the conditions under which a (receptive) handshake component may be replaced by a non-receptive component without taking the risk of computation interference. The scope of the following (re-)definition is restricted to the current section.

Definition 7.4

0. A handshake process is a handshake structure that satisfies all conditions of 2.19 except for condition 2.19.4.

1. A *receptive* handshake processes is a handshake process that also satisfies condition 2.19.4. All other handshake processes are *non-receptive*.

□

Let P be a handshake process, and let $t \in \mathbf{t}P^{\leq}$ and $a \in \mathbf{i}P$, such that $ta \in (\mathbf{a}P)^H$.

Receptiveness implies $ta \in \mathbf{t}P^{\leq}$. Conversely, $ta \notin \mathbf{t}P^{\leq}$ implies that P is non-receptive. Examples of non-receptive processes are *NMIX* and *NVAR* of Example 2.23.

This generalized form of handshake processes is distinctly more complicated than receptive handshake processes. It is not our intention to develop a theory of (non-receptive) handshake processes. We point out two essential differences in order to prepare ourselves for a particular application of non-receptive handshake processes.

Refinement ordering of handshake processes is rather subtle compared to that of receptive processes. Consider receptive process $CON\cdot(a^\circ, c^\bullet)$. Its trace set is also a valid trace set for a process with port structure A, where $A = a^\circ \cup b^\circ \cup c^\bullet$. Let Q be the handshake process with port structure A and the quiescent trace set of $CON\cdot(a^\circ, c^\bullet)$. Process Q is clearly non-receptive, since trace b_0 is not in its trace set. Interestingly, we have $\mathbf{t}Q \subset \mathbf{t}MIX\cdot(a^\circ, b^\circ, c^\bullet)$. However, this does not make Q a suitable implementation of the mixer!

Also parallel composition of handshake processes is more complicated when processes are not receptive. Consider the parallel composition of Q and $RUN\cdot\{b^\bullet\}$. In their asynchronous interaction, process $RUN\cdot\{b^\bullet\}$ will output a b_0 despite the fact that Q is not ready to receive it: a clear case of interference.

Note that the trace set of Q **w** $RUN\cdot\{b^\bullet\}$ equals that of Q. Weaving ignores the distinction between input and output, and is therefore not a suitable basis for parallel composition when there the danger of interference exists.

The parallel composition of Q and $STOP\cdot\{b^\bullet\}$ is free of interference. (With concealment of b we obtain the receptive process $CON\cdot(a^\circ, c^\bullet)$.) In other words, only if Q operates in an environment that never produces a b_0, may the mixer be replaced by Q. More generally, a process may be refined into a process with fewer possible behaviors when the environment restrains itself appropriately.

Let, for the remainder of this section, P, Q, and R be handshake processes, such that $\mathbf{p}P = \mathbf{p}Q$ and $\mathbf{p}P \bowtie \mathbf{p}R$.

Definition 7.5 (strong refinement in context)

P strongly refines to Q in the context of R, denoted by $P \sqsubseteq_R Q$, if

$$\mathbf{t}P \supseteq \mathbf{t}Q \quad \wedge \quad (\mathbf{t}(P \mathbf{w} R) = \mathbf{t}(Q \mathbf{w} R))$$

□

Property 7.6

0. Strong refinement in context is a preorder (cf. 7.2).
1. For receptive handshake processes P, Q and R we have

$$P \sqsubseteq_R Q \;\Rightarrow\; P \sqsubseteq_R Q$$

□

Example 7.7

The relation

$$MIX \cdot (b^\circ, c^\circ, d^\bullet) \;\sqsubseteq_R\; NMIX \cdot (b^\circ, c^\circ, d^\bullet)$$

is a strong refinement for

$$R \in \{OR \cdot (a^\circ, b^\bullet, c^\bullet)\ ,\ SEQ \cdot (a^\circ, b^\bullet, c^\bullet)\ ,\ STOP \cdot \{b^\bullet, c^\bullet\}\}$$

but not for

$$R \in \{PAR \cdot (a^\circ, b^\bullet, c^\bullet)\ ,\ RUN \cdot \{b^\bullet, c^\bullet\}\}$$

□

A nice property of strong refinement in context is that it does not introduce interference. The following theorem assumes that the domain of parallel composition (cf. Definition 3.14) is extended to handshake processes.

Theorem 7.8

If $P \mid\mid R$ is free of interference and $P \sqsubseteq_R Q$ then $Q \mid\mid R$ is also free of interference.

□

Remember that if P and R are both receptive, absence of interference is guaranteed. In particular, a component in a handshake circuit and its environment are receptive. If this component is e.g. a *MIX* component it may be replaced by an *NMIX* on account of Example 7.7, provided that the environment avoids overlaps of the handshakes through the mixer's passive ports. The *MIX* components introduced in the compilation function of Chapter 6 are all placed in such a restricted environment. Consequently:

Theorem 7.9

All *MIX* components introduced on ground of the separation Property (cf. 6.23) can be strongly refined into *NMIX* components in their respective contexts.

□

Similar strong refinements in context apply to directed *MIX* components. Also, *VAR* components can be strongly refined into a non-receptive component without overlap in read and write access, because of the required conformance of alphabet structures (cf. Definition 4.12).

7.3 Handshake refinement

Symbols have been introduced as names of events of interest to describe the interaction of a handshake process and its environment. In this section we relate these symbols to wires and to transitions of the states of these wires. In CMOS these state transitions usually correspond to voltage transitions (cf. Section 0.1).

Let p be a port of handshake process P. First we assume that we may use a wire for each symbol of p, the so-called *One-Hot* encoding[0]. With symbol a we associate wire a. The states of a wire will be referred to as *low* and *high*. A transition of wire a from low to high will be denoted by $a\uparrow$, and a transition vice versa by $a\downarrow$. If we assume that the initial state of a wire is low, the observed behavior of the state of the wire can be recorded by a sequence in which $a\uparrow$ and $a\downarrow$ alternate, starting with an $a\uparrow$.

The notion of a handshake process can be refined accordingly. We will not develop a formal handshake-process model at the level of transitions. Instead, we simply require port structures to be of the form $A \times \{\uparrow,\downarrow\}$. Also, the projection of a trace of $\mathcal{H}\cdot(A \times \{\uparrow,\downarrow\})$ on a single port must result in the proper alternation of up and down transitions. Without loss of generality we assume that all wires are low initially and that the first event on each wire is therefore an $\uparrow$ transition. A handshake process with these properties is referred to as a *transition handshake process*. Similarly, we speak of transition handshake components and transition handshake circuits.

Useful shorthands for ports $\langle p,\uparrow\rangle$ and $\langle p,\downarrow\rangle$ are $p\uparrow$ and $p\downarrow$. The alphabet structure $A \times \{\uparrow,\downarrow\}$ will be abbreviated to A_t. The set of all transition handshake processes with port structure A_t will be denoted by $\prod_t \cdot A$.

[0]More economic encodings are discussed in Section 7.4.

Handshake action $p\uparrow^\circ$ assumes both wires p_0 and p_1 to be low, and, after successful termination, leaves both wires high. Similarly, action $p\downarrow^\circ$ assumes both wires to be high, and leaves both wires low, provided successful termination. This suggests that we can still use the handshake calculus of Chapter 4, provided that the described process satisfies the rule of alternation of up and down transitions.

In this section we investigate various ways to implement a handshake process by a transition handshake process. The central notion is that of phase reduction.

Definition 7.10 (phase reduction)

A *phase reduction* is a partial function $\phi : \prod_t \cdot A \rightarrow \prod \cdot A$ that satisfies:

0. ϕ is surjective
1. $\phi \cdot (P \sqcap Q) = \phi \cdot P \sqcap \phi \cdot Q$
2. $\phi \cdot (P \sqcup Q) = \phi \cdot P \sqcup \phi \cdot Q$
3. $\phi \cdot (P \parallel Q) = \phi \cdot P \parallel \phi \cdot Q$

□

Let $R = \phi Q$. Then R is said to be the phase reduction of Q. Alternatively, Q will be called a *handshake refinement* of R. A phase reduction is a homomorphism on account of 7.10.1 and 7.10.2.

Transition handshake processes P and Q are *equivalent* if $\phi \cdot P = \phi \cdot Q$. This equivalence is actually a *congruence*, and is the *kernel* of ϕ (cf. [DP90] page 116).

Let P, Q and R be transition handshake processes. On account of 7.10.3 we conclude $P \parallel R$ and $Q \parallel R$ are equivalent if P and Q are equivalent. More generally, the replacement of a transition handshake component by an equivalent one in a transition handshake circuit results in an equivalent handshake circuit.

Two classes of phase reductions are studied in some detail: 2-phase and 4-phase reductions. The associated handshake refinements are called 2-phase and 4-phase refinements respectively.

2-phase refinements

The simplest handshake refinement is based on the phase reduction obtained by ignoring the distinction between up and down transitions.

Definition 7.11 (2-phase reduction)

0. Let t be a handshake trace in A_t^H. The *2-phase reduction* of t, denoted by $\phi_2 \cdot t$, is defined by:

$$\begin{array}{lllll} \phi_2 \cdot t = & \textbf{if} & t = \varepsilon & \rightarrow & \varepsilon \\ & [] & t = c\uparrow\ u & \rightarrow & c\ \phi_2 \cdot u \\ & [] & t = c\downarrow\ u & \rightarrow & c\ \phi_2 \cdot u \\ & \textbf{fi} & & & \end{array}$$

1. Let P be a transition-handshake process. $\phi_2 \cdot P$ is the handshake process obtained by applying the 2-phase reduction to all traces of P.

□

The following theorem is hardly a surprise.

Theorem 7.12

ϕ_2 is a phase reduction.

□

ϕ_2 is total and is clearly a bijection. Its inverse will be called the *2-phase handshake refinement* of a handshake process. The 2-phase handshake refinements of a number of handshake components are given next.

Example 7.13

0. $CON \cdot (a^\circ, b^\bullet) = \#[a\uparrow^\circ: b\uparrow^\bullet; a\downarrow^\circ: b\downarrow^\bullet]$

1. $SEQ \cdot (a^\circ, b^\bullet, c^\bullet) = \#[a\uparrow^\circ: (b\uparrow^\bullet; c\uparrow^\bullet); a\downarrow^\circ: (b\downarrow^\bullet; c\downarrow^\bullet)]$

2. $REP \cdot (a^\circ, b^\bullet) = a\uparrow^\circ: \#[b\uparrow^\bullet; b\downarrow^\bullet]$

3. $MIX \cdot (a^\circ, b^\circ, c^\bullet) = M_{(0,0)}$,
where

$$\begin{array}{lll} M_{(0,0)} & = [a\uparrow^\circ: c\uparrow^\bullet;\ M_{(1,0)} & \mid\ b\uparrow^\circ: c\uparrow^\bullet;\ M_{(0,1)}] \\ M_{(1,0)} & = [a\downarrow^\circ: c\downarrow^\bullet;\ M_{(0,0)} & \mid\ b\uparrow^\circ: c\downarrow^\bullet;\ M_{(1,1)}] \\ M_{(0,1)} & = [a\uparrow^\circ: c\downarrow^\bullet;\ M_{(1,1)} & \mid\ b\downarrow^\circ: c\downarrow^\bullet;\ M_{(0,0)}] \\ M_{(1,1)} & = [a\downarrow^\circ: c\uparrow^\bullet;\ M_{(0,1)} & \mid\ b\downarrow^\circ: c\uparrow^\bullet;\ M_{(1,0)}] \end{array}$$

□

The above mixer in particular is considerably more complicated than the *MIX* component of Example 4.3. The 2-phase refinements of directed components such as variables, multiplexers and adders are distinctly more complex than the above mixer.

The handshake protocol that results from 2-phase refinement is also known as *2-cycle signaling* or *non-return-to-zero signaling* [Sei80]. The good news about 2-phase refinement is that it results in handshake circuits in which components interact by the minimum number of transitions possible. Consequently, these circuits are potentially as fast and energy-efficient as possible. The bad news is that circuits sensitive to voltage transitions tend to be significantly larger than circuits sensitive to voltage levels [Sei80]. This overhead in circuit size may reduce the speed and power benefits considerably.

The advantages of 2-phase refinements are likely to dominate in the case of off-chip communication and, to a lesser extent, for long-distance on-chip communication.

4-phase refinements

4-phase refinements form practical alternatives to 2-phase refinements. The resulting handshake protocols are known as Muller signaling, 4-cycle signaling or return-to-zero signaling [Sei80]. The essence of 4-phase refinements is that handshakes are implemented by a signaling sequence of four communications. A first form is based on *complete 4-phase reduction*:

Definition 7.14 (complete 4-phase reduction)

Let $\langle A_t, T\rangle$ be a transition handshake process, and let $C = \mathbf{0}A\uparrow \cup \mathbf{1}A\downarrow$ (C consists of the first and fourth phases of a 4-phase handshake). The *complete 4-phase reduction* of $\langle A_t, T\rangle$, denoted by $\phi_{4c}\cdot\langle A_t, T\rangle$, is defined only for $\langle A_t, T\rangle$ that satisfy

$$(\forall t : t \in T : suc\cdot(t, \mathbf{t}T) \subseteq \mathbf{i}A_t \cup C)$$

and results in the handshake process

$$\langle A, \{t : t \in T \ \wedge \ suc\cdot(t, \mathbf{t}T) \subseteq C : \phi_2\cdot(t\lceil C)\}\rangle$$

□

Complete 4-phase reduction is based on the concealment of symbols in the complement of C, viz. $\mathbf{0}A\downarrow \cup \mathbf{1}A\uparrow$. The restriction on the domain of ϕ_{4c} excludes transition handshake processes that become quiescent while capable of doing

an output in the complement of C. The reduction selects only those traces for projection on C that have successors in C.

Theorem 7.15

ϕ_{4c} is a phase reduction.

□

In contrast to ϕ_2, the 4-phase reduction is *not* a bijection. The complete 4-phase handshake refinement of a handshake process is usually not unique. Examples of complete 4-phase expansions of some handshake components are given below.

Example 7.16

0. The two-phase connector is suitable in a four-phase setting as well. Alternatives are

$$\#[a\uparrow^\circ: (b\uparrow^\bullet; b\downarrow^\bullet); a\downarrow^\circ]$$

and

$$\#[a\uparrow^\circ; a\downarrow^\circ: (b\uparrow^\bullet; b\downarrow^\bullet)]$$

1. A suitable four-phase version of the sequencer is

$$\#[a\uparrow^\circ: (b\uparrow^\bullet; b\downarrow^\bullet; c\uparrow^\bullet); a\downarrow^\circ: c\downarrow^\bullet]$$

An alternative is

$$\#[a\uparrow^\circ: b\uparrow^\bullet; a\downarrow^\circ: (b\downarrow^\bullet; c\uparrow^\bullet; c\downarrow^\bullet)]$$

2. A four-phase repeater can be identical to the two-phase version.

3. The four-phase *MIX* is remarkably simple compared to the two-phase version:

$$MIX\cdot(a^\circ, b^\circ, c^\bullet) = \#[[a\uparrow^\circ: c\uparrow^\bullet; a\downarrow^\circ: c\downarrow^\bullet \mid b\uparrow^\circ: c\uparrow^\bullet; b\downarrow^\circ: c\downarrow^\bullet]]$$

□

An important property of complete 4-phase refinement is that the wires of a port are in their initial states after the completion of each handshake. In most cases the circuits are therefore simpler than their 2-phase counterparts. An

obvious disadvantage is the doubling of the number of transitions, with associated penalties in power consumption and computation time. The latter disadvantage can be relaxed somewhat by adopting the following alternative 4-phase reduction:

Definition 7.17 (quick 4-phase reduction)

Let $\langle A_t, T\rangle$ be a transition handshake process, and let $C = \mathbf{0}A\uparrow \cup \mathbf{1}A\uparrow$ (With this choice, C consists of the first and second phases of a 4-phase handshake). The *quick 4-phase reduction* of $\langle A_t, T\rangle$, denoted by $\phi_{4q} \cdot \langle A_t, T\rangle$, is defined in the same way as ϕ_{4c}, taking the difference in symbol set C into account.

□

Theorem 7.18

ϕ_{4q} is a phase reduction.

□

ϕ_{4q} is not a bijection and a associated handshake refinement is therefore not unique. Quick 4-phase refinements tend to be faster than complete 4-phase refinements, because the environment does not need to participate in the return-to-zero transitions. The price for this gain in speed is that the circuits tend to be more complex, because after output transition $a_1\uparrow$ the component must be receptive for $a_0\downarrow$ while possibly engaging in other handshakes.

Mixed forms of complete and quick 4-phase refinements may be considered, with the objective of taking the best of both worlds: quick 4-phase refinement when the speed gain is substantial and the overhead in circuit complexity is acceptable, and complete 4-phase refinement in all other instances. Of course, such mixed refinments must be based on a proper phase reduction. A useful transition handshake component to convert a complete 4-phase refinement into a quick one on a single-port basis is the *quick-return linkage* [1].

Example 7.19 (quick-return linkage)

The transition handshake component $QRL \cdot (a^\circ, b^\bullet)$ is defined as

$$\#[(a\uparrow^\circ : b\uparrow^\bullet); (a\downarrow^\circ \,||\, b\downarrow^\bullet)]$$

□ [1] In [Udd84] attributed to C.L. Seitz.

Component *QRL* is a quick 4-phase handshake refinement of a connector. Observe that $a\uparrow^\circ$ and $b\uparrow^\bullet$ are coupled as in *CON*, and that $a\downarrow^\circ$ and $b\downarrow^\bullet$ can occur independently as in *PAR*.

Transferrers

Realizations of handshake components using 4-phase handshake refinements lead to reasonably efficient VLSI circuits. The current Tangram compiler (cf. Chapter 8) uses the complete 4-phase handshake refinement for all handshake components, *except* for the transferrer. Recall that transferrers are introduced abundantly in the compilation of Tangram programs. The behavior of a transferrer with activation port a°, input $b^\bullet$ and output $c^\bullet$ is defined by (assuming appropriate declarations of a, b, c and x):

$$\#[a^\circ : (b^\bullet?x; c^\bullet!x)]$$

A complete 4-phase refinement is:

$$\#[a\uparrow^\circ: (b\uparrow^\bullet?x\ ;\ b\downarrow^\bullet?x\ ;\ c\uparrow^\bullet!x\ ;\ c\downarrow^\bullet!x)\ ;\ a\downarrow^\circ]$$

Other 4-phase refinements have in common that the b and c handshakes are strictly sequential, requiring costly storage of the incoming value between the communications $b\downarrow^\bullet?x$ and $c\uparrow^\bullet!x$.

A transition handshake component with a behavior similar to the transferrer, and with an extremely cheap circuit realization is (cf. Section 7.5):

$$\#[a\uparrow^\circ: (b\uparrow^\bullet?x\ ;\ c\uparrow^\bullet!x)\ ;\ a\downarrow^\circ: (b\downarrow^\bullet?x\ ;\ c\downarrow^\bullet!x)]$$

The reductions in cost and delays have been achieved by creating an overlap between the b and the c handshake. It turns out that, with few exceptions, this handshake refinement of the transferrer is allowed in the compiled handshake circuits. This can be checked for each syntax/compilation rule that introduces transferrers. Exceptions are assignments of the form $x := E$ in which E depends on the value of x. For these so-called *auto assignments* (e.g. $i := i + 1$) we have to accept the more expensive handshake refinement.

7.4 Message encoding

In the previous section we assumed a One-Hot encoding of data: to each symbol in the two symbol sets of a port we assigned a wire. A set of 16 symbols then requires 16 wires. On the other hand, 16 wires may encode as many as 2^{16}

(= 65536) values. This suggests ample room for improvement over One-Hot encoding. Our prime interest is in encodings that preserve the delay-insensitive nature of the communication among handshake components. [Ver88] presents a definition and an overview of these so-called "delay-insensitive codes". In this section we repeat this definition and link it to the most popular delay-insensitive code: the double-rail code, also known as the dual-rail code.

A *code* is a pair (I, C), where I is a finite set of indexed wires, and C is a set of subsets of I: the code words. The size of I is called the *length* of the code, and the size of C is called the code's *size*. A One-Hot code of size n has length n.

The implementation of port $\langle \mathbf{0}p, \mathbf{1}p \rangle$ requires a code for both $\mathbf{0}p$ and $\mathbf{1}p$. In most cases, however, at least one of these two sets is a singleton (code size = 1), and a single wire suffices (code length = 1).

A code word is an element of C and indicates along which wires a transition will be sent for the transmission of the corresponding message. Not all code words are suitable for delay-insensitive communication. For instance, the empty set is useless, because the receiver would not be able to detect its arrival.

Definition 7.20

A code (I, C) is *delay insensitive* when [Ver88] when

$$(\forall x, y : x \in C \wedge y \in C \wedge x \subseteq y : x = y)$$

□

That is, when no code word is contained in another code word. This property allows the receiver to detect the arrival of a message. After a transition has arrived on each wire of a code word, the receiver can detect that it has received a complete message.

The *concatenation* of codes (I, C) and (J, D) with $I \cap J = \emptyset$ is the code $(I \cup J, CD)$, where CD is defined by

$$\{x, y : x \in C \wedge y \in D : x \cup y\}$$

The concatenation of two delay-insensitive codes is also delay-insensitive.

The well-known *Double-Rail code* [Sei80] can now be introduced as the concatenation of n (disjoint) One-Hot codes of length 2. Using 16 wires, a Double-Rail code of 8 wire pairs encodes 2^8 (=256) code words, which is a clear improvement over the 16 code words of the One-Hot code. Arrival detection of Double-Rail encoded messages is simple, so is the conversion from and to

(delay-sensitive) Single-Rail codes. These properties make Double-Rail codes fairly popular in the design of delay-insensitive and other self-timed circuits.

However, compared with clocked circuits, in which a transition of the clock announces the arrival of a message, Double-Rail codes are rather wasteful. For a given code size, the overhead in number of wires is 100 %, which is considerable in a technology where costs are dominated by wires. Other delay-insensitive codes [Ver88] have considerably less overhead. However, their use in the realization of handshake circuits is constrained by:

- the lack or excessive costs of circuits for arithmetic with such codes,
- the size of encoding and decoding circuits near storage elements, and
- the delays involved in encoding and decoding.

For off-chip and long-distance on-chip communication these overheads of coding and decoding may nevertheless be worthwhile.

When wire delays can be sufficiently controlled, delay-*sensitive* codes become attractive for circuit realization. With n wires, a code of size 2^{n-1} can be implemented, in which one wire is used to signal the arrival of a message. Of course, the delay along that wire must exceed the delay in each of the $n-1$ other wires. This is sometimes called a data-bundling constraint [Sut89]. In practice this requires sufficient control over the spatial layout of handshake components and the connecting wires, introduction of additional delays or a combination of both. Conversion circuits from and to Double-Rail codes are given in [Sei80].

7.5 Handshake components → VLSI circuits

The previous sections show how the specification of a handshake component can be refined by:

0. reducing receptiveness (depending on the component's context),
1. refining handshakes, and
2. encoding messages.

These refinements result in the specification of a circuit in terms of transitions on individual input and output wires. The next step is to decompose such a specification into a circuit of available VLSI primitives such as inverters and *NAND* gates. Methods for these decompositions are emerging ([Mar89,Ebe89, MBM90,JU91]), with different choices in and emphases on:

0. the handshake protocol chosen,
1. the available VLSI primitives,
2. the degree of delay-insensitivity.

In this section we review decompositions of a few handshake components into circuits of so-called *VLSI operators*.

VLSI operators and (isochronic) forks

The behavior of a VLSI operator is defined by the Boolean values of the output wire(s) in terms of present and past values of the inputs. The behavior of a monadic (single output) operator is specified by a pair of so-called *production rules* [Mar89]

$$\begin{aligned} F &\mapsto z\uparrow \\ G &\mapsto z\downarrow \end{aligned}$$

F and G are Boolean expressions called the *guards* of the operator. The identifiers in F and G are the inputs of the operator. z is the output of the operator. $z\uparrow$ and $z\downarrow$ are shorthand for $z := \mathit{true}$ and $z := \mathit{false}$ respectively. The production rule $F \mapsto z\uparrow$ can be read as "when F holds z becomes *true*".

The guards of an operator are required to be mutually exclusive, i.e. $\neg F \vee \neg G$ must hold at any time. Furthermore, the guards have to be *stable*, i.e. once a guard evaluates to *true*, it has to remain *true* until the completion of the corresponding output transition. Stability of the guards is *not* a property of the operator: it must be satisfied by the environment of the operator. The same holds for the mutual exclusion of the guards of operators for which $\neg F \vee \neg G$ is not a tautology.

An input transition denotes the change of an input variable. An input transition is *productive* if it causes an output transition, and *void* otherwise. The time between a productive input transition and the corresponding output transition may be arbitrary (i.e. positive and finite).

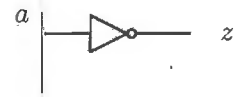
An operator is called "combinational" if $F \vee G$ is a tautology, and "sequential" or "state-holding" otherwise.

Some examples of operators are given below. They will return in later examples.

Example 7.21

0. An inverter is specified by

$$\begin{array}{rcl} \neg a & \mapsto & z\uparrow \\ a & \mapsto & z\downarrow \end{array}$$

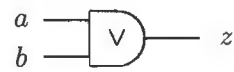


1. The familiar *AND* operator is specified by

$$\begin{array}{rcl} a \wedge b & \mapsto & z\uparrow \\ \neg a \vee \neg b & \mapsto & z\downarrow \end{array}$$

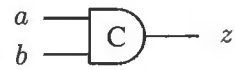
2. Similarly, the *OR* operator is specified by

$$\begin{array}{rcl} a \vee b & \mapsto & z\uparrow \\ \neg a \wedge \neg b & \mapsto & z\downarrow \end{array}$$



3. The previous three operators are combinational operators. A well-known example of a sequential operator is the Muller-C element:

$$\begin{array}{rcl} a \wedge b & \mapsto & z\uparrow \\ \neg a \wedge \neg b & \mapsto & z\downarrow \end{array}$$



Note that this specification allows two successive transitions on an input, provided stability of the guards.

□

CMOS implementations of VLSI operators are discussed in [Mar89,vB91]. An example of a CMOS circuit for a Muller-C element is depicted in Figure 7.0. It consists of a Majority circuit with it output z fed back to one of its inputs. Note that during the (dis-)charging of wire y two paths of transistors pull together.

VLSI operators may be connected by (point-to-point) wires. Wires themselves may be regarded as VLSI operators:

$$\begin{array}{rcl} a & \mapsto & z\uparrow \\ \neg a & \mapsto & z\downarrow \end{array}$$

However, since VLSI operators may have arbitrary delays themselves, there is no point in introducing extra variables here. Therefore we treat wires as single variables.

When a value is to be transmitted to the inputs of two operators a *FORK* operator must be used. The *FORK* operator has two outputs, both following the input:

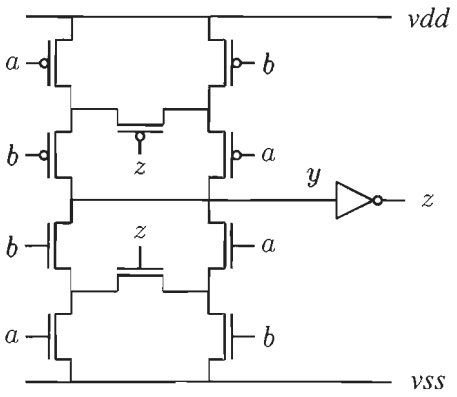
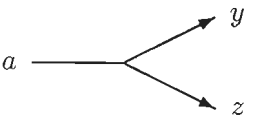


Figure 7.0: A CMOS circuit for a Muller-C element. Wires with the same label are connected. *vdd* and *vss* denote the power and ground rail respectively..

$$\begin{array}{lcl} a & \mapsto & y\uparrow, z\uparrow \\ \neg a & \mapsto & y\downarrow, z\downarrow \end{array}$$

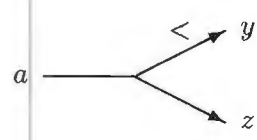


The comma between the output transitions expresses concurrency: the two events may occur in either order and no assumption is made about the time duration between these events. (Simultaneous occurrence of both events cannot be expressed in the model). In implementation technologies where wire delays may dominate other delays (such as CMOS) it turns out to be necessary to represent the outputs of the fork by two distinct variables.

A network of VLSI operators and forks is said to be *delay insensitive* if it functions correctly under arbitrary and possibly varying delays in operators and wires. This rather extreme class of asynchronous circuits has the additional advantage that it simplifies the layout: delays introduced by wires do not affect the behavior of the circuit. Unfortunately, the class of (purely) delay-insensitive circuits constructed from operators and wires only is small and not very interesting from a practical view point (cf. [BE90] and [Mar90]).

The "weakest possible compromise" [Mar90] with respect to delay insensitivity seems to be a forking wire with constraints on the arrival times of transitions at the ends of the fork: the *isochronic fork*. An isochronic forks is a special case of the *FORK* operator. Below we present two types of isochronic forks. An *asymmetric* isochronic fork guarantees that one output transition occurs before the other, as expressed by the semicolon:

$$\begin{array}{lcl} a & \mapsto & y\uparrow; z\uparrow \\ \neg a & \mapsto & y\downarrow; z\downarrow \end{array}$$



In circuit diagrams such forks are indicated by a '<' at the fast end. A *symmetric* isochronic fork guarantees that both output transitions occur at the same time: the names of the outputs are simply aliases for the same variable. In circuit diagrams such forks are indicated by a '=' near the fork.

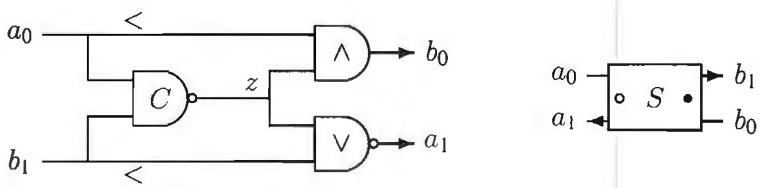
The above classification of isochronic forks is a coarse one. By requiring different timing behaviors for up- and down-going transitions finer classifications can be obtained. Isochronic forks must be applied with caution and implemented with care [vB92].

Networks of VLSI operators and (isochronic) forks are *speed independent* [Mil65,Rem91].

A useful auxiliary circuit for the realization of complete 4-phase transition handshake components is the *S-element* [vB92]. An S-element has as port structure $a^\circ \cup b^\bullet$, and can be specified by the transition handshake process

$$\#[a\uparrow^\circ: (b\uparrow^\bullet; b\downarrow^\bullet); a\downarrow^\circ]$$

A possible circuit realization of the S-element in terms of VLSI operators is given below.



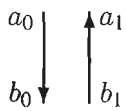
Initially, $a_0 = a_1 = b_0 = b_1 = \mathit{false}$ and $z = \mathit{true}$. The forks connected to a_0 and b_1 are both isochronic. In [Mar89] the circuit is called a Q-element. The D-element of [BM88] behaves similarly, but is not strongly initializable (cf. Section 7.6), whereas the S-element is.

Circuit realizations for some handshake components

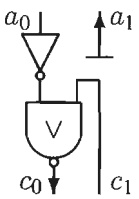
The circuit realizations of the handshake components below are all based on complete 4-phase refinements (cf. Example 7.13).

Example 7.22

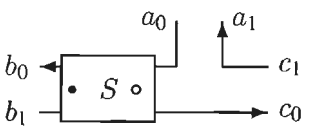
0. A circuit realization of a connector consists of wires only:



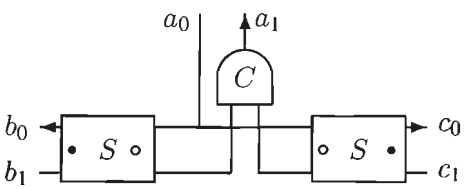
1. A repeater consists of a *NOR* and an inverter. Output a_1 is connected to ground and will not be involved in any transition.



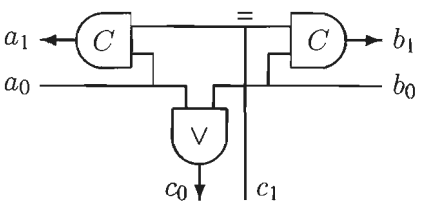
2. A circuit for the sequencer is based on the S-element.



3. The *PAR* component can for instance be realized as:

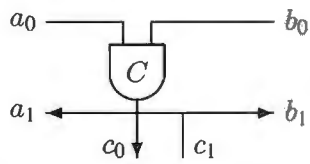


4. A circuit for the non-receptive mixer is:

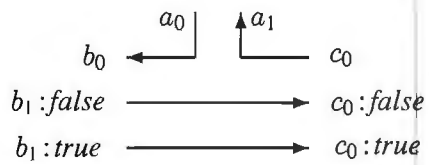


The fork connected to c_1 is isochronic.

5. An undirected *JOIN* can be realized by

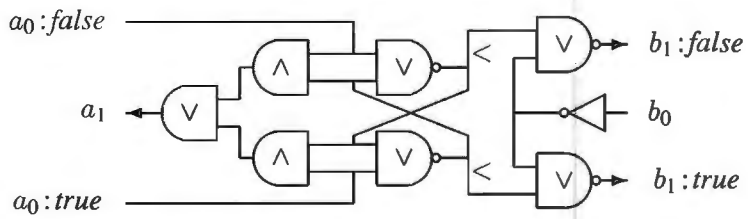


6. A Boolean transferrer can be realized by wires only, provided that the b and c handshakes may overlap (cf. Section 7.3).



For an n-bit transferrer ($n > 1$) the above transferrer can be extended by adding $n - 1$ wire pairs from port b to port c.

7. The circuit below is that of a Boolean variable with a single read port. Transition $a_1\uparrow$ acknowledges a write action; transition $b_0\uparrow$ is a read request.



The circuit of a 2-bit variable can be realized by two 1-bit variables, a fork connected to the two read-request inputs, and a Muller-C element that joins the write-acknowledgements (with inputs and outputs appropriately labeled).

□

The presented circuit solutions are not claimed to be optimal. Clever combinations of operators on the transistor level often yield interesting savings. The circuits of Example 7.22 are sufficient to realize the optimized handshake circuit of the two-place buffer of Example 1.4.

7.6 Initialization

When a VLSI circuit is connected to a power supply, the circuit generally does not proceed to an initial state by itself. *If* the circuit becomes quiescent after some time, the resulting state may not even be reachable from an initial state. Also, the circuit may start oscillating (diverging), even when its specification does not allow for such oscillations.

So, we have a circuit-initialization problem: how to force a VLSI circuit into its initial state. This problem is not specific to self-timed or other asynchronous circuits. In clocked circuits, this problem is solved by the introduction of additional *reset* circuitry. This circuitry can be used to force a well-chosen subset of all wires into their initial states. This strategy is also applicable to the VLSI circuits studied in this chapter. Nevertheless, we develop a different strategy that avoids the need for additional circuitry and that builds on the properties of compiled handshake circuits.

First we take stock of the properties of handshake components and circuits that will be used:

- The initial state of handshake components is passive: only an input event can cause a handshake component (and hence a handshake circuit) to leave the initial state (cf. 3.19).
- Handshake components and handshake circuits have the initial-when-closed property. Hence, a handshake circuit is in its initial state if and only if all its ports (both internal and external) are also (cf. 3.19).
- The environment of a handshake circuit has only control over the inputs of the external ports of that handshake circuit.

It must be stressed that the behavior of the VLSI circuit after power-on *cannot* be analyzed within the model for handshake circuits, since all kinds of interference may occur. Fortunately, we are not interested in this behavior; we only want a guarantee that the circuit will arrive in an initial state within a finite and predictable amount of time.

The initialization properties of handshake circuits will be analyzed in terms of the binary relation $\rightsquigarrow$ between symbol sets.

Definition 7.23 (initializes)

Let B and C be a symbol sets. $B \rightsquigarrow C$ (pronounced as "B initializes C") is a binary relation with the following properties:

0. $C \subseteq B \;\Rightarrow\; B \rightsquigarrow C$

1. $B \rightsquigarrow C \wedge C \rightsquigarrow D \;\Rightarrow\; B \rightsquigarrow D$

2. $B \rightsquigarrow C \wedge D \rightsquigarrow E \;\Rightarrow\; B \cup C \rightsquigarrow D \cup E$

□

The following properties follow immediately from the definition of $\rightsquigarrow$.

Property 7.24

0. From 0 we can see that $\rightsquigarrow$ is reflexive, i.e. $B \rightsquigarrow B$.

1. Together with 1 we conclude that $\rightsquigarrow$ is a preorder.

2. Combining 0 and 1 yields $B \rightsquigarrow C \wedge B \subseteq D \;\Rightarrow\; D \rightsquigarrow C$.

3. Alternatively, $B \rightsquigarrow C \wedge D \subseteq C \;\Rightarrow\; B \rightsquigarrow D$.

4. Finally, combining 0 and 2 yields $B \rightsquigarrow C \;\Rightarrow\; B \rightsquigarrow B \cup C$.

□

Our aim is to develop a $\rightsquigarrow$ relation on $(\cup P : P \in H : \mathbf{a}P)$ for handshake circuit H, given such relation for the constituent handshake components.

Definition 7.25 (weak initializability)

0. A handshake component P is weakly initializable if it is passive, initial-when-closed and $\mathbf{i}P \rightsquigarrow \mathbf{o}P$. On ground of Property 7.24.4 the latter is equivalent to $\mathbf{i}P \rightsquigarrow \mathbf{a}P$.

1. A handshake circuit H is weakly initializable if its constituent handshake components are and $\mathbf{i}(\mathbf{e}H) \rightsquigarrow \mathbf{o}(\mathbf{e}H)$, where $\mathbf{e}H$ denotes the external port structure of H (cf. Definition 3.8).

□

Clearly, a weakly initializable handshake component can be forced into its initial state by making all inputs initial.

Example 7.26

All circuits of Example 7.22 are weakly initializable.

□

Unfortunately, requiring all handshake components to be weakly initializable (Definition 7.25.0) is not sufficient to make a handshake circuit weakly initializable (Definition 7.25.1). In order to make handshake circuits weakly initializable, additional provisions are required. We first examine a simple strategy that is effectively and efficiently applicable to undirected handshake circuits. A more general, but also more elaborate strategy is sketched next.

A simple initialization strategy

Definition 7.27 (strong initializability)

0. A weakly initializable handshake component P is strongly initializable if $\mathbf{i}P^{\circ} \rightsquigarrow \mathbf{o}P^{\bullet}$.

1. Accordingly, a weakly initializable handshake circuit H is strongly initializable if its constituent handshake components are and $\mathbf{i}(\mathbf{e}H)^{\circ} \rightsquigarrow \mathbf{o}(\mathbf{e}H)^{\bullet}$.

□

Example 7.28

All circuits of Example 7.22 *except* the transferrer are strongly initializable.

□

The next theorem expresses that strong initializability is preserved under parallel composition, provided that the associated *activity graph* is acyclic.

Definition 7.29 (activity graph)

An activity graph is a directed graph. The activity graph associated with a handshake circuit has one node for each handshake component and one arc for each channel, directed from the active port to the passive port of that channel.

□

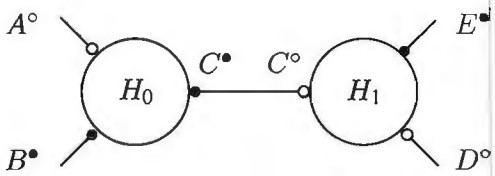
Theorem 7.30 (initialization)

Let H be a handshake circuit whose associated activity graph is acyclic and

whose constituent handshake components are strongly initializable. Then H is weakly as well as strongly initializable.

Proof When circuit H is empty or a singleton, the theorem is trivial. In the case H consists of at least two components, it can be decomposed into two non-empty subcircuits H_0 and H_1 such that $\mathbf{p}^\circ H_0 \cap \mathbf{p}^\bullet H_1 = \emptyset$ (here we use the acyclicity of H). We use the following abbreviations (see picture below):

- $A^\circ = \langle \mathbf{p}^\circ H_0, \emptyset \rangle$
- $B^\bullet = \langle \emptyset, \mathbf{p}^\bullet H_0 \setminus \mathbf{p}^\circ H_1 \rangle$
- $C^\bullet = \langle \emptyset, \mathbf{p}^\bullet H_0 \cap \mathbf{p}^\circ H_1 \rangle$
- $C^\circ = \langle \mathbf{p}^\bullet H_0 \cap \mathbf{p}^\circ H_1, \emptyset \rangle$
- $D^\circ = \langle \mathbf{p}^\circ H_1 \setminus \mathbf{p}^\bullet H_0, \emptyset \rangle$
- $E^\bullet = \langle \emptyset, \mathbf{p}^\bullet H_1 \rangle$



Note that $\mathbf{p}H_0 = \langle A, B \cup C \rangle$ and $\mathbf{p}H_1 = \langle C \cup D, E \rangle$. Also, $\mathbf{i}C^\circ = \mathbf{o}C^\bullet$ and $\mathbf{o}C^\circ = \mathbf{i}C^\bullet$. *Weak* initializability of H is now proven by (the derivation uses property 7.24.4 implicitly):

$\mathbf{i}A^\circ \cup \mathbf{i}B^\bullet \cup \mathbf{i}D^\circ \cup \mathbf{i}E^\bullet$

$\rightsquigarrow$ { H_0 is strongly initializable }

$\mathbf{o}B^\bullet \cup \mathbf{o}C^\bullet$

$\rightsquigarrow$ { H_1 is weakly initializable; $\mathbf{o}C^\bullet = \mathbf{i}C^\circ$ }

$\mathbf{o}C^\circ \cup \mathbf{o}D^\circ \cup \mathbf{o}E^\bullet$

$\rightsquigarrow$ { $\mathbf{o}C^\circ = \mathbf{i}C^\bullet$; H_0 is weakly initializable }

$\mathbf{o}A^\circ$

Clearly, all symbols are initializable from the external inputs. Strong initializability which is proven similarly.

□

Theorem 7.31

Let T be a Tangram program. The activity graph associated $\mathcal{C} \cdot T$ is acyclic.

Proof Can be checked easily from the diagrams that depict the compilation function in Chapter 6.

□

Weak initializability of a handshake circuit is sufficient for practical purposes. Strong initializability allows the environment to postpone the initialization of the active input wires until all passive outputs wires have become low. For the set of undirected handshake components of Example 2.23 four-phase realizations can be designed that are strongly initializable. The simple strategy is then effective. The time needed to initialize a handshake circuit is proportional to the length of the longest directed path in the associated activity graph. In practice this amounts to less than a micro second for current CMOS realizations.

A more general initialization strategy

For the handshake components needed for the implementation of Tangram strongly initializable realizations exist (4-phase). However, for a few components weakly initializable realizations exist, that are significantly cheaper than their strongly initializable counterparts. These cheap variants often have properties that are useful for more elaborate initialization strategies. For example, for $TRF_{T^\cdot}(a^\circ, b^\bullet, c^\bullet)$ a very cheap 4-phase realization exists (cf. Example 7.22) that satisfies

$$\mathbf{i}a^\circ \rightsquigarrow \mathbf{o}b^\bullet \ \wedge \ \mathbf{i}b^\bullet \rightsquigarrow \mathbf{o}c^\bullet \ \wedge \ \mathbf{i}c^\bullet \rightsquigarrow \mathbf{o}a^\circ$$

This implies that port $b^\bullet$ must be initialized before $c^\bullet$ can be initialized. Acyclicity of the associated activity graph of a handshake circuit is then insufficient for weak initializability. However, depending on the initialization properties of the components involved, specific classes of weakly initializable handshake circuits may exist. It can, for instance, be proven that with the above transferrer, compiled handshake circuits of full Tangram are still weakly initializable.

7.7 Testing

Introduction

Fabrication of ICs introduces defects on the surface of the IC, such as spurious blobs of metal, impurities in the oxide layers, silicon-crystal defects, and cracks

in wires. These defects cannot be avoided completely and have a density of a few per cm^2 of IC area. Unfortunately, they may cause malfunctioning of the circuit. The fraction of defect-free chips for a given IC technology depends mostly on the size of the chip and ranges from over 90 % for chips of a few mm^2 to less then 10 % for a large IC of, say, 2 cm^2.

The main purpose of testing is *fault detection*: the discrimination between correctly manufactured circuits and faulty circuits. Another purpose is that of *fault location* [Fuj85]. This section only addresses the former.

An effective test procedure must make assumptions on how defects affect the circuit behavior. The set of assumptions is commonly referred to as the *fault model*. A popular fault model is the so-called *stuck-at* fault model, which models defects that prevent a wire to be pulled-up from a low state (stuck-at 0) or to be pulled-down from a high state (stuck-at 1). It must be noted that the stuck-at model only addresses those wires that connect logical gates (cf. VLSI operators, Section 7.5). More elaborate fault models also include e.g. bridging faults (spurious connections between two wires) and crosspoint faults (redundant transistors). Despite its limitations, the stuck-at model is widely used.

Testing of asynchronous circuits received little attention in literature. The subject is suggested to be difficult ([Fuj85], page 81):

> Test generation is much more difficult for asynchronous circuits than for synchronous circuits, because of races, hazards, or oscillations.

Also, the class of asynchronous circuits considered is usually restricted. Recent work [DGY90,BM91,MH91] suggests that for asynchronous circuits that are "sufficiently" delay insensitive, the prospects for testability are fairly promising.

The purpose of this section is to show that testing of CMOS realizations of handshake circuits is viable and that the costs of testing can be kept relatively modest. First we address the issue of the generation of test traces: traces that can be used to detect faults. This is based on the stuck-at model, restricted to stuck-at faults on (gate) outputs. For a more general approach that also includes stuck-at faults on inputs see [MH91].

Unfortunately, the length of a test trace or the time to execute a test trace may grow exponentially with the size of the circuit. In order to control the costs of test-trace generation and execution, it is necessary to modify the circuit with the objective to reduce these costs. This so-called *testability enhancement* will be addressed at the handshake-circuit level and forms the second topic of this section.

Both topics are treated informally and rather sketchy. This reflects the immaturity of the discipline of asynchronous-circuit testing and the presence of open problems.

Test traces

Our analysis starts with considering only those wires that connect handshake components. Assume, without loss of generality, that all these wires are required to be low at an initial state of the circuit. Let a be such a wire, and let Q be the handshake component for which a is an input wire. Stuck-at faults on wire a may have quite different effects on the circuit behavior:

0. A stuck-at 0 on a does not interfere with the initialization of the circuit. (It may even speed up the initialization procedure.) A subsequent up transition $a\uparrow$, however, will never arrive at Q, as if it experiences an infinite delay. Component Q can therefore not participate in any trace that involves $a\uparrow$. In most cases (see below) this can eventually be observed externally by the inhibition of an output transition.

1. A stuck-at 1 on a prevents the correct initialization of Q and hence of the circuit. Unfortunately, this stuck-at may have the same effect on Q as a (premature) up transition on a. In general, not much can be said about the response of Q to such a premature transition. We assume, however, that Q is not able to participate in a subsequent handshake that involves $a\downarrow$.

In either case, the handshake circuit cannot participate in a trace that contains both $a\uparrow$ and $a\downarrow$.

An *internal test trace* is defined as a trace in $\mathbf{W}\cdot H$ that causes each channel wire to make an up and a down transition. An *external test trace*, or *test trace* for short, is a trace $t \in (\mathbf{e}H)^H$ that satisfies:

$$(\forall u : (u\lceil \mathbf{e}H) = t : u \text{ is an internal test trace})$$

The idea is that the behavior of a handshake circuit cannot display a test trace in the presence of a stuck-at fault. Given this definition of test trace, three important questions arise:

0. under which circumstances does a test trace exist?

1. how to compute a test trace?

2. can test traces be executed?

For many handshake circuits no test trace exists, as is reviewed below. Fortunately, for practical circuits escapes are often available.

0. Some wires never make a transition. For instance, wire a_1 of the repeater in Example 7.22. These wires are clearly redundant and should be removed. In the sequel we assume that circuits are not redundant.

1. Sometimes a wire makes at most one transition. For instance, wire a_0 of the repeater of Example 7.22 makes at most one up-transition. There are two ways to deal with this situation. One can either relax the definition of test trace and add circuitry to observe stuck-at faults on these wires, or one can modify the circuit such that it is able to execute a signaling sequence with both transitions (see below under test enhancement).

2. Even if every wire can make both transitions, not all wires need to be covered by a single trace. The way out here is to concatenate several traces, linked with initialization steps into a single test trace.

3. Divergences cause special problems. The occurrence of a divergence can easily be observed in a CMOS realization of a handshake circuit, by measuring the supply current in an (externally) quiescent state. However, since it is not possible in all cases to identify the wires involved in a divergence, a test trace may not exist in a divergent circuit.

4. A more serious problem may be that of internal nondeterminism. Some forms of internal nondeterminism, such as the circuits compiled from Tangram programs with uninitialized variables, are relatively innocent. Other forms of internal nondeterminism involve circuit redundancy. The resulting nondeterminism must then be restricted during test time. Clearly more research is needed for testing internally nondeterministic circuits.

In summary, a test trace does exist, provided that

- the circuit is not redundant, non-diverging, and internally deterministic;
- provisions are made to deal with wires that (would otherwise) make at most one transition;
- re-initializations are allowed.

An undirected handshake channel is tested after the completion of a 4-phase handshake. Testing of a double-rail encoded channel requires at least two 4-phase

handshakes, so that both wires of each wire pair make both transitions. But how about the wires internal to the handshake components?

It turns out that the handshake components of Example 4.35 can be realized such that the internal wires are covered by a test trace that tests the external wires. This also holds for the handshake components of Example 4.37, except for some of the binary operators, such as adders. For adders an additional handshake is necessary to fully cover the testing of the carry chain.

Given the above testability properties of handshake components, it is in many cases straightforward to compute a test trace from the handshake circuit and even from the original Tangram program.

Example 7.32

0. Example of a test trace for $BUF_2(a, c)$ (cf. Figure 1.4):

$$\triangleright^\circ : (a^\bullet?0\ ;\ c^\bullet!0\ ;\ a^\bullet?1\ ;\ c^\bullet!1)$$

This trace tests a ripple buffer of arbitrary capacity! The test time can be reduced by changing the order of $c^\bullet!0$ and $a^\bullet?1$.

1. A test trace of $WAG(a, c)$ (cf. Figure 1.5) is

$$\triangleright^\circ : (a^\bullet?0\ ;\ c^\bullet!0\ ;\ a^\bullet?0\ ;\ c^\bullet!0\ ;\ a^\bullet?1\ ;\ c^\bullet!1\ ;\ a^\bullet?1\ ;\ c^\bullet!1)$$

In order to test the two parallel paths in the handshake circuit, twice as many communications are required in comparison with the test trace of $BUF_2(a, c)$.

□

Note that a test trace also detects multiple stuck-at faults on outputs. Masking of one fault by another fault cannot occur.

Test-trace execution

Can a test trace be executed? That is, is it possible to force a correctly manufactured IC to display the behavior specified by the test trace? In a strict sense, this is seldomly possible, because of reordering of output transitions. If the question is interpreted "modulo reordering", there still is a problem: that of external nondeterminism.

For instance, the first N outputs of an N-place shift register (cf. Section 1.3) are unkown at test time. The resulting nondeterministic behavior is relatively

innocent, because the communication behavior at the port level is not affected. More erratic forms of external nondeterminism are hard to handle by existing test equipment.

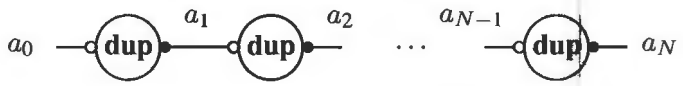
Given an executable test trace, ICs can be tested. An IC is free of stuck-at faults, if the complete test trace can be executed, The costs of testing are largely determined by the length of the test trace and the time needed to execute it. As a rule, shorter traces require less test time.

Testability enhancement

Testability enhancement of a circuit involves the modification of the circuit with the purposes of

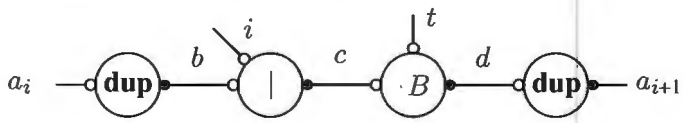
- reducing the length or execution time of a test trace;
- establishing the existence of a test trace.

Consider the duplicator chain of Example 3.24, consisting of N duplicators:



Completion of the handshake through port a_0 requires 2^N handshakes through port a_N. Clearly, the time to execute a test trace of the duplicator chain grows exponentially with the circuit size. The problem is not as artificial as it may seem: a watch is basically a set of counters that can be realized with duplicator chains. The circuitry that counts leap years must then also be tested!

The explosion in test time can be avoided by cutting the chain into two parts, more or less equal in size, and to test them independently. This can be realized by inserting a mixer and a *break* component B.



The behavior of component B is defined by (with m a local Boolean variable)

$$
\begin{array}{l}
m := \mathit{false};\ \#[\ \ [\ \ t^{\circ} : m := \neg m \\
\qquad\qquad\quad |\ \ c^{\circ} :\ \textbf{if}\ m \rightarrow \textbf{skip}\ [\!]\ \neg m \rightarrow d^{\bullet}\ \textbf{fi} \\
\qquad\quad]\ ]
\end{array}
$$

Variable m records whether B is in *test mode* . Initially, B is not in test mode. A handshake through t sets B in test mode, and a second handshake through t resets m. If B is in test mode, communications through c° are simply absorbed. If B is in not in test mode it acts like a connector.

The effect of B halfway in the duplicator chain is dramatic. The front subchain of length N **div** 2 can be tested in test mode in $2^{(N \textbf{ div } 2)}$ time units. The back subchain of length $N - (N \textbf{ div } 2)$ can be tested in normal mode roughly in the same amount of time. This results in an overall reduction by a *factor* of $2^{(N \textbf{ div } 2)-1}$.

In general, insertion of mixers and breaks makes it easier to obtain test traces in a systematic way. As illustrated above, it may also reduce the test time significantly.

An example of testability enhancement of the second kind, viz. one that helps to establish the existence of a test trace, is the following. A repeater can be equipped with a passive port that is used to (re-)set the repeater in test mode, similar to the break component in the duplicator chain. By modifying the behavior of the repeater such that in test mode it behaves like a connector, the wires connected to the passive port of the repeater can conveniently be tested. As a bonus, most handshake circuits will then in test mode complete the handshake through port $\triangleright^\circ$.

Chapter 8

In practice

Handshake circuits and the associated compilation method from CSP-based languages were conceived during 1986 at Philips Research Laboratories. A first IC (7000 transistors) was designed using experimental tools to manipulate and analyze handshake circuits (then called "abstract circuits") and to translate them into standard-cell netlists. The IC realized a subfunction of a graphics processor [SvB88] and proved "first-time-right" (September 1987). Extensive measurements hinted at interesting testability and robustness properties of this type of asynchronous circuits [vBS88].

Encouraged by these early results the emphasis of the research shifted from the design of the graphics processor to VLSI programming, compilation methods and tool design. Generalization and systematization of the translation method resulted in an experimental silicon compiler during spring 1990 [vBKR*91].

A second test chip has been designed and verified during the autumn of 1991. In addition to some test structures, the IC contains a simple processor, including a four-place buffer, a 100-counter, an incrementer, an adder, a comparator, and a multiplier in the Galois Field $GF(2^8)$. The Tangram program was fully automatically compiled into a circuit consisting of over 14 thousand transistors. Extensive testing and mearsuring demonstrated functional and structural correctness over a supply-voltage range from 1.2 Volt to 7.5 Volt.

Current work on the compiler aims at extending its input language Tangram [SvBB*91,KvBB*92] and improving the efficiency of the generated circuits. Most of the theory reported in this thesis evolved in conjunction with the work on the method and tools. This final chapter reports some of our practical experiences with VLSI programming and silicon compilation. It concludes with an appraisal of asynchronous circuits.

8.0 VLSI programming and compilation

Experiences with VLSI programming and compilation will be presented from a programmer's viewpoint. After an overview of the design tools, a benchmark program (that of a Compact Disc error decoder) is used to illustrate various VLSI programming and compilation issues.

Tool overview

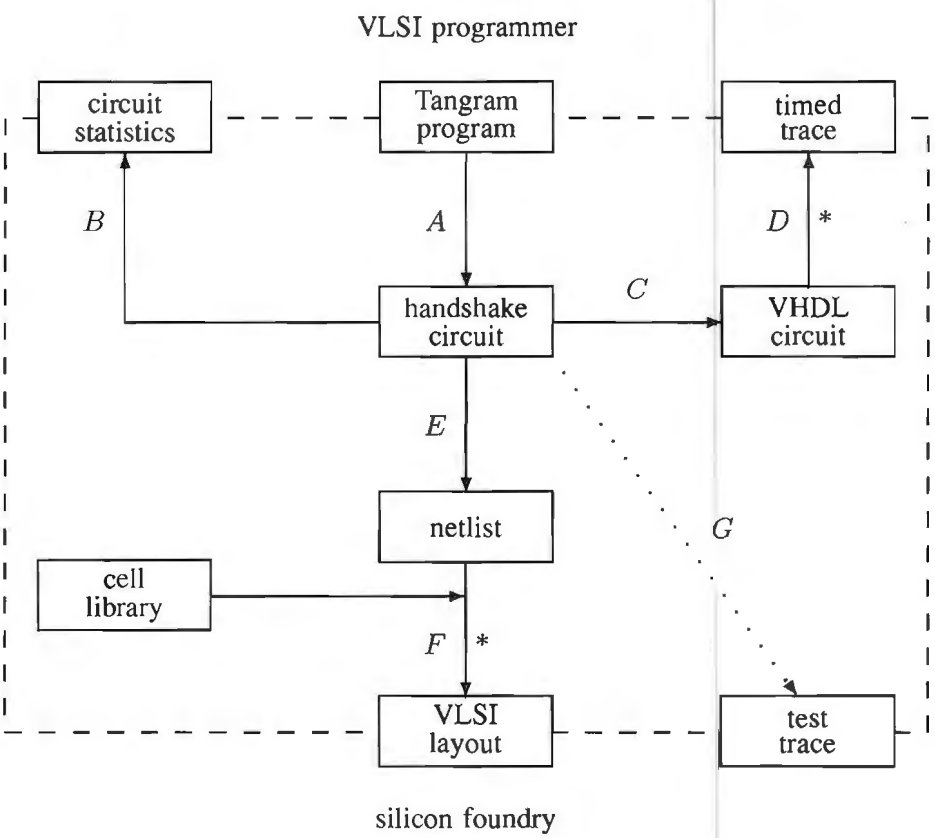


Figure 8.0: Overview of the main Tangram-compilation tools. Boxes denote design representations, arrows denote tools.

An overview of the design tools is depicted in Figure 8.0. Design representations are shown as boxes, tools as arrows. Arrows labeled with an asterisk

denote commercially available tools. The main Tangram-compilation tools are:

- *A*: a translator from Tangram to handshake circuits (text format);
- *B*: an analyzer of handshake circuits that produces statistics at the levels of handshake circuits, CMOS circuits and layout;
- *C*: a converter of handshake circuits into VHDL [LSU89] descriptions;
- *D*: a VHDL simulator that produces a trace with detailed timing information;
- *E*: a generator that expands the handshake circuit into a netlist of standard cells;
- *F*: a standard-cell layout package that performs placement of the standard cells and cell-to-cell routing according to the netlist;
- *G*: a test trace generator (under development).

The vehicle

The benchmark for this chapter is a simple error decoder with application in Compact Disc players (cf. Section 0.0). A precise description of this function can be found in [KvBB*92]. A global description of the error decoder is the following. The decoder receives code words of 32 symbols (of 8 bit), of which four are designated as parity symbols. These parity symbols allow for the correction of two erroneous symbols. The benchmark program can only locate single errors. For each code word the decoder produces an error status (0, 1 or more errors) and, in the case of a single error, an error location and an error value. The actual correction is not performed by the decoder. Code words arrive at a rate of one per 70 μseconds.

VLSI programming

A Tangram program for the decoder can be found in [KvBB*92]. Schematically it can be described by

$$\#[x := input(a);\ \ s := syndrome(x);\ \ e := search(s);\ \ c!e]$$

The incoming code word (through port a) is stored in variable x by function *input*. Function *syndrome* then computes the syndrome of x, which is stored

in s (a tuple of four symbols). The syndrome contains the error information in an implicit form. Function *search* makes this information explicit in variable e, using a linear search. The number of steps in this search varies between 0 (for correct words) and 32 (in the case of more than one error). Finally, the error information is output through port c. An important improvement on the above program is obtained by computing the syndrome "on the fly" and thus avoiding the costly storage of the incoming code word.

For the detailed program of the decoder the subset of Tangram of Chapter 5 is insufficient. The applied arithmetic in the Galois Field $GF(2^8)$ requires provisions for the definition of the appropriate types and associated operations. These provisions include tuple construction and selection, type casting, and type fitting [SvBB*91]. The structure of the program benefits from function and procedure definitions. Sharing of a number of these procedures (cf. Section 1.5) avoids duplication of circuitry with hardly a penalty in performance.

The program text consists of 68 lines, divided over three paragraphs that are more or less equal in size. The first paragraph contains type and function definitions for the Galois Field arithmetic. The second paragraph consists of declarations of variables, functions and procedures and the third paragraph is the detailed version of the above command.

The transparency of the translation method plays an important role in VLSI programming. As the coarse performance can be read directly from the Tangram text, the selection of the above algorithm may be justified at an early stage. Also the choice of the amount of parallelism in the syndrome computation and in the error search is guided by the observation that most elementary operations in the Galois Field are very cheap.

Simple analysis [KvBB*92] shows that the decoder takes in the worst case about 20 μseconds per codeword, which clearly suffices. In the case of stricter performance requirements, the following program for the decoder may be considered:

$$\begin{array}{ll} & \#[x := input(a);\ \ s := syndrome(x);\ \ b!s] \\ \| & \#[b?r;\ \ e := search(r);\ \ c!e] \end{array}$$

It consists of two parallel processes: one for computing the syndrome and one for searching the error. The type of the internal channel b is a tuple of four symbols. The resulting form of pipelining is akin to that of the ripple buffer in Section 1.1. The throughput of the above program is approximately twice as large compared with the first decoder program. Aspects of a more detailed comparison of both decoder programs recur below.

Compilation to handshake circuits

The compilation of Tangram programs has been implemented according to an extension of the method of Chapter 6. The compiler translates directly to complete 4-phase handshake circuits, and many of the optimizations of Section 7.1 and 7.2 are included. The compiler generates a handshake circuit in a simple textual format.

The compiler uses the handshake components of Examples 2.23 and 4.37. The extensions of Tangram mentioned above require only a few extra handshake components.

The compiled decoder consists of 523 handshake components, including 174 connectors. The pipelined version contains 741 handshake components, of which 244 are connectors.

Simulation

Additional confidence in the correctness of a Tangram program can be gained from simulation of the compiled handshake circuit. We have based our simulation tools on a commercially available VHDL simulator. A simple program translates the handshake circuit into an equivalent VHDL architecture [LSU89]. Together with a library of VHDL models for the various handshake components this provides access to simulation tools used in main-stream VLSI design. A major advantage over a specific handshake-circuit simulator is that the above setup also allows interfacing to other circuits, including clocked ones, within the VHDL framework.

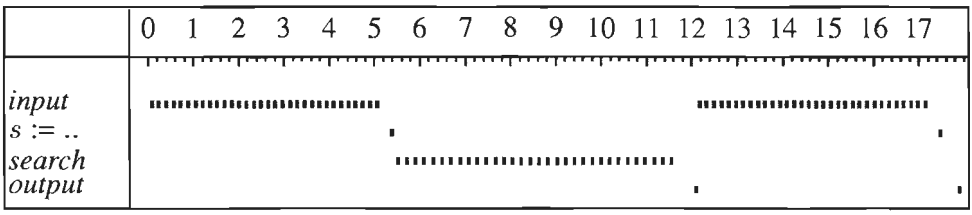


Figure 8.1: Timing diagram of the standard error decoder. The time scale is in microseconds.

Both decoder programs have been simulated with input ports connected to data files containing several code words. By inspection of the files connected to the output ports the correct functional behavior was verified. An alternative to

file I/O is to model the environment by a Tangram program and to inspect the data transferred along internal channels.

In addition to functional verification, simulation proves useful in analyzing the detailed timing behavior of the compiled circuit. The VHDL models of the handshake circuits have been characterized with regard to timing, based on the timing of its constituent operators and conservative estimations of the average wire capacitances. An interactive post processor of the generated timing data has been used to generate the timing diagrams Figures 8.1 and 8.2.

Figure 8.1 displays the timing behavior of the standard decoder.. The square dots are actually short line segments, each indicating a handshake interval, marked from the first phase to the fourth phase. The top line shows the handshakes of the input port, 32 per code word. The second line depicts an intermediate step between the input phase and the search. The third line shows the search: for the first code word (with two errors) the search takes 32 steps, and for the second code word (correct) 0 steps. The output of the error information is on line four. An incorrect code word takes at most 12 μseconds, a correct word about 6 μseconds.

The timing diagram of the pipelined error decoder in Figure 8.2 is markedly different. The same channels have been monitored as in the standard decoder, with the intermediate step replaced by the internal communication along channel *b*. It is clearly visible that the input of the second code word is in parallel with the error search of the first code word.

The throughput of the pipelined decoder is indeed twice that of the standard decoder, viz. one code word per 6 μseconds. (The simulation interval of 17 μseconds in the timing diagram of Figure 8.2 left thus plenty of room for the decoding of a third code word.) The elapsed time for an incorrectable code word has also improved, because less overhead is involved in the sharing of procedures.

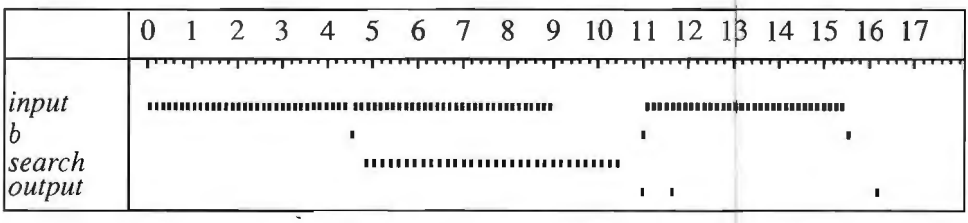


Figure 8.2: Timing diagram of the pipelined error decoder. The time scale is in microseconds.

For critical designs, the timing model may be too conservative. Significant improvements, however, can only be obtained when the handshake circuit is "back annotated" with wiring capacitances from the layout (cf. simulation of power consumption at the end of this section).

VHDL simulation provides detailed feedback, but is too slow for heavy work. The simulation of large Tangram programs and large input files, as required for e.g. digital video signal processing, requires more powerful simulation tools. A solution may be to compile Tangram code directly to VHDL, or to an ordinary programming language such as C or Pascal. At the expense of the accuracy of the timing information, several orders of magnitude may be gained in simulation speed.

Circuit statistics

Comparison of different VLSI programs and optimization of the chosen program with regard to layout area requires feedback about circuit and layout costs. Unfortunately, the automatic generation of relatively small layouts may take several hours. Larger layouts may involve interactive floorplanning and optimization, and their generation may then take a few days or weeks, or even longer.

A quick form of feed back is a table of statistics computed from the handshake circuit. These statistics include the area covered by standard cells, but ignores the wiring area, which is sometimes a serious limitation. Excerpts from the statistics generated for the standard decoder are displayed in Table 8.0.

The first paragraph reports the MOS transistor count and the area occupied by the standard cells. These quantities are detailed by function, based on the role of the handshake component in the computation (e.g. sequencer: control, mixer: communication, binary operator: logic, and variable: memory). These functional profiles vary considerably from one Tangram program to another.

The second paragraph presents the standard cell counts. It is confined to the six most frequently used cells, accounting for about two thirds of the transistors and cell area.

The third paragraph gives an estimation of the area of the standard-cell part of the layout. Here it is assumed that the routing channels occupy the same area as the standard cells. For this example this is quite accurate, as we shall see later.

The pipelined decoder counts 11376 transistors and 989 cells, occupying a cell area of 1.5 mm^2. The estimated core area is 3.0 mm^2, which will turn out to be rather optimistic.

Part	#MOSTs	cell area [mm^2]
Decoder	7292 [100.0 %]	0.931 [100.0 %]
control	1010 [13.9 %]	0.121 [13.0 %]
communication	938 [12.9 %]	0.142 [15.3 %]
logic	2772 [38.0 %]	0.252 [27.0 %]
memory	2568 [35.2 %]	0.415 [44.5 %]

Cells	#cells	#MOSTs
Decoder	610 [100.0 %]	7292 [100.0 %]
C2	26 [4.3 %]	312 [4.3 %]
C3	37 [6.1 %]	444 [6.1 %]
EQL	122 [20.0 %]	2440 [33.5 %]
OR2	130 [21.3 %]	780 [10.7 %]
S	34 [5.6 %]	612 [8.4 %]
VAR1	54 [8.9 %]	864 [11.8 %]

estimated core area	:	1.9 [mm^2]
estimated transistor density:		3916 [mm^-2]

Table 8.0: Circuit statistics for the standard decoder.

The problem here is how to take the wiring area into account, without laying out the complete circuit. The ratio core-area/cell-area has been observed to vary between 1.8 and 3. Conclusion: statistics are helpful with the selection among alternatives and the tuning of a final program, but layout generation is necessary for an accurate assessment of the circuit size of a VLSI program.

Layout generation

The generation of a layout from a handshake circuit involves two steps. First the handshake circuit is expanded into a netlist of standard cells. These standard cells are then placed and interconnected according to the connectivity information described in the netlist.

We have developed a library of twenty-odd standard cells. Most standard cells implement a single VLSI operator; some implement a combination of a few

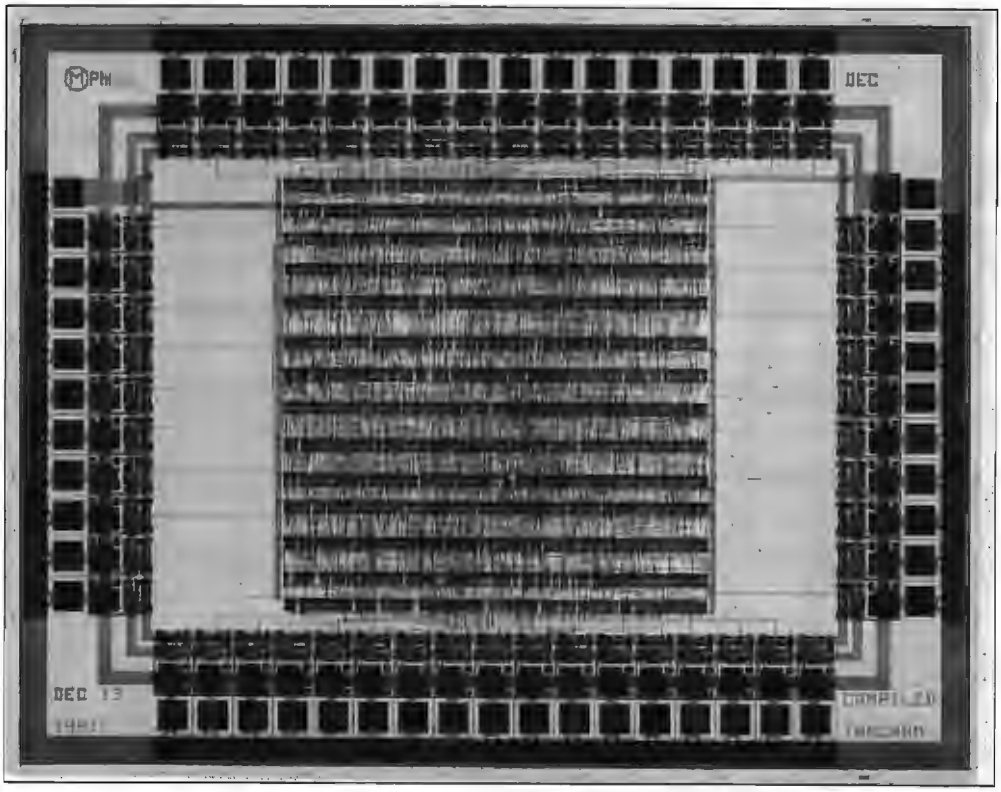


Figure 8.3: Layout of the standard error decoder.

operators in order to economize on layout area.

The expansion of handshake components also resolves their parameters (e.g. word width, number of read ports). Placement and routing are performed by commercially available layout tools.

Figure 8.3 shows a layout of the standard error decoder, synthesized fully automatically from the Tangram program. The core area (standard cells + inter-cell routing) measures 2.2 mm^2 in a 1.2 μ double-metal CMOS process.

A layout of the pipelined error decoder is shown in Figure 8.4. The difference between the measured core area (4.0 mm^2) and the estimated core area (3.0 mm^2) is visible in the relatively wide routing channels. This is partly a consequence of the 32 $\times$ 2 wires for realizing channel b. Both layouts contain 55 pads: 22 inputs, 31 outputs, 1 power and 1 ground.

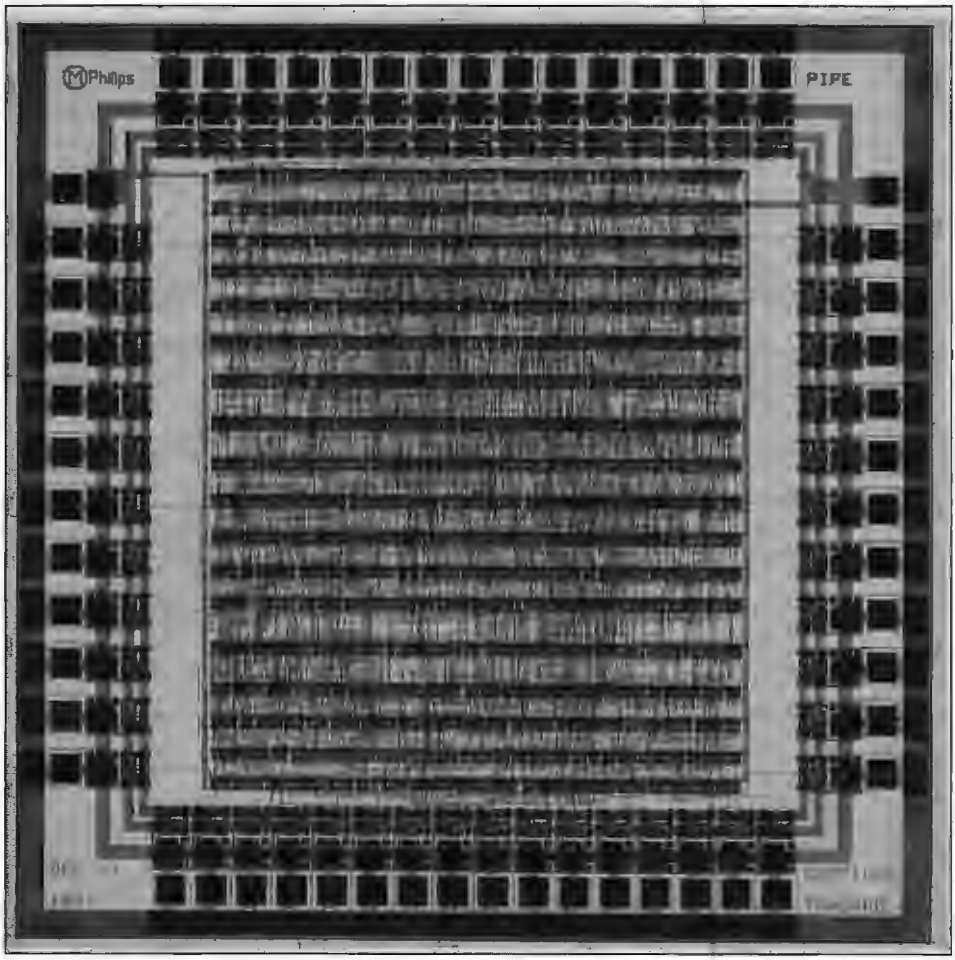


Figure 8.4: Layout of the pipelined error decoder.

Simulation of energy consumption

In addition to circuit size and circuit speed, energy consumption is a third cost/performance indicator. The energy consumed and dissipated by the decoder is analyzed next.

The number of steps in the error search varies from 0 to 32. The total time to decode a word by the standard decoder varies correspondingly between 6 and 12 μseconds (cf. Figure 8.1). We may therefore expect to find a variation in energy consumption as well (cf Section 0.1).

By good approximation the *energy* consumption of a static CMOS circuit can be calculated by summing over all wires the quantity $\frac{N}{2}CV^2$, where N is the number of transitions on that wire, C the capacitance of the wire (including that of the transistor gates connected to it), and V the supply voltage. Accurate values of all capacitances can be extracted from the layout. The number of transitions can be obtained for given input stimuli by means of switch level simulation. The above summation then yields the energy required for the computation. Division by the specified (or simulated) computation time results in an indication for the power consumption of the circuit.

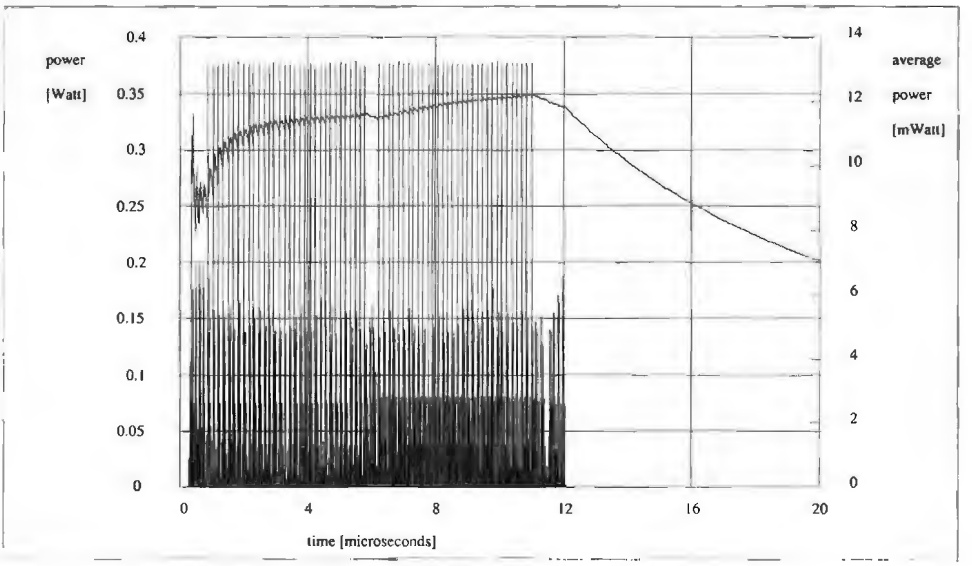


Figure 8.5: Power consumption of the standard error decoder for an incorrectable code word.

Results of such simulations are given in Figures 8.5 and 8.6. The spikiness has

no physical meaning, but is characteristic for event-driven simulations, which condense all energy of simultaneously occurring transitions into a single time instant.

Figure 8.5 shows the simulation results for an incorrectable codeword. The two main phases, viz. input with on-the-fly syndrome computation (32 steps) and error search (again 32 steps) are clearly visible. The circuit activity after the search includes output of the error information. Then the circuit is quiescent: *no* power is consumed until a next code word is offered. No energy is dissipated in clock distribution or in a controller that issues "skip" instructions.

The smooth curve represents the average power: the energy consumed so far divided by the elapsed time. The power consumption is 2.5 mWatt, assuming a rate of one *incorrectable* code word per 70 μseconds.

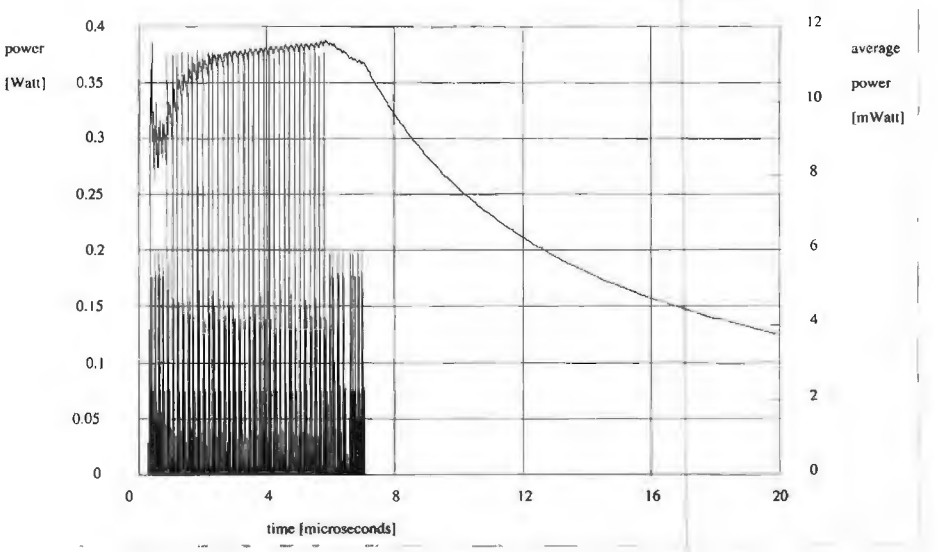


Figure 8.6: Power consumption of the standard error decoder for a correct code word.

For correct code words, quiescence is reached soon after the syndrome computation, as shown in Figure 8.6. The power consumption is then only 1.2 mWatt, assuming a rate of one *correct* code word per 70 μseconds. This example nicely shows that asynchronous circuits consume energy only when and where needed.

Test-trace generation

A test trace consisting of 3 code words (with 0, 1 and 2 errors) has been verified to test for all stuck-at faults in both decoders. The trace has been generated by hand[0] from the Tangram program text. The transparent compilation rules and some simple properties of the circuit realizations of the handshake components made this a feasible task. The fault coverage was checked by switch-level simulation. It may be interesting to note that a single code word covers already 97 % of the possible stuck-at faults.

8.1 An appraisal of asynchronous circuits

In Chapter 7 we have seen that handshake circuits are most naturally realized by asynchronous circuits. However, the overwhelming majority of today's VLSI circuits are synchronous. Are there good reasons to educate and train a new generation of designers in designing asynchronous circuits and VLSI programming? It is hard to tell. The balance of pros and cons is mixed. This section aims at reviewing this balance.

Research on asynchronous circuits is booming. With few exceptions, this research is carried out in academic research institutes. We may expect significant progress in the understanding of these circuits, and we may hope for further improvements in their cost and performance. Synchronous-circuit design has a respectable tradition of several decades, whereas asynchronous circuit design is making its first steps along the learning curve. The appraisal below is therefore only a 1992 snapshot.

A major problem in comparing asynchronous and synchronous circuits is the large variety in their characteristics and realizations. For both synchronous and asynchronous circuits there exist different architectures, different (detailed) timing disciplines and different building blocks. Quantitative comparisons are therefore hard to make.

The following aspects will be reviewed: ease of design, circuit speed, robustness, testability, circuit size, and energy consumption.

Ease of design

A clock is an artefact. It has been introduced to solve timing problems at the *circuit* level, viz. the controlled usage of latches and the avoidance of critical

[0] Marly Roncken's

races. By the evolution of VLSI circuits, the clock gradually became an important *system*-level design issue.

A choice for a single clock frequency in a VLSI system has many repercussions on the modularity of the system, as well as on its performance. In a synchronous circuit there is an excess of synchronization among subcircuits, viz. at the end of each clock period. In an asynchronous circuit synchronization is reduced to its (functional) minimum. This leads to a high degree of modularity, allowing modules to be designed and optimized independently [Sei80,Sut89,MBM90]. One of the earliest projects in hardware design that addressed the issue of modularity is that of 'Macro modules' [OSC67].

Furthermore, there is no need to design the circuits for the basic functions (shifting, addition, multiplication, etc.) under the restriction that they can be 'evaluated' well within the clock period. In an asynchronous circuit, the slowness of a multiplier may be compensated by a fast shift or transfer step in a computation.

Problems with clock distribution and clock skew are of course absent in asynchronous circuits. A gain in design productivity is also expected from VLSI programming and compilation techniques described in this thesis.

Circuit speed

Less synchronization among subcircuits makes circuits faster. Consider for instance two parallel processes, one that performs an addition followed by a multiplication, the other performs both operations in the opposite order. (Symbolically: "$(+;\ *)\ ||\ (*;\ +)$".) Furthermore, let the time required for a multiplication m be considerably larger than the time required for an addition a. Asynchronous execution of this process requires $m + a$ time units. Synchronization of the two parallel processes at the semicolon, as happens in clocked circuits, increases this to $2m$ time units.

Asynchronous circuits also allow one to take advantage of the data-dependence of computation times. A well-known example is that of the n-bit parallel adder in [Sei80]. The average addition time is proportional to the average carry-ripple path ($O(\log n)$), whereas the worst case addition time is proportional to n. In synchronous circuits an $O(\log n)$ response time can only be obtained with additional carry-acceleration circuits. Then the worst-case and average-case performances are equal.

Circuit speed has been an important motivation for asynchronous circuits in [KTT*88,KTT*89,MBM90].

The asynchronous circuits realized from handshake circuits have another

speed advantage that stems from the extreme form of control distribution. e.g. a sequencer that activates two transferrers avoids the large timing overheads common to central controllers.

Although the number of synchronizations is smaller in asynchronous circuits, their explicit nature has its price. Especially in complete four-phase signaling, the associated overheads may easily outweigh the above advantages for regular computations. Quick four-phase refinements and perhaps two-phase refinements may help here.

Robustness and reliability

Insensitivity to delay seems to relate to insensitivity to variations in IC-processing parameters and operating conditions. These issues have hardly been researched, and will only be touched upon by means of a few examples.

A decrease in the width of the polysilicon tracks in a CMOS circuit yields faster transistors (shorter channels) and slower wires (higher resistance). In synchronous circuits this may result in violations of the set-up and hold times of the latches. In speed-independent circuits however, variations in the widths of polysilicon tracks may influence the circuit's performance, but not its correct operation.

Measurements of our test silicon [vBS88] showed a high degree of robustness with respect to variations in power supply voltage. A large self-inductance in the power-supply wires caused the supply voltage to drop below 0 volt for brief periods of time, without affecting the functional correctness of the circuit. In [MBL*89] an asynchronous microprocessor is reported to operate with a supply voltage in the range of 0.35 to 7 Volt.

On a related note, it has been reported that fundamental reliability problems that come with the synchronization of independently clocked circuits [Ano73, CM73] can be avoided in delay-insensitive circuits [Sei80]. The occurrence of glitches (the occurrence of a metastable state) does not lead to anomalous behavior, because subsequent computations are simply delayed until metastability has been resolved.

Testability

There are indications that the testing of asynchronous circuits is feasible. It is even expected that speed-independent circuits may be simpler to test, because stuck-at faults can be observed through deadlock [BM91,MH91]. This feature also highly simplified the generation of tests for our graphics chip [vBS88] and

the decoder of Section 8.0. The progress in research towards automatic generation of test traces is nevertheless slow. This presumably reflects that this problem is, "combinatorially speaking", a hard one.

Circuit area

In most cases the layout area is dominated by circuits that store, communicate or process data; circuitry for control covers usually less than 15 % of the area (cf. the statistics of the decoder in Section 8.0). With Double-Rail encoding, which requires two wires per bit, we may then not expect to obtain circuits that can compete in silicon area. What should we think of this roughly 100 % area overhead?

Firstly, there are compensations. Availability of two wires per bit simplifies some operations: e.g. the Boolean complement is obtained by simply crossing the two wires. Also, as we have seen with the shift registers in Section 1.3, slave latches can be saved selectively, whenever allowed by the performance requirements. Another form of compensation is realized by the extreme form of control distribution intrinsic to the applied form of syntax-directed translation. This leads to fewer and *shorter* wires. An example of an incidental compensation is the absence of carry acceleration circuitry for applications where the *average* throughput matters.

Secondly, other delay-insensitive codes that require less wires may be considered (cf Section 7.4). Clever encoder and decoder circuits may reduce the costs of (de-)multiplexing, synchronizing and long-distance communications.

Thirdly, and presumably most significantly, compromises with regard to delay insensitivity are required to arrive at competing circuits. Natural candidates for first experiments in this respect are: off-chip communication, Random-Access Memories (RAMs) and Read-Only Memories (ROMs). Existing standards and the availability of well-engineered embedded memories make it impractical to insist on delay-insensitive circuit realizations in these cases. In many ICs the input/output circuitry (including bonding pads) together with the memories account for well over half the circuit area.

Single-rail encoding for the remaining circuitry has been proposed in [Sut89]. Ultimately this may result in circuits that are even smaller than their synchronous counterparts. Avoidance of interference (cf. Section 0.1), however, requires delay circuitry [KTT*88,KTT*89] or completion detection by using alternative circuit techniques [MBM90]. On a more speculative note, it may be interesting to investigate circuit techniques that encode three states in a single wire.

Unfortunately, compromises with regard to delay insensitivity may jeopardize testability and complicate layout design.

Finally, VLSI programming and automatic silicon compilation allow the designer to construct and compare many alternative solutions for the specification at hand. By exploring a large portion of the 'solution space' he may expect to find cheaper designs compared with current VLSI-design methods.

Energy consumption

Self-timed circuits consume potentially less energy than clocked circuits[1]. There are several reasons for this. Absence of interference (see Section 0.1) and other transient phenomena such as hazards make each transition productive.

Also, there is no dissipation in clock signals. In high-throughput applications, such as in video-signal processing, the distribution of a high-frequent clock may account for well over 20 % of the total power consumption.

Furthermore, control distribution leads to a high degree of locality, thus avoiding the power consumption in central controllers and the long wires from and to these controllers.

Viewed differently, one may say that a self-timed circuit only consumes energy where and when needed. The circuit compiled from a Tangram procedure or function consumes *no* energy when it is not invoked. The error decoder of Section 8.0 consumes energy only during the first 12 μseconds of the 70 μseconds available. For correct code words the required energy is even only half that required for incorrect code words.

To what extent these potential advantages can be realized highly depends on the chosen handshake refinements and data encoding.

As a rule, energy savings increase with a decrease in regularity of the computation. Error decoding, where the work load depends on the correctness of the code words, is a nice example of an irregular computation. Circuits that stand by for most of the time, but have to respond to exceptional conditions represent another example (e.g. a processor triggered by key-board inputs).

This observation concludes both the appraisal of asynchronous circuits and the last chapter of this monograph.

[1] Here we assume a circuit technology without static dissipation, such as CMOS.

Appendix A

Delay insensitivity

Introduction

In [Udd84,Udd86,Ebe89,UV88] the notions delay insensitivity, independent alphabet and absence of computation interference have been defined for *directed processes*. In this section we investigate to what extent these notions apply to handshake processes.

Definition A.0 (directed process)

A directed process T is a triple $\langle \mathbf{i}T, \mathbf{o}T, \mathbf{t}T \rangle$, in which $\mathbf{i}T$ and $\mathbf{o}T$ are disjoint sets of symbols and $\mathbf{t}T$ is a non-empty, prefix-closed subset of $(\mathbf{i}T \cup \mathbf{o}T)^*$.

□

A handshake process is not a directed process: the alphabet of a handshake process has more structure and the trace set is not prefix closed. However, to every handshake process P there *corresponds* a directed process, viz. $\langle \mathbf{i}P, \mathbf{o}P, \mathbf{t}P^{\leq} \rangle$.

All port structures in this appendix have no internal ports.

Composability

Composability of traces captures the notion that symbols communicated between processes arrive no earlier than they were sent. Consider directed processes P and Q such that $\mathbf{i}P = \mathbf{o}Q$ and $\mathbf{o}P = \mathbf{i}Q$. Let $s \in \mathbf{t}P$ and $t \in \mathbf{t}Q$. Composability restricts the way how the pair (s, t) *may* evolve from $(\varepsilon, \varepsilon)$. Let $a \in \mathbf{i}Q$ (and therefore $a \in \mathbf{o}P$). Then ε is composable to a, but the converse is not true,

because a must be sent by P before it can be received by Q. Similarly, for $b \in \mathbf{o}Q$, we have b composable to ε. Also, trace s is composable to ta if s is composable to t and a is on its way, i.e. $len \cdot (t\lceil a) < len \cdot (s\lceil a)$. With this introduction we have prepared ourselves for the following definition.

Definition A.1 (composability)

0. Let I and O be two disjoint sets of symbols. $\mathbf{C}_{(I,O)}$ is the smallest binary relation on $(I \cup O)^*$ such that for all symbols $a \in I$, symbols $b \in O$, and traces $s, t \in (I \cup O)^*$:

$$\begin{array}{lcl} \varepsilon\ \mathbf{C}_{(I,O)}\ \varepsilon & = & \mathit{true} \\ \varepsilon\ \mathbf{C}_{(I,O)}\ tb & = & \varepsilon\ \mathbf{C}_{(I,O)}\ t \\ sa\ \mathbf{C}_{(I,O)}\ \varepsilon & = & s\ \mathbf{C}_{(I,O)}\ \varepsilon \\ s\ \mathbf{C}_{(I,O)}\ ta & = & s\ \mathbf{C}_{(I,O)}\ t \ \wedge\ len \cdot (t\lceil a) < len \cdot (s\lceil a) \\ sb\ \mathbf{C}_{(I,O)}\ t & = & s\ \mathbf{C}_{(I,O)}\ t \ \wedge\ len \cdot (t\lceil b) > len \cdot (s\lceil b) \\ sa\ \mathbf{C}_{(I,O)}\ tb & = & sa\ \mathbf{C}_{(I,O)}\ t \vee s\ \mathbf{C}_{(I,O)}\ tb \end{array}$$

Sets I and O contain the input and output symbols respectively. When (I, O) are clear from the context, $\mathbf{C}_{(I,O)}$ will be shortened to $\mathbf{C}$.

1. Let A be a port structure. $\mathbf{c}_A$ is a relation on A^H and is defined by

$$s\ \mathbf{c}_A\ t = s\ \mathbf{C}_{(\mathbf{i}A,\mathbf{o}A)}\ t$$

i.e. the restriction of $\mathbf{C}$ to A^H.

□

Relation $\mathbf{C}_{(I,O)}$ is the converse of the composable relation introduced by [Udd84, UV88]. Relation $\mathbf{c}$ is a preorder on A^H. Consequently, we shall write $S^{\mathbf{c}}$ to denote the composability closure of S and $(\mathbf{c})\cdot S$ to denote the composability closedness of S (cf. Section 2.1). Both operators are lifted to handshake structures in the obvious way. The composability relation plays a central role in much of the theory on delay insensitivity. We shall therefore first analyze a number of its properties.

property A.2

Let A be a port structure.

0. $s, t \in A^H \wedge a \in \mathbf{i}A \wedge s\ \mathbf{C}\ ta \ \Rightarrow\ ta \in A^H$
1. $s, t \in A^H \wedge b \in \mathbf{o}A \wedge sb\ \mathbf{C}\ t \ \Rightarrow\ sb \in A^H$

□

Relations $\mathbf{r}$ and $\mathbf{c}$ are related by:

Property A.3

For $s, t \in A^H$ we have $s\ \mathbf{r}_A\ t\ =\ s\ \mathbf{c}_A\ t \wedge (\#_s = \#_t)$, where $\#_t$ denotes the bag of symbols of trace t.

□

Another way of relating $\mathbf{r}$ and $\mathbf{c}$ is suggested by "welcoming the traveling symbols":

Property A.4

For port structure A and $t, u \in A^H$ we have:

$$\begin{array}{lll} u\ \mathbf{c}\ t\ = & (& \exists t', u' \\ & : & (\#'_t = \#_{(u\lceil \mathbf{i}A)} \setminus \#_{(t\lceil \mathbf{i}A)}) \wedge (\#'_u = \#_{(t\lceil \mathbf{o}A)} \setminus \#_{(u\lceil \mathbf{o}A)}) \\ & : & uu'\ \mathbf{r}\ tt' \\ &) & \end{array}$$

Proof We derive:

$u\ \mathbf{c}\ t$

$=$ { Property A.2.0; definition of $\mathbf{c}$ }

$(\exists t' : \#_{t'} = \#_{(u\lceil \mathbf{i}A)} \setminus \#_{(t\lceil \mathbf{i}A)} : u\ \mathbf{c}\ tt')$

$=$ { Property A.2.1; definition of $\mathbf{c}$ }

$(\exists t', u' : (\#_{t'} = \#_{(u\lceil \mathbf{i}A)} \setminus \#_{(t\lceil \mathbf{i}A)}) \wedge (\#_{u'} = \#_{(t\lceil \mathbf{o}A)} \setminus \#_{(u\lceil \mathbf{o}A)}) : uu'\ \mathbf{c}\ tt')$

$=$ { Property A.3, using $len \cdot uu' = len \cdot tt'$ }

$(\exists t', u' : (\#_{t'} = \#_{(u\lceil \mathbf{i}A)} \setminus \#_{(t\lceil \mathbf{i}A)}) \wedge (\#_{u'} = \#_{(t\lceil \mathbf{o}A)} \setminus \#_{(u\lceil \mathbf{o}A)}) : uu'\ \mathbf{r}\ tt')$

□

Relations $\mathbf{c}$, $\mathbf{r}$ and $\mathbf{x}$ are related in a remarkable way for prefixed closed handshake structures, as shown in the next theorem.

Theorem A.5

For handshake structure S, such that $(\leq) \cdot S$ we have:

$$(\mathbf{c}) \cdot S = (\mathbf{r}) \cdot S\ \wedge\ (\mathbf{x}) \cdot S$$

Proof Let $t \in \mathbf{t}S$ and $u \in (\mathbf{p}S)^H$. We derive for $LHS \Leftarrow RHS$:

$t \in \mathbf{t}S \wedge u \; \mathbf{c} \; t$

$=$ { Property A.4 }

$(\exists t', u' : t \in (\mathbf{i}A)^* \wedge u \in (\mathbf{o}A)^* : uu' \; \mathbf{r} \; tt' \wedge t \in \mathbf{t}S)$

$\Rightarrow$ { $(\mathbf{x})\cdot S$ }

$(\exists t', u' : t \in (\mathbf{i}A)^* \wedge u \in (\mathbf{o}A)^* : uu' \; \mathbf{r} \; tt' \wedge tt' \in \mathbf{t}S)$

$\Rightarrow$ { $(\mathbf{r})\cdot S$; calculus }

$(\exists u' : u' \in (\mathbf{o}S)^* : uu' \in \mathbf{t}S)$

$\Rightarrow$ { $(\leq)\cdot S$ }

$(\exists u' : u' \in (\mathbf{o}S)^* : u \in \mathbf{t}S)$

$\Rightarrow$ { calculus }

$u \in \mathbf{t}S$

$LHS \Rightarrow RHS$ follows readily from the definitions of $\mathbf{c}$, $\mathbf{r}$ and $\mathbf{x}$.

□

Corollary A.6

For handshake process P we have $(\mathbf{c})\cdot P^{\leq}$.

□

Equipped with the above property of handshake processes we are ready to analyze the delay insensitivity of handshake processes.

Delay insensitivity

Delay insensitivity of directed processes has been defined in many ways [Udd84, Udd86,Ebe89,UV88]. The cited definitions are all provably equivalent.

Definition A.7 (delay insensitive)

0. A directed process $\langle I, O, T \rangle$ is delay insensitive if

$$
\begin{array}{l}
(\quad \forall s, t, a \\
: \quad s \in T \wedge t \in T \\
: \quad (a \in I \wedge s \, \mathbf{C}_{(I,O)} \, ta \Rightarrow ta \in T) \wedge (b \in O \wedge sb \, \mathbf{C}_{(I,O)} \, t \Rightarrow sb \in T) \\
)
\end{array}
$$

   (Cf. Definition 23 and Lemma 5 in [UV88]; recall the reversal of the arguments with respect to their definition of $\mathbf{C}$.)

1. A handshake process is delay insensitive if the corresponding directed process is.

□

Theorem A.8

Handshake processes are delay insensitive.

Proof Let P be a handshake process and $s, t \in \mathbf{t}P^{\leq}$.

Case $a \in \mathbf{i}P$. We derive:

$$
\begin{array}{ll}
 & a \in \mathbf{i}P \wedge s \, \mathbf{C} \, ta \\
\Rightarrow & \{ \text{ Property A.2 } \} \\
 & ta \in A^H \\
\Rightarrow & \{ \; ta \, \mathbf{c} \, t \wedge (\mathbf{c}) \cdot P^{\leq} \; \} \\
 & ta \in \mathbf{t}P^{\leq}
\end{array}
$$

Case $b \in \mathbf{o}P$. We derive:

$$
\begin{array}{ll}
 & a \in \mathbf{o}P \wedge sb \, \mathbf{C} \, t \\
\Rightarrow & \{ \text{ Property A.2 ; Definition of } \mathbf{C} \; \} \\
 & sb \in A^H \wedge sb \, \mathbf{c} \, t \\
\Rightarrow & \{ \; (\mathbf{c}) \cdot P^{\leq} \; \} \\
 & sb \in \mathbf{t}P^{\leq}
\end{array}
$$

□

In [Udd84] the notion of independence of a symbol set is introduced. It is nice to view a handshake process as a directed process in which a port forms the 'unit' of independence. Independence of a symbol set C with respect to P embodies the notion that if an input symbol is allowed to occur in $P\lceil C$ it is also allowed to occur in P.

Definition A.9 (independent alphabet)

0. Let P be a delay-insensitive directed process and C a set of symbols such that $C \subseteq (\mathbf{i}P \cup \mathbf{o}P)$. C is *independent* with respect to P if:

$$\begin{array}{ll} & (\forall s, a : s \in \mathbf{t}P \wedge a \in (C \cap \mathbf{i}P) : (sa\lceil C \in \mathbf{t}P\lceil C) \;=\; (sa \in \mathbf{t}P)) \\ \wedge & (\forall s, a : s \in \mathbf{t}P \wedge a \in (\overline{C} \cap \mathbf{i}P) : (sa\lceil \overline{C} \in \mathbf{t}P\lceil \overline{C}) \;=\; (sa \in \mathbf{t}P)) \end{array}$$

   where the complement of C with respect to $(\mathbf{i}P \cup \mathbf{o}P)$ is denoted by $\overline{C}$.

1. Let P be a handshake process and A a port structure such that $A \subseteq \mathbf{p}P$. A is independent with respect to P if $\mathbf{a}A$ is independent with respect to the directed process corresponding to P.

□

Hardly surprising, given the receptiveness of handshake processes, we arrive at the following theorem.

Theorem A.10

For handshake process P and port structure A such that $A \subseteq \mathbf{p}P$, we have: A is independent with respect to P.

□

Computation interference

The justification of the definition of parallel composition of handshake processes relies on the absence of interference. Interference may manifest itself in two forms [vdS85]: *transmission* interference and *computation* interference. Transmission interference occurs when more than one transition is on its way along the same link. The restriction to handshake traces excludes this form of interference right from the start. The absence of computation interference in handshake circuits requires some elaboration.

Definition A.11 (computation interference)

0. Directed processes P and Q are connectable if and only if the sets $\mathbf{i}P \cap \mathbf{i}Q$ and $\mathbf{o}P \cap \mathbf{o}Q$ are empty.

1. Let H be a finite set of delay-insensitive directed processes, such that elements of H are pairwise connectable. H is *free of computation interference* if [Ebe89]

$$(\forall t, P, a : t \in \mathbf{W} \cdot H \wedge P \in H \wedge a \in \mathbf{o}P : ta \lceil \mathbf{p}P \in \mathbf{t}P \Rightarrow ta \in \mathbf{W} \cdot H)$$

2. Handshake circuit H is free of computation interference if the set of corresponding directed processes is.

□

Theorem A.12

Handshake circuits are free of computation interference.

□

A similar result has been suggested in Property 4.10 of [vdS85]. Absence of computation interference in handshake circuits follows directly from the receptiveness of handshake processes. If output a may occur for some component P after trace t, trace ta will be in $\mathbf{W} \cdot H$, either because a is external, or because there is another component that is receptive for a.

Appendix B

Failure semantics

Introduction

In Chapter 5 we have developed a handshake semantics for Tangram. An alternative semantics for Tangram can be based on *failure processes* [BHR84]. Failure processes form the underlying model of CSP [Hoa85], and are the basis for a well-established theory for CSP, including a powerful calculus [RH88].

The availability of two distinct semantics for the same program notation suggests several questions, including:

0. Is the handshake-process semantics consistent with the failure semantics? If so, in what sense?

1. Can VLSI programmers use calculi that are based on failure semantics?

The last question is of obvious practical significance.

This appendix starts with a description of failure processes. By means of a simple example it is shown that an embedding of failure processes into all-active handshake processes does not exist. A more subtle approach is chosen to link handshake semantics and failure semantics are linked, resulting in positive answers to the above questions.

Failure processes

This subsection describes a process model based on failures. The description below is rather concise; for a more extensive treatment the reader is referred to [BHR84], [BR85] and [Hoa85].

An alphabet structure defines an *alphabet* as a set of communications.

Definition B.0 (alphabet of an alphabet structure)

Let A be an alphabet structure.

0. A *communication* of A is a pair $a{:}v$, such that $a \in \mathbf{c}A$ and $v \in \tau_A \cdot a$.

1. The *alphabet* of A is the set of all communications of A and is denoted by $\mathbf{a}A$.

□

Note that an alphabet is finite, on account of the finite number of ports and the finiteness of types. In CSP a communication is an event in which a process can engage. An alphabet is the set of all communications of interest. The actual occurrence of a communication is regarded as an instantaneous (atomic) event without duration.

Traces on $\mathbf{a}A$ are used to record the communication events in which a process has engaged up to some moment in time. The linear ordering of events in a trace assumes that the simultaneous occurrence of two events can be ignored. When simultaneity of two events is important, as with the synchronization of two processes, it will be represented by a single communication.

Definition B.1 (failure structure)

0. A *failure structure* is a pair $\langle A, F \rangle$, where A is an alphabet structure and F the so-called *failure set*: a relation between $(\mathbf{a}A)^*$ and $\mathcal{P} \cdot (\mathbf{a}A)$.

1. Elements of F are called *failures*. Let $\langle t, X \rangle$ with $t \in (\mathbf{a}A)^*$ and $X \in \mathcal{P} \cdot (\mathbf{a}A)$ be such a failure. Then t is referred to as its *trace* and X as its *refusal set.*

2. Let S be a failure structure. Then $\mathbf{A}S$ denotes its alphabet structure and $\mathbf{f}S$ denotes its failure set. $\mathbf{a}S$ is a shorthand for $\mathbf{a}(\mathbf{A}S)$.

□

Definition B.2 (failure process)

A failure process is a failure structure $\langle A, F \rangle$ that satisfies the following conditions:

0. $\langle \varepsilon, \emptyset \rangle \in F$

1. $\langle st, X\rangle \in F \Rightarrow \langle s, \emptyset\rangle \in F$
2. $\langle s, Y\rangle \in F \wedge X \subseteq Y \Rightarrow \langle s, X\rangle \in F$
3. $\langle s, X\rangle \in F \ \wedge \ x \in \mathbf{a}P \Rightarrow \langle s, X \cup \{x\}\rangle \in F \ \vee \ \langle sx, \emptyset\rangle \in F$

□

This is essentially the definition of [BHR84], restricted to finite alphabets. A quote from [Hoa85] explains the idea behind failures (page 129):

> "If $\langle s, X\rangle$ is a failure of [process] P, this means that P can engage in the sequence of events recorded by s, and then refuse to do anything more, in spite of the fact that its environment is prepared to engage in any of the events of X."

The four conditions have the following implications.

0. A process is a non-empty failure structure; failure $\langle \varepsilon, \emptyset\rangle$ represents its initial state.
1. If trace st can be observed, trace s must be observable as well.
2. If X can be refused then all subsets of X can be refused.
3. After any trace, a particular communication may happen, can be refused, or both.

[BR85] and [Hoa85] present "an improved failures model for communicating processes". The improvement consists of the possibility to distinguish among various forms of deadlock. The improved model is more powerful and supports a slightly more elegant algebra. For brevity's sake, this improvement is not included in this thesis.

Definition B.3 (maximal failures)

Let F be a failure set. The *maximal failures* of F, denoted by $Max{\cdot}\, F$, is defined as

$$\{t, R : \langle t, R\rangle \in F \wedge \neg(\exists R' : \langle t, R'\rangle \in F : R \subset R') : \langle t, R\rangle\}$$

□

On account of Definition B.2.2 we may conclude that the failure set of a failure process is fully characterized by its maximal failures. The set of all failure processes with alphabet structure A is denoted by $\prod_{\mathcal{F}} \cdot A$.

With each Tangram program a failure process can be associated, by means of a mapping $\mathcal{F}$: from Tangram to $\prod_{\mathcal{F}} \cdot A$. For details of such a mapping we refer to [Hoa85].

The following example provides the failure processes that correspond to a number of elementary Core Tangram programs. They are included as illustration and for later reference. For brevity's sake only the maximal failures are enumerated.

Example B.4

0. Synchronization on a. The failures of $(a^{\sim}) \cdot a$ are:

$$\{\langle \varepsilon, \emptyset \rangle, \langle a, \{a\} \rangle\}$$

1. Extension of a with port b. The failures of $(a^{\sim}, b^{\sim}) \cdot a$ are:

$$\{\langle \varepsilon, \{b\} \rangle, \langle a, \{a, b\} \rangle\}$$

2. Sequential composition of a and b. The failures of $(a^{\sim}, b^{\sim}) \cdot (a; b)$ are:

$$\{\langle \varepsilon, \{b\} \rangle, \langle a, \{a\} \rangle, \langle ab, \{a, b\} \rangle\}$$

3. Parallel composition of a and b. The failures of $(a^{\sim}, b^{\sim}) \cdot (a \parallel b)$ are:

$$\{\langle \varepsilon, \emptyset \rangle, \langle a, \{a\} \rangle, \langle b, \{b\} \rangle, \langle ab, \{a, b\} \rangle, \langle ba, \{a, b\} \rangle\}$$

4. Internal choice between a and b. The failures of $(a^{\sim}, b^{\sim}) \cdot (a \sqcap b)$ are:

$$\{\langle \varepsilon, \{a\} \rangle, \langle \varepsilon, \{b\} \rangle, \langle a, \{a, b\} \rangle, \langle b, \{a, b\} \rangle\}$$

5. External choice between a and b. This does not correspond to any Core Tangram. A possible syntax is $(a^{\sim}, b^{\sim}) \cdot [a \mid b]$, with failures:

$$\{\langle \varepsilon, \emptyset \rangle, \langle a, \{a, b\} \rangle, \langle b, \{a, b\} \rangle\}$$

6. Internal choice between $a; b$ and $b; a$. The failures of $(a^{\sim}, b^{\sim}) \cdot (a; b \sqcap b; a)$ are:

$$\{\langle \varepsilon, \{a\} \rangle, \langle \varepsilon, \{b\} \rangle, \langle a, \{a\} \rangle, \langle b, \{b\} \rangle, \langle ab, \{a, b\} \rangle, \langle ba, \{a, b\} \rangle\}$$

7. External choice between $a;b$ and $b;a$. The failures of $(a^{\sim}, b^{\sim})\cdot[a;b \mid b;a]$ are:

$$\{\langle \varepsilon, \emptyset\rangle, \langle a, \{a\}\rangle, \langle b, \{b\}\rangle, \langle ab, \{a,b\}\rangle, \langle ba, \{a,b\}\rangle\}$$

□

The remainder of this section is used to discuss the structure of $\coprod_{\mathcal{F}}\cdot A$. For more background, appreciation and proofs, the reader is referred to earlier cited material.

Failure processes with the same alphabet structure can be ordered.

Definition B.5 (refinement order)

Let P and Q be failure processes with alphabet structure A.

0. P refines to Q, denoted by $P \sqsubseteq Q$, if $\mathbf{f}P \supseteq \mathbf{f}Q$. Process Q has less failures than P and is therefore better.

1. Process $CHAOS\cdot A$ is defined as $\langle A, (\mathbf{a}A)^* \times \mathcal{P}\cdot(\mathbf{a}A)\rangle$.

2. An (ascending) chain is an infinite sequence $(i : 0 \leq i : P_i)$ of processes such that $P_i \sqsubseteq P_{i+1}$.

□

Clearly, $\langle \coprod_{\mathcal{F}}\cdot A, \sqsubseteq\rangle$ is a partial order. According to [BR85], $(\coprod_{\mathcal{F}}\cdot A, \sqsubseteq)$ is also a CPO, with $CHAOS\cdot A$ as least element and $(\sqcup\, i : 0 \leq i : P_i)$ as limit of chain $(i : 0 \leq i : P_i)$.

Example B.6

In the following refinements $S \sqsubseteq T$ is a shorthand for $\mathcal{F}\cdot S \sqsubseteq \mathcal{F}\cdot T$, where S and T are Core Tangram programs.

0. $(a^{\sim}, b^{\sim})\cdot(a \sqcap b) \;\sqsubseteq\; (a^{\sim}, b^{\sim})\cdot a$

1. $(a^{\sim}, b^{\sim})\cdot(a; b \sqcap b; a) \;\sqsubseteq\; (a^{\sim}, b^{\sim})\cdot(a; b)$

2. $(a^{\sim}, b^{\sim})\cdot(a; b \sqcap b; a) \;\sqsubseteq\; (a^{\sim}, b^{\sim})\cdot(a \parallel b)$

3. $(a^{\sim}, b^{\sim})\cdot[a; b \mid b; a] \;=\; (a^{\sim}, b^{\sim})\cdot(a \parallel b)$

□

Embedding of failure processes into handshake processes

We are looking for a mapping from failure processes to handshake processes that preserves the 'essential' properties of the original processes. By this we mean that the mapping must respect ordering and that the image of all failure processes must be equally rich in structure. Such a mapping is called an *embedding*:

Definition B.7 (embedding)

Let $\mathcal{E}$ be a function from CPO X to CPO Y. Function $\mathcal{E}$ is an embedding [DP90] of X into Y if:

0. $\mathcal{E} \cdot P \sqcap \mathcal{E} \cdot P = \mathcal{E} \cdot (P \sqcap Q)$

1. $\mathcal{E} \cdot P \sqcup \mathcal{E} \cdot P = \mathcal{E} \cdot (P \sqcup Q)$

2. $P = Q \equiv \mathcal{E} \cdot P = \mathcal{E} \cdot Q$

□

The following property of an embedding follows immediately.

Property B.8

An embedding is order preserving, i.e. $P \sqsubseteq Q \Rightarrow \mathcal{E} \cdot P \sqsubseteq \mathcal{E} \cdot Q$.

□

Our search for such an embedding starts with comparing a few refinements in the two process models. In the domain of failure processes we have (cf. Example B.6.2):

$$a; b \sqcap b; a \sqsubseteq a \,||\, b$$

A similar refinement in the domain of handshake processes, however, does not hold:

$$a^\bullet; b^\bullet \sqcap b^\bullet; a^\bullet \not\sqsubseteq a^\bullet \,||\, b^\bullet$$

The left-hand side requires the handshakes through $a^\bullet$ and $b^\bullet$ to exclude each other in time. The parallel composition at the right hand side, however, has e.g. $a_0 b_0 a_1 b_1$ as quiescent trace.

The above example shows that a mapping based on $\mathcal{H}$ (cf. Section 5.3) is not order preserving, and hence not an embedding. It also suggests that there does not exist an embedding from failure processes to all-active handshake processes.

However, in the space of *all-passive* processes we do have (cf. B.6.2)

$$a^\circ; b^\circ \sqcap b^\circ; a^\circ \sqsubseteq a^\circ \parallel b^\circ$$

Moreover, as with failure processes we have (cf. B.6.3):

$$[a^\circ; b^\circ \mid b^\circ; a^\circ] = a^\circ \parallel b^\circ$$

Both examples show that order of passive handshakes is masked by reordering. Because of this masking effect there is less distinction in the space of all-passive processes than in the space of all-active processes. This insight will be elaborated along two different lines that will meet at the end of this appendix:

- *handshake expansion*: an embedding of failure processes into the set of all-passive handshake processes, and
- *passivation*: a transformation of an all-active process into an all-passive process.

Handshake expansion

Handshake expansion is a mapping from failure structures to handshake structures. Handshake expansion is also defined for alphabet structures, traces, refusal sets and failures.

Definition B.9 (handshake expansion)

0. The *handshake expansion* of alphabet structure A, denoted by $\mathcal{E} \cdot A$ is the port structure defined by

$$\begin{array}{lcl} (\mathcal{E} \cdot A)^\circ & = & \{a : a \in \mathbf{p}?A : a^\circ ?\tau_A \cdot a\} \cup \{a : a \in \mathbf{p}!A : a^\circ !\tau_A \cdot a\} \\ (\mathcal{E} \cdot A)^\bullet & = & \emptyset \end{array}$$

   Note that all ports are chosen to be passive.

1. The *handshake expansion* of trace t with respect to alphabet structure A, denoted by $\mathcal{E} \cdot (t, A)$, is defined by

$$\begin{array}{lcl} \mathcal{E} \cdot (\varepsilon, A) & = \varepsilon & \\ \mathcal{E} \cdot (a{:}v\ t, A) & = & \textbf{if} \quad a \in \mathbf{p}!A \quad \rightarrow \quad a_0\ a_1{:}v\ \ \mathcal{E} \cdot (t, A) \\ & & \ [] \quad a \in \mathbf{p}?A \quad \rightarrow \quad a_0{:}v\ \ a_1\ \ \mathcal{E} \cdot (t, A) \\ & & \textbf{fi} \end{array}$$

2. The *handshake expansion* of refusal set X with respect to alphabet structure A, denoted by $\mathcal{E}\cdot(X, A)$, is defined as the symbol set

$$\{a, v : a \in \mathbf{p}?A \wedge a{:}v \in X : a_0{:}v\} \cup \{a : a \in \mathbf{p}!A \wedge a \in X : a_0\}$$

These symbols are received, but refused in the sense that they are not acknowledged.

3. The *handshake expansion* of a failure $\langle t, X\rangle$, with respect to alphabet structure A, denoted by $\mathcal{E}\cdot(\langle t, X\rangle, A)$, is defined as the handshake-trace set

$$\{u : \#_u = \mathcal{E}\cdot(X, A) : \mathcal{E}\cdot(t, A)u\}^{\mathbf{r}}$$

where $\#_u$ denotes the bag of symbols of trace u. Actually, u is a permutation of $\mathcal{E}\cdot(X, A)$.

4. The *handshake expansion* of failure structure $\langle A, F\rangle$, denoted by $\mathcal{E}\cdot\langle A, F\rangle$ is defined as the handshake structure

$$\langle \mathcal{E}\cdot A, \{f : f \in F : \mathcal{E}\cdot(f, A)\}\rangle$$

□

The crux of the above definition is in the handshake expansion of a failure $\langle t, X\rangle$ (cf. Definitoin B.9.3). The postfix u corresponds with refusal set X. If the environment continues with handshakes through all ports in X (by sending the corresponding ˜-symbols) the state resulting after tu is quiescent.

The following property is helpful in understanding function $\mathcal{E}$.

Property B.10

Let f and g be failures defined on alphabet structure A. Then

$$f \neq g \;\Rightarrow\; \mathcal{E}\cdot(f, A) \cap \mathcal{E}\cdot(g, A) = \emptyset$$

□

Theorem B.11

Let A be an alphabet structure and let $P \in \prod_{\mathcal{F}}\cdot A$. Then:

0. $\mathcal{E}$ is an embedding;

1. $\mathcal{E} \cdot P$ is a handshake process;

2. $\mathcal{E} \cdot CHAOS \cdot A = CHAOS \cdot \mathcal{E} \cdot A$;

3. $\mathcal{E}$ is continuous;

4. Hence, $\mathcal{E} \cdot (\prod_{\mathcal{F}} \cdot A)$ is a CPO.

□

The definition of $\mathcal{E}$ ignores the issue of successful termination. Extending $\mathcal{E}$ to such a more comprehensive process model is relatively straightforward. Given such an extended embedding, equalities such as

$$\begin{array}{rcl} \mathcal{E} \cdot (P;Q) & = & \mathcal{E} \cdot P; \mathcal{E} \cdot Q \\ \mathcal{E} \cdot (P \,||\, Q) & = & \mathcal{E} \cdot P \,||\, \mathcal{E} \cdot Q \\ \mathcal{E} \cdot \#[P] & = & \#[\mathcal{E} \cdot P] \end{array}$$

can easily be verified.

Passivation

Another way to obtain an all-passive process is to connect passivators to the active ports of a handshake process. The following definition is restricted to all-active processes with undirected ports only. Extension to general handshake processes is straightforward.

Definition B.12 (passivation)

The passivation of an all-active handshake process P, denoted by $\pi \cdot P$, is defined as

$$\underline{l} \cdot P \;||\; (||\; a : a \in \mathbf{p}^{\bullet} P : PAS \cdot (la^{\circ}, a^{\circ}))$$

where $\underline{l} \cdot P$ denotes the l-renaming of P defined in Definition 6.0.

□

The effect of passivation is illustrated by the following example.

Example B.13

$$(la^{\bullet}; lb^{\bullet} \sqcap lb^{\bullet}; la^{\bullet}) \,||\, \#[la^{\circ} : a^{\circ}] \,||\, \#[lb^{\circ} : b^{\circ}] \;=\; (a^{\circ}; b^{\circ} \sqcap b^{\circ}; a^{\circ})$$

□

Let T be a Tangram program. The following theorem expresses that the passivation of $\mathcal{H} \cdot T$ equals the handshake expansion of $\mathcal{F} \cdot T$.

Theorem B.14

$$\pi \circ \mathcal{H} \;=\; \mathcal{E} \circ \mathcal{F}$$

□

An important corollary is obtained when this theorem is combined with the compilation theorem. Graphically this corollary is illustrated in Figure B.0.

Corollary B.15

$$\pi \circ || \circ \mathcal{C} \;=\; \triangleright^* \circ \mathcal{E} \circ \mathcal{F}$$

□

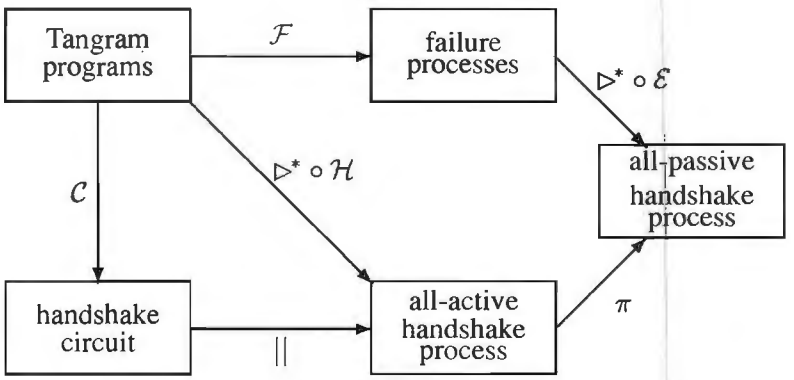


Figure B.0: Failure processes related to handshake circuits.

Corollary B.15 may be applied as follows. We call two all-active processes P and Q *π-equivalent* if their passivations are identical. Let the behaviors of two compiled handshake circuits G and H be π-equivalent. Then there does not exist any third compiled handshake circuit that can distinguish G from H when it is connected to them by passivators. Under such circumstances, the designer may use the all-passive semantics of Tangram, as obtained by $\pi \circ \mathcal{H}$. The existence of the embedding $\mathcal{E}$ then demonstrates that the VLSI programmer can then apply programming laws that are based on a failure semantics of Tangram (cf. [RH88]).

Bibliography

[Ano73] Anonymous. Science and the citizen. *Scientific American*, 228:43–44, 1973.

[BE90] J.A. Brzozowski and J.C. Ebergen. *On the Delay-sensitivity of Gate Networks*. Technical Report 90/5, Eindhoven University of Technology, 1990.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, 1984.

[BM88] Steven M. Burns and Alain J. Martin. Synthesis of Self-Timed Circuits by Program Transformation. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 99–116. Elsevier Science Publishers B.V., 1988.

[BM91] Peter Beerel and Teresa Meng. Semi-Modularity and Self-Diagnostic Asynchronous Control Circuits. In Carlo H. Sequin, editor, *Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 103–117. The MIT Press, 1991.

[BR85] S.D. Brookes and A.W. Roscoe. An Improved Failures Model for Communicating Sequential Processes. In *Proceedings NSF-SERC Seminar of Concurrency*, pages 281–305. Springer-Verlag, 1985.

[Bro89] R.W. Brockett. Smooth Dynamical Systems which Realize Arithmetical and Logical Operations. In Hendrik Nijmeijer and Johannes M. Schumacher, editors, *Three Decades of Mathematical Systems Theory: A Collection of Surveys at the Occasion of the 50th Birthday of J.C. Willems*, pages 19–30. Springer-Verlag, 1989.

[Bro90] Geoffrey M. Brown. Towards Truly Delay-Insensitive Circuit Realizations of Process Algebras. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*, pages 120–152. *Workshops in Computing*. Springer-Verlag, 1990.

[BS89] Erik Brunvand and Robert Sproull. Translating Concurrent Programs into Delay-Insensitive Circuits. In *IEEE International Conference on Computer Aided Design; Digest of Technical Papers*, pages 262–265. IEEE Computer Society Press, 1989.

[CM73] T.J. Chaney and C.E. Molnar. Anomalous Behavoir of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, C-22(4):421–422, 1973.

[DGY90] Ilana David, Ran Ginosar, and Michael Yoeli. *Self-Timed is Self-Diagnostic*. Technical Report, University of Utah, 1990.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. An ACM Distinguished Dissertation*, MIT Press, 1989.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order. Cambridge Mathematical Textbooks*, Cambridge University Press, 1990.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[Ebe89] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. CWI Tract 56 (Centre for Mathematics and Computing Science, Amsterdam), 1989.

[Elf76] Joost Elfers. *Tangram; the Ancient Chinese Shapes Game*. Penguin Books, 1976.

[Fuj85] Hideo Fujiwara. *Logic Testing and Design for Testability*. The MIT Press, 1985.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes. Series in Computer Science*, Prentice-Hall International, 1985.

[INM89] INMOS Limited, editor. *Occam 2 Programming Manual. Series in Computer Science*, Prentice-Hall International, 1989.

[JHJ89] Mark B. Josephs, C.A.R. Hoare, and He Jifeng. A Theory of Asynchronous Processes. manuscript, 1989.

[Jon85] B. Jonsson. A model and proof system for asynchronous networks. In *Proc. 4th ACM Symposium on Principles of Distributed Computing*, pages 49–58. ACM, 1985.

[Jos90] M.B. Josephs. *Receptive Process Theory*. Computing Science Note 90/8, Eindhoven University of Technology, 1990.

[JU91] Mark B. Josephs and Jan Tijmen Udding. An Algebra for Delay-Insensitive Circuits. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 147–175. Volume 3. AMS-ACM, 1991.

[Kal86] Anne Kaldewaij. *A Formalism for Concurrent Processes*. PhD thesis, Eindhoven University of Technology, 1986.

[Kes91a] J.L.W. Kessels. Designing Counters with Bounded Response Time. In W.H.J. Feijen and A.J.M van Gasteren, editors, *C.S. Scholten dedicata: van oude machines en nieuwe rekenwijzen*, pages 127–140. Academic Service, Schoonhoven The Netherlands, 1991.

[Kes91b] J.L.W. Kessels. The Systematic Design of a Systolic RSA Converter. In *Proc. Workshop on Correct Hardware Design Methodologies*, pages 243–260. 1991.

[KR89] Anne Kaldewaij and Martin Rem. A derivation of a systolic rank order filter with constant response time. In J.L.A.

van de Snepscheut, editor, *Mathematics of Program Construction*, pages 281–296. Volume 375 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[KS90] J.L.W. Kessels and F.D. Schalij. VLSI Programming for the Compact Disc Player. *Science of Computer Programming*, 15:235–248, 1990.

[KTT*88] Shinji Komori, Hidehiro Takata, Toshiyuki Tamura, Fumiyasu Asai, Takio Ohno, Osamu Tomisawa, Tetsuo Yamasaki, Kenji Shima, Katsuhiko Asada, and Hiroaki Terada. An Elastic Pipeline Mechamism by Self-Timed Circuits. *IEEE Journal of Solid-State Circuits*, 23(1):111–117, 1988.

[KTT*89] Shinji Komori, Hidehiro Takata, Toshiyuki Tamura, Fumiyasu Asai, Takio Ohno, Osamu Tomisawa, Tetsuo Yamasaki, Kenji Shima, Hiroaki Nishikawa, and Hiroaki Terada. A 40-MFLOPS 32-bit Floating-Point Processor with Elastic Pipeline Scheme. *IEEE Journal of Solid-State Circuits*, 24(5):1341–1347, 1989.

[KU91] Anne Kaldewaij and Jan Tijmen Udding. Rank Order Filters and Priority Queues. manuscript, 1991.

[KvBB*92] Joep Kessels, Kees van Berkel, Ronan Burgess, Marly Roncken, Ronald Saeijs, and Frits Schalij. A Tangram Program for Error Decoding in the Compact Disc Player. In *Proceedings of the European Design Automation Conference*, pages –. 1992.

[KZ90] Anne Kaldewaij and Gerard Zwaan. A systolic design for acceptors of regular languages. *Science of Computer Programming*, 15:171–184, 1990.

[LSU89] Roger Lipsett, Carl Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic publishers, 1989.

[LvMvdW*91] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSeeney, J.O. Huisken, and O.P. McArdle. PHIDEO: A Silicon Compiler for High Speed Algorithms. In *Proceedings of the European Design Automation Conference*, pages 436–441. 1991.

[Mar85a] Alain J. Martin. *Compiling Communicating Processes into Delay-Insensitive VLSI Circuits.* Technical Report, California Institute of Technology, Department of Computer Science, Pasadena CA 91125, USA, 1985.

[Mar85b] Alain J. Martin. The Design of a Self-Timed Circuit for Distributed Mutual Exclusion. In Henry Fuchs, editor, *Chapel Hill Conference on VLSI*, pages 245–260. 1985.

[Mar89] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C.A.R. Hoare, editor, *UT Year of Programming; Institute on Concurrent Programming.* Addison-Wesley, 1989.

[Mar90] Alain J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[MBL*89] Alain J. Martin, Steven M. Burns, T.K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The First Asynchronous Microprocessor: The Test Results. In *Computer Architecture News*, pages 95–110. Volume 17. MIT Press, 1989.

[MBM90] Teresa H.-Y. Meng, Robert W. Broderson, and David G. Messerschmitt. A Clock-Free Chip Set for High-Sampling Rate Adaptive Filters. *Journal of VLSI Signal Processing*, 1(4):345–365, 1990.

[McK65] W.M. McKeeman. Peephole Optimization. *Communications of the ACM*, 8:443–444, 1965.

[MH91] Alain J. Martin and Pieter J. Hazewindus. Testing Delay-Insensitive Circuits. In Carlo H. Sequin, editor, *Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 118–132. The MIT Press, 1991.

[Mil65] Raymond E. Miller. *Switching Theory Volume II: Sequential Circuits and Machines.* John Wiley & Sons Inc., 1965.

[Mis84] J. Misra. Reasoning about networks of communicating processes. 1984. Presented at INRIA Advanced Nato Study Institute on

Logics and Models for Verification and Specification of Concurrent Systems, La Colle-sur-Loupe, France.

[Mis87] F.C. Mish et al. *Webster's Ninth New Collegiate Dictionary*. Merriam-Webster Inc., 1987.

[NvBRS88] Cees Niessen, C.H. (Kees) van Berkel, Martin Rem, and Ronald W.J.J. Saeijs. VLSI Programming and Silicon Compilation; A Novel Approach from Philips Research. In *Proceedings of the 1988 IEEE Int. Conf. on Computer Design: VLSI in Computers & Processors*, pages 150–151. 1988.

[OSC67] S.M. Ornstein, M.J. Stucki, and W.A. Clark, A Functional Description of Macromodules. In *Sprint Joint Computer Conference*, pages 337–355. AFIPS, 1967.

[Phi90] Philips Components. *Integrated Circuits Data Handbook; Radio, audio and associated systems; Bipolar, Mos; CA3089 to TDA1510A*. Philips Components, 1990.

[Rem81] Martin Rem. The VLSI Challenge: Complexity Bridling. In John P. Gray, editor, *VLSI 81*, pages 65–74. Academic Press, 1981.

[Rem87] Martin Rem. Trace Theory and Systolic Computations. In J.W. de Bakker et al., editor, *PARLE Parallel Architectures and Languages in Europe*, pages 14–33. Volume 258 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987.

[Rem91] Martin Rem. The Nature of Delay-Insensitive Computing. In Graham Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 105–122. Springer-Verlag, 1991.

[RH88] A.W. Roscoe and C.A.R. Hoare. The Laws of OCCAM Programming. *Theoretical Computer Science*, 60:177–229, 1988.

[Sei80] Charles L. Seitz. System Timing. In C.A. Mead and L.A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[Sei84] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, 1984.

[SM77] Ivan. E. Sutherland and Carver A. Mead. Microelectronics and Computer Science. *Scientific American*, 237(9):210–228, 1977.

[Sut89] Ivan Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[SvB88] Ronald W.J.J. Saeijs and C.H. (Kees) van Berkel. The Design of the VLSI Image-Generator ZaP. In *Proceedings of the 1988 IEEE Int. Conf. on Computer Design: VLSI in Computers & Processors*, pages 163–166. 1988.

[SvBB*91] Frits Schalij, Kees van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken, and Ronald Saeijs. What makes Tangram a general-purpose VLSI-programming language? manuscript, 1991.

[Udd84] Jan Tijmen Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Eindhoven University of Technology, 1984.

[Udd86] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1(4):197–204, 1986.

[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, 1988.

[vB91] C.H. van Berkel. *Beware the isochronic fork*. Technical Report UR 003/91, Philips Research, 1991.

[vB92] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):, 1992.

[vBKR*91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald W.J.J. Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of the European Design Automation Conference*, pages 384–389. 1991.

[vBRS88] C.H. (Kees) van Berkel, Martin Rem, and Ronald W.J.J. Saeijs. VLSI Programming. In *Proceedings of the 1988 IEEE Int.*

Conf. on Computer Design: VLSI in Computers & Processors, pages 152–156. 1988.

[vBS88] C.H. (Kees) van Berkel and Ronald W.J.J. Saeijs. Compilation of Communicating Processes into Delay-Insensitive Circuits. In *Proceedings of the 1988 IEEE Int. Conf. on Computer Design: VLSI in Computers & Processors*, pages 157–162. 1988.

[vdS85] Jan L.A. van de Snepscheut. *Trace Theory and VLSI Design*. Volume 200 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.

[Ver88] Tom Verhoeff. Delay-insensitive codes – an overview. *Distributed Computing*, 3:1–8, 1988.

[Ver89] Tom Verhoeff. Personal communications. 1989. Eindhoven University of Technology.

[WD89] R. Woudsma and A. Delaruelle. The Design of DSP Components for the CD Digital Audio System using Silicon Compilation Techniques. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 20.4.1–20.4.5. 1989.

Glossary of Symbols

Sets

Notation	*Meaning*	*Definition*
$B^{\leq}$	$\leq$-closure of set B	2.0
$(\leq)\cdot B$	B is $\leq$-closed	2.0
$B \rightsquigarrow C$	B initializes set C	7.23

Traces

Notation	*Meaning*	*Definition*
ε	empty trace	2.7
$len\cdot t$	length of trace t	2.7
$s \leq t$	s is a prefix of t	2.7
st	concatenation of traces s and t	2.7
B^*	set of all traces over alphabet B	2.7
$t\lceil B$	projection of t on B	2.7
$\#_t$	bag of symbols in t	A.3

Ports, port structures and port definitions

Notation	*Meaning*	*Definition*
$\mathbf{a}p$	symbol set of port p	2.3
$\mathbf{0}p$	**0** symbols of p	2.3
$\mathbf{1}p$	**1** symbols of p	2.3
A°	passive ports of port structure A	2.3
$A^\bullet$	active ports of A	2.3
$\mathbf{i}A$	input symbols of A	2.4
$\mathbf{o}A$	output symbols of A	2.4

a°	passive nonput port (structure) a	2.5
$a^\circ?T$	passive input port (structure) a of type T	2.5
$a^\circ!T$	passive output port (structure) a of type T	2.5
$a^\bullet$	active nonput port (structure) a	2.5
$a^\bullet?T$	active input port (structure) a of type T	2.5
$a^\bullet!T$	active output port (structure) a of type T	2.5
$\overline{A}$	reflection of A	3.0
$\mathbf{e}A$	external port structure of A	3.8
$A \bowtie B$	A and B are connectable	3.0
$A \mathbin{\vartriangle\!\!\vartriangle} B$	A and B are conformant	4.12
$A \cup B$	pairwise union of A and B	2.3
$A \setminus B$	pairwise set difference of A and B	2.3
$\triangleright^\circ$	passive port 'go'	5.6

Handshake traces and handshake-trace sets

Notation	*Meaning*	*Definition*
$closed \cdot t$	all handshakes in handshake trace t are closed	2.26
A^H	set of handshake traces over port structure A	2.8
$s\ \mathbf{r}_A\ t$	s reorders t w.r.t. A	2.14
$s\ \mathbf{r}\ t$	$s\ \mathbf{r}_A\ t$ with A obvious from context	2.14
$B^{\mathbf{r}}$	**r**-closure of handshake trace set B	2.0
$(\mathbf{r}) \cdot B$	B is **r**-closed	2.0
$s\ \mathbf{x}_A\ t$	s is an input extension of t w.r.t. A	2.17
$s\ \mathbf{x}\ t$	$s\ \mathbf{x}_A\ t$ with A obvious from context	2.17
$B^{\mathbf{x}}$	**x**-closure of B	2.0
$(\mathbf{x}) \cdot B$	B is **x**-closed	2.0
$s\ \mathbf{c}_A\ t$	s is composable with t w.r.t. A	A.1
$s\ \mathbf{c}\ t$	$s\ \mathbf{c}_A\ t$ with A obvious from context	A.1
$B^{\mathbf{c}}$	**c**-closure of B	2.0
$(\mathbf{c}) \cdot B$	B is **c**-closed	2.0

Handshake structures

Notation	*Meaning*	*Definition*
$\mathbf{p}S$	port structure of handshake structure S	2.10
$\mathbf{t}S$	handshake-trace set of S	2.10

$suc \cdot (t, S)$	successor set of trace t in S	2.11
$pas \cdot (t, S)$	t is passive in S	2.11
$Pas \cdot S$	passive restriction of S	2.11
$after \cdot (t, S)$	handshake structure after t	2.25
$div \cdot S$	divergences of S	3.10
$S \sqsubseteq T$	S refines to T	2.29
$S \sqcap T$	union of S and T	2.34
$S \sqcup T$	intersection of S and T	2.36
$(\sqcup\, i : 0 \leq i : S)$	limit of chain $(i : 0 \leq i : S)$	2.37
$S \mathbf{w} T$	weave of S and T	3.3
$S \mathbf{b} T$	blend of S and T	3.8
$S \mid\mid T$	parallel composition of S and T	3.14

Handshake processes

Notation	*Meaning*	*Definition*
$P \sqsubseteq_R Q$	P refines to Q in the context of R	7.1
$P \underline{\sqsubseteq}_R Q$	P strongly refines to Q in the context of R	7.5
$\phi \cdot P$	phase reduction of P	7.10
$\phi_2 \cdot P$	2-phase reduction of P	7.11
$\phi_{4c} \cdot P$	complete 4-phase reduction of P	7.14
$\phi_{4q} \cdot P$	quick 4-phase reduction of P	7.17
$\pi \cdot P$	passivation of P	B.12

Handshake circuits

Notation	*Meaning*	*Definition*
$\underline{\bowtie} H$	handshake circuit H is connectable	3.20
$\mathbf{e} H$	external port structure of H	3.21
$\mathbf{W} \cdot H$	weave of H	3.21
$\mathbf{B} \cdot H$	blend of H	3.21
$\|\| \cdot H$	parallel composition of the components of H	3.21

Sequential handshake processes

Notation	*Meaning*	*Definition*
skip, **stop**		4.9
a°	passive handshake through port a	4.9
$a^\bullet$	active handshake through a	4.9
$P =_{\cdot} Q$	sequential handshake process P refines to Q	4.4
$P \sqcup Q$	intersection of P and Q	4.6
$P \mid^{\cdot} Q$	P in parallel with connectable Q	4.10
$P \overset{\bullet\bullet}{\|\|} Q$	all-active P in parallel with all-active Q	5.5
$(A)\cdot P$	P extended with port structure A	4.15
$\|[A \cdot P]\|$	P with A concealed	4.18
$P \mid\mid Q$	nondeterministic choice between P and Q	4.5, 4.21
$P; Q$	P followed by Q	4.23
$\#N[P]$	N-fold repetition of P	4.26
$\#[P]$	infinite repetition of P	4.28
$a^\circ : P$	P enclosed by a°	4.30
$[P \mid Q]$	choice between guarded processes P and Q	4.33
$\triangleright^* \cdot P$	repeatable go of P	5.7
σ	σ function	6.22

Alphabet structures

Notation	*Meaning*	*Definition*
$\mathbf{p}A$	ports of alphabet structure A	5.0
$\mathbf{p}?A$	input ports of A	5.0
$\mathbf{p}!A$	output ports of A	5.0
$\mathbf{v}A$	variables of A	5.0
$\mathbf{v}?A$	read ports of A	5.0
$\mathbf{v}!A$	write ports of A	5.0
τ_A	type function of A	5.0
$A \mathbin{\triangle\!\!\triangle} B$	A and B are conformant	5.2
$A \cup_{\wedge\!\wedge} B$	conformant union of A and B	5.2
$A \setminus_{\triangle\!\!\triangle} B$	conformant difference of A and B	5.2
$A \bowtie B$	A and B are connectable	5.2
$A \cup_{\bowtie} B$	connectable union of A and B	5.2
$\mathcal{H}\cdot A$	port structure of A	5.3
$\underline{l}\cdot A$	l-renaming of A	6.0

$\underline{r}\cdot A$	r-renaming of A	6.0
$\mathcal{E}\cdot A$	handshake expansion of A	B.9

Core Tangram

Notation	*Meaning*	*Definition*
$a\tilde{}$	synchronization port	5.1
skip		5.4
stop		5.4
a	synchronization on a	5.4
$(A)\cdot S$	S extended with alphabet structure A	5.4
$S \sqcap T$	nondeterministic choice between S and T	5.4
$S;T$	S followed by T	5.4
$\#N[S]$	N-fold repetition of S	5.4
$\#[S]$	infinite repetition of S	5.4
$S \mid\mid T$	S in parallel with T	5.4
$\|[A \mid S]\|$	S with A concealed	5.4
$\underline{l}\cdot S$	l-renaming of S	6.0
$\underline{r}\cdot S$	r-renaming of S	6.0
$\mathcal{C}\cdot S$	handshake circuit of S	6.20
$\mathcal{F}\cdot S$	failure process of S	in B

Failure structures

Notation	*Meaning*	*Definition*
$\mathbf{A}S$	alphabet structure of failure structure S	B.1
$\mathbf{f}S$	failure set of S	B.1
$S \sqsubseteq T$	S refines to T	B.5
$\mathcal{E}\cdot S$	handshake expansion of S	B.9

Index

Samenvatting

Het proefschrift handelt over het ontwerpen van digitale VLSI schakelingen. De volgende ontwerpaanpak wordt hierbij verondersteld:

- een ontwerper beschrijft zijn systeem in een geschikte programmeertaal;
- een zogenaamde siliciumcompiler vertaalt dit programma in een VLSI schakeling.

De keuze van de programmeertaal is bepalend voor het toepassingsgebied, het gemak van ontwerpen en de efficiëntie van het resultaat. In het proefschrift wordt de taal Tangram geïntroduceerd als een algemeen toepasbare VLSI-programmeertaal. Tangram is gebaseerd op Communicating Sequential Processes (CSP). De geschiktheid van de taal wordt beargumenteerd en geïllustreerd aan de hand van een aantal voorbeelden in Hoofdstuk 1. Hoofdstuk 5 beschrijft een preciese definitie van Tangram.

Bij het vertalen van Tangram programma's naar VLSI circuits spelen zogenaamde handshake circuits een centrale rol. Een handshake circuit is een netwerk van elementaire asynchrone bouwstenen die onderling communiceren volgens een handshake protocol. Een theorie over handshake circuits vormt het hart van het proefschrift. Deze theorie omvat:

- een procesmodel ("handshake-processen") waarin de gedragingen van handshake circuits, de bijbehorende bouwstenen, en Tangram programma's kunnen worden vastgelegd (Hoofdstuk 2),
- een bijbehorende algebra (Hoofdstuk 4),
- een analyse van eigenschappen van handshake circuits, zoals vertragingsongevoeligheid (Hoofdstuk 3 en Appendix A), en
- een inbedding van CSP-processen gebaseerd op een zogenaamde failuresemantiek in handshake-processen (Appendix B).

De vertaling van Tangram programma's naar handshake circuits is gedefinieerd als een recursieve functie, gestructureerd volgens de Tangram-grammatica. Voor de kern van Tangram wordt een precies gedefinieerde equivalentie tussen Tangram programma's en door vertaling verkregen handshake circuits bewezen in Hoofdstuk 6.

Van de afbeelding van handshake circuits naar asynchrone VLSI circuits worden een aantal aspecten schetsmatig behandeld in Hoofdstuk 7: (peephole) optimalisaties, handshake protocollen, waardecoderingen, circuit-initialisatie en testbaarheid.

In een afsluitend hoofdstuk wordt ingegaan op enige praktische ervaringen met het ontwerpen van VLSI systemen in Tangram en het automatisch vertalen van deze programma's naar VLSI layouts middels een bij Philips Research ontwikkelde siliciumcompiler. Deze compiler is gebaseerd op de in het proefschrift behandelde methode. Hoofdstuk 8 behandelt tevens een aantal voor- en nadelen van asynchrone schakelingen, mede aan de hand van gepubliceerde resultaten van derden.

Curriculum vitae

Op 15 juni 1956 werd ik geboren in Leimuiden. Na in 1974 het diploma Atheneum B behaald te hebben aan het Bonaventura College te Leiden, begon ik de studie Elektrotechniek aan de Technische Hogeschool Delft, thans TU Delft. Het afstuderen vond plaats in de vakgroep Netwerktheorie onder leiding van prof. P. Dewilde met als onderwerp "Automatic Integrated Circuit Layout Verification; Boolean Operations on IC masks". In 1980 studeerde ik met lof af.

Sindsdien ben ik werkzaam bij het Philips Natuurkundig Laboratorium te Eindhoven. Na een aantal jaren onderzoek te hebben verricht aan computerondersteuning bij VLSI-layoutontwerp, verschoof mijn interesse en werk in de richting van parallelle berekeningen, asynchrone schakelingen en siliciumcompilatie. Onderzoek op deze gebieden heeft onder leiding van prof. M. Rem geleid tot deze dissertatie.

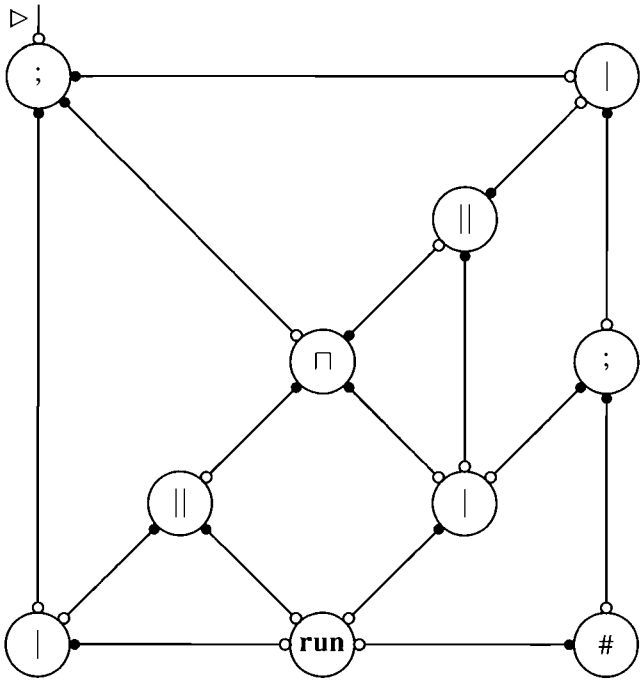
Stellingen

behorende bij het proefschrift

Handshake circuits: an intermediary between communicating processes and VLSI

van

Kees van Berkel



Eindhoven
6 mei 1992

0. Omdat asynchrone schakelingen alleen aktief zijn op plaatsen en tijden dat het nodig is, en bovendien geen energie verspillen aan races en klokdistributie, zijn ze potentieel zuiniger met energie.

   [Lit.] Hoofdstuk 8 van dit proefschrift.

1. Een nacking arbiter (ook wel non-blocking arbiter genoemd) wordt gespecificeerd door het handshake proces

$$
\begin{array}{l}
(a^\circ!\textbf{bool}\ \&\ b^\circ!\textbf{bool}). \\
\ |[\ \ xa, xb : \textbf{var bool} \\
\ |\ \ xa, xb := \textbf{true}, \textbf{true} \\
\quad ;\#[\ \ [\ a^\circ!xb;\ \ xa := xa \neq xb \\
\qquad\quad |\ b^\circ!xa;\ \ xb := xb \neq xa \\
\qquad\quad] \\
\quad\ \ ] \\
]|
\end{array}
$$

   [Lit.] Mark. B. Josephs and Jan Tijmen Udding. Delay-insensitive circuits: an algebraic approach to their design. In *CONCUR'90; Theories of Concurrency: Unification and Extension*, pages 343–366. Volume 458 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

2. Ten behoeve van de correcte implementatie van isochrone vorken is het belangrijk de spreiding in logische drempelspanningen van VLSI operatoren beperkt te houden. Sequentiële VLSI operatoren gerealiseerd met behulp van een zogenaamde trickle inverter [0] vertonen een inherent grote spreiding. VLSI operatoren met uniforme logische drempelspanningen kunnen worden gerealiseerd volgens [1].

   [0] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C.A.R. Hoare, editor, *UT Year of Programming; Institute on Concurrent Programming*, Addison-Wesley, 1989.

   [1] Kees van Berkel. *Beware the Isochronic Fork.* Verschijnt in *Integration, the VLSI journal*, 13(2), 1992.

3. Een sequentiële VLSI-operator gespecificeerd door de produktieregels:

$$
\begin{array}{rcl}
F & \rightarrow & z\uparrow \\
G & \rightarrow & z\downarrow
\end{array}
$$

   kan worden ontleed in een combinatorische operator gespecificeerd door:

$$
\begin{array}{rcl}
F \vee (z'' \wedge \neg G) & \rightarrow & z'\uparrow \\
G \vee (\neg z'' \wedge \neg F) & \rightarrow & z'\downarrow
\end{array}
$$

en een isochrone vork met ingang z' en uitgangen z en z''.

[**Lit.**] C.H. van Berkel. *Beware the isochronic fork.* Technical Report UR 003/91, Philips Research, 1991.

4. Het gebruik van dynamische circuits bij de realisatie van vertragingsongevoelige schakelingen leidt tot een interessante reductie in circuitafmetingen.

5. Het terugmeldcircuit van de schrijfpoort van een 1-bit *VAR* component kan met slechts 2 NMOS transistoren worden gerealiseerd.

   [**Lit.**] C.H. van Berkel and R.W.J.J. Saeijs *Latch with write acknowledge.* NL. Patent Application 9000544, 1990.

6. Het toepassen van inverse logica (**false** correspondeert met de voedingsspanning en **true** met 0 Volt) bij de CMOS realisatie van 4-fase handshake componenten levert een aantrekkelijk voordeel op in schakelsnelheid.

7. Voor een gegeven schermgrootte kan het rasteren van orthogonale rechthoeken in "constante tijd" worden uitgevoerd. Met hedendaagse CMOS technologie kan dit bovendien ruim binnen 1 μ seconde.

   [**Lit.**] C.H. van Berkel and R.H.W. Salters. *Box addressable memory with decision tree.* US. Patent 4845678.

8. Door de afwezigheid van globale synchronisatie en door de vergaande mogelijkheden voor gedistribueerde besturing leent CSP zich bij uitstek voor het beschrijven van flexibele productiesystemen zoals kanban.

   [**Lit.**] Ronald W.J.J. Saeijs and C.H. (Kees) van Berkel. The Design of the VLSI Image-Generator ZaP. In *Proceedings of the 1988 IEEE Int. Conf. on Computer Design: VLSI in Computers & Processors*, pages 163–166, 1988.

   [**Lit.**] R.J. Schönberger, *Japanese Manufacturing Techniques, Nine Hidden Lessons in Simplicity*, The Free Press, New York, 1982.

9. De mogelijke rol van de wiskunde in VLSI ontwerp, ontwerpmethoden en ontwerpgereedschappen wordt overschat door wiskundigen en informatici en onderschat door electrotechnici en VLSI ontwerpers.

10. Het is kenmerkend voor de eenvoud van het Nederlandse belastingstelsel dat het aangiftebiljet A de laatste 10 jaar 10 keer is gewijzigd.