An Algebraic Approach to Hardware Description and Verification

by

Luca Cardelli

Doctor of Philosophy

University of Edinburgh

1982

Abstract

We apply algebraic techniques to various aspects of hardware description and verification, with particular emphasis on VLSI (Very Large Scale Integration) circuit design.

A simple and uniform notation for the description of networks of hardware components is introduced. It is shown how to impose planarity constraints, and how to treat regular and repetitive structures in convenient ways.

The notation is applied to several examples of hardware networks. All these examples constitute different levels of description in the process of translating behavioural specifications into VLSI circuits. A formal semantics is given for the topmost level. Algorithms are given for the translation of purely topological planar stick expressions into metric structures from which layouts can be generated.

The implementation of an experimental VLSI design system is described which uses algebraic concepts to hide detailed geometrical information. Geometric layouts are introduced as an abstract data type in a general purpose functional programming language and considerable advantages over traditional design systems are demonstrated with respect to the user interface.

On the semantic side, two different formal frameworks are defined for the description of systems developing in continuous time. The emphasis is again algebraic, and techniques of both denotational and operational semantics are used. In the operational framework nondeterministic systems can be treated in a natural way, and it is possible to precisely formulate the behaviour of synchronous and asynchronous systems and to study their interactions.

Acknowledgements

I would like to thank first of all my supervisor, Gordon Plotkin, for his help, encouragement and precise hints in the development of this thesis.

I regard this thesis as much a product of the Edinburgh environment as my own and I find it difficult to sort out all the ideas great and small which have been transmitted to me during these years. I would like to thank Robin Milner and Matthew Hennessy for continual assistance; Mike Gordon, Glynn Winskel, Kevin Mitchell, Igor Hansen and Marc Snir for helpful talks; Jeff Tansley and Irene Buchanan for getting me involved in VLSI; and the "system people" for providing some concreteness and an enjoyable computational environment.

During my stay I have been supported by a scholarship of the Italian National Research Council and a scholarship of the University of Edinburgh.

Declaration

This thesis has been composed by myself and the work is my own, under the guidance of my supervisor Gordon Plotkin. Parts of Chapters 1 and 2, produced in different form in collaboration with Gordon Plotkin, appear in [Cardelli 81a]. Chapter 3 is [Cardelli 81b], Chapter 4 is [Cardelli 80] and Chapter 5 is an extended version of [Cardelli 82]. Sections 2.5-2.6 have been submitted for publication.

Contents

0. Introduction

0.1 The Past

We begin with a review of those recent developements in the fields of microelectronics, design tools and semantics which are relevant to this thesis.

0.1.1 Microelectronics

During the past few years, the steady progress in microelectronics has reached a point where completely untrained people can be taught, in the span of few weeks, to conceive and design highly complex hardware systems.

This fact may come as a great surprise to two categories of people. On the one side professional hardware designers have seen the complexity of systems growing beyond any control, to the point where the technology is clearly more powerful than the ability to use it. It seems then unlikely that untrained people might do better.

On the other side the average (computer) scientist who has been trained to think that "the hardware is made by the engineers", suddenly discovers that in a couple of months he can design and receive, his pet architecture; one the big manufacturers had thoughtfully refused to consider. However, it may seem unlikely that he can really do it if those expert manufacturers would not.

The fact is that, until recently, the problem of managing the complexity of VLSI (Very Large Scale Integration) systems was not adequately considered. Design methodologies were developed which encouraged circuit efficiency at a very low level, often at the expense of global optimisations and disregarding elegance and structure. The work done by Carver Mead, Lynn Conway and their collaborators [Mead 80] has completely changed this picture. Structured methodologies have proven to be more reliable, to extend smoothly to big systems and in many cases to provide more efficient and totally unexpected solutions. The simplicity reintroduced by structured methodologies allows people to learn quickly and to quickly produce non-optimal but working devices. In many cases the achievements of these newcomers [Conway 80] sound astounding with respect to average industrial products [Steele 80, Rivest 80, Masumoto 80]. Structured methodologies are now beginning to be systematically used by big manufacturers, and the results are equally encouraging [Lattin 81, Mudge 81].

0.1.2 Design Tools

Another key contribution has been the definition of a clean interface between design and fabrication [Mead 80]. While any such interface necessarily introduces some inefficiency, it allows the designer to ignore most of the inessential aspects of the fabrication process. Moreover it seems sensible to expect that in future fabrication processes will be designed to match this kind of interface, so that many of the inefficiencies will disappear [Mikkelson 81].

The coincidence of structured methodologies, clean interfaces and high level of integration, has inspired a sudden and rapidly spreading interest outside the microelectronic environment. It is most fortunate that this sudden "discovery" of VLSI, comes at a moment where the traditional architectures and design techniques used in microelectronics are showing their limits, and where there is a great need for complexity management techniques.

In fact, the management of complexity has always been the main problem in software engineering and programming language design, and structured methodologies are now simply common sense in those areas. The hope is that as a result of the experience gained in software we shall not have to wait long before getting very effective high level tools for hardware design. A subordinate hope is that we shall be able to completely omit the "batch" and "FORTRAN" stages of design tools.

An interesting parallel can be made between the state of hardware design today and the state of software design in the fifties and early sixties. Layouts (the end-product of any VLSI design activity) have many of the characteristics of machine language programs. They are powerful enough to fully exploit the technology and can lead to great efficiency when used at the lowest level. On the other hand they are inscrutable, and difficult to modify, maintain and debug, and very prone to trivial and repetitive (yet fatal) errors. The information they convey is inflexible and absolute, and in general they encourage programming styles which lack clarity and elegance.

Most of the VLSI design tools today are based on layouts. As a consequence of the low level notations used, many of these are concerned with recovering from errors which have already been made, or with recovering structure which has been lost at some previous stage of design.

For example, <u>design rule checkers</u> are needed because people are allowed to draw wires of the wrong thickness, or to put transistors in the wrong places. Again, <u>electrical rule checkers</u> are needed because the low level of primitives allows designers to combine them in meaningless ways. And again, <u>node extractors</u> are needed because the initial description of the circuit is not semantically structured, or because the structure has been flattened out by some other tool.

Other tools are hampered by tasks which are not their own. <u>Graphics editors</u> are sometimes equipped for checking design rules, or even electrical parameters. <u>Simulators</u> are used in the detection of errors which are clearly syntactical, such as wires which fall short of their intended contact points, switches connected in meaningless ways, transistors introduced by accident, power supply lines disconnected or wrongly connected, etc.

Recently, "assembly languages" have been devised ([Locanthi 78] and many others), where symbolic names and locations can be used instead of bare numbers. High level control structures can be used and syntactic correctness checks can be performed so that some of the syntactic properties of the output, like wire thickness, are guaranteed to be correct. The primary task of these tools is however to describe layouts, not computations, and they are strongly process (or process-class) dependent because they aim to give full access to the lowest level of description. For this reason they should still be considered to be low-level tools.

Continuing the analogy, why cannot we have compilers? The features of a general purpose <u>silicon compiler</u> are easily listed: it should be process-independent, it should be able to express any range of architectures at the behavioural level, and given a syntactically correct input it should <u>always</u> produce syntactically correct code. We should be able to formally describe the compiler (i.e. no "hacking") and maybe prove its correctness, or at least believe in it!

The production of a silicon compiler is a very complex problem. We know what the output should be, namely layouts, but we do not know how to produce it and we do not have any clear ideas about what the input should look like. The choice of a convenient input notation might deeply influence (and maybe simplify) the translation process, and conversely translation techniques may impose restrictions on our notation. It is not clear whether we should first fix the notation or study the translations, or proceed by attempts in both directions until some satisfactory meeting point is reached.

As to the linguistic problems, there is no doubt about the advantages of a high level notation, as far as programming is concerned. For example in many cases high-level language programs can be debugged by typechecking and proof-reading, while "tracing" (which corresponds to simulation) is essential for assembly language programs. Moreover, if we consider the elegance of, for example, a one-pass Pascal compiler with respect to an n-pass macro-assembler, we can also clearly see the implementational advantages of a well structured and powerful notation.

The problem of compiling into two-dimensional structures, even if frequently found in design automation problems, seems to be rather new in formal language and compiler construction theory. There is a little recent interesting work [Floyd 80, Forster 81] at the formal language end. Pioneering work towards full-scale silicon compilers is reported in [Johannsen 79] and [Rupp 81]. Unfortunately the vast literature in hardware routing and placement problems does not seem to apply very directly to VLSI; indeed for compilation it is not enough use techniques like general routers which often only solve 95% of each problem.

On the positive side, a series of remarkable design tools for VLSI has emerged in recent years. Many of these tools share many of the criticisms we have expressed, but they are indisputable milestones in their area.

Some design tools computerise boring hand-drawing activities, by using interactive graphics displays. In this class we can mention, for layouts ICARUS [Fairbairn 78], for stick diagrams STICKS [Williams 78] culminating in REST [Mosteller 81], and for cell composition the Chip Assembler [Tarolli 80].

The prototypical text-oriented system is LAP [Locanthi 78], which embeds a very simple graphical notation [Sproull 80] in a general purpose high-level language (this idea comes from standard graphics techniques [Newman 79]). The crudeness of the graphics primitives is compensated for by the ability to use the control constructs of the language for parameterisation and abstraction, achieving an effectiveness far greater than graphics editors (but with very little user-friendliness).

More ambitious systems try to integrate several tools [Buchanan 80], often into a <u>workstation</u> with special purpose programming languages or packages and sophisticated graphic interfaces. On the layout level we have the LISP-based DPL [Batali 81], and on the sticks level MULGA [Weste 81]. Both these systems are truly remarkable, even if the complexity of the former seems excessive. Many similar systems are now being developed; they mostly use a personal computer together with a high resolution colour display and a pointing device.

0.1.3 Semantics

A very sharp distinction should be made between the means and ends of formal <u>description</u> and formal <u>verification</u>. These two activities are often inversely proportional, in the sense that very powerful description systems can be so detailed and complicated as not to allow any general view of the problems (for example, consider quantum mechanics as a way of describing an armchair: can we formally verify that the armchair is comfortable?). Conversely we might have verification systems which at a certain level of abstraction allow us to easily verify any property we want, but which are unable to describe part of the realities we are interested in (suppose we have a nice theory of armchairs and soft materials; what happens if we ship the armchair to a black hole?). Unfortunately one can also come up with questions which require both powerful descriptions and flexible theories (if we do send the armchair to the black hole, will it keep being comfortable?), and the problem is then to maximise the usefulness of the whole system, and not just the descriptive or the proof-theoretical part.

In mathematics some sort of optimality has been reached, if we consider for example how analysis merges smoothly into topology. Not so in computer science; the considerable descriptive success of denotational and algebraic semantics has not yet led to satisfactory theories of programs (even if it has led to satisfactory theories of models). Properties which are considered obvious to programmers escape, on large programs, any verification or even formalisation.

It is well known that concurrent systems are much more difficult to describe and verify than sequential ones. In this field, denotational and algebraic semantics found descriptive difficulties, while powerful descriptive systems like Petri nets do not seem to offer striking advantages for verification purposes.

From this point of view, hardware systems seem to summarise many difficult problems in semantics; they are of extensively concurrent nature, and the behaviour of even the simplest components is difficult to describe and context-dependent.

Hardware is semantically unexplored at intermediate levels. For low level hardware, the main semantic description available is device physics, which is not very helpful when the number of devices exceeds one. Very powerful techniques have been developed in electronics for the study of analog circuits, but are not of general application to digital circuits. For gate-level hardware, we have satisfactory theories like switching theory (for small combinational systems), and automata theory (for larger sequential systems) which however do not work very well for complex systems built out of many parts, like microprocessors. Very little exists between device physics and switching theory, which is unfortunately exactly what we need for low-level VLSI. Moreover automata theory is not very suitable for studying interconnected networks of processors, which is what we need for high-level VLSI.

Part of these problems, which are common to concurrency problems, have been attacked by the use of operational techniques [Plotkin 81], which can conveniently describe concurrency, joined to algebraic techniques [Milner 80], which lead to flexible proof systems. Recent work on synchronous concurrent systems [Milner 81] (which extends smoothly to asynchronous systems) seem to be particularly well suited both to hardware description and verification, as most hardware systems today are internally synchronous.

0.2 This Thesis

The first chapter of this thesis is dedicated to the task of providing a simple and uniform notation for the description of networks of hardware components. The approach is algebraic in nature and derives from work on the syntax of concurrent systems [Milner 79]. After a general introduction to many-sorted algebras (section

1.2), a "pure" formalism of <u>net</u> <u>expressions</u> is introduced in section 1.3, together with a set of equational laws expressing the equivalence of networks. Networks are regarded as graphs with an interface, and together with net expressions they form what we call a net algebra (net algebras are compared to Milner's flow algebras in section 1.5). In section 1.6 we characterise the initial net algebra in terms of particular kinds of graphs, and we prove soundness, completeness and definability theorems with respect to the net expressions and laws. Some additional structure is then imposed on net algebras in view of the use we shall make of them in chapter 2. Section 1.7 treats planar networks, and sections 1.8 and 1.9 introduce the idea of a <u>bunch</u> (a way of structuring interfaces) which is essential when programming in net algebras. Some bizarre examples of net algebras are given in section 1.10 in order to explore the power of the formalism, while section 1.11 introduces a hardware network which will be used as an example throughout chapter 2.

The second chapter applies the notation developed in the first chapter to several examples of hardware networks. All these examples constitute different levels of description (i.e. different net algebras) in the process of translating behavioural specifications into VLSI circuits. Even if we occasionally attack the problem of algorithmic translations into two-dimensional structures, we concentrate in general on formalisms which can be considered as prototype textual languages for silicon assemblers and compilers, on much the same lines as [Rem 81]. This leaves uncovered a wide area of research, namely graphical languages and graphical interaction. Although it is rather natural to imagine graphical counterparts for some of the textual programming constructs we present, it is not clear how to define purely graphical systems of the same power as text-oriented systems (see [Trimberger 79] for an effort towards integrated text-graphics systems). This is mostly due to the lack of a good graphical analog for parameterisation. Hence, even if we think that graphical interfaces are essential to easily usable systems, we generally concentrate on textual expressions denoting graphical entities.

The topmost level of description, called Clocked Transition Algebra (CTA, section 2.3), is concerned with the behavioural specification of synchronous systems. A formal semantics of CTA is given by a translation to Synchronous CCS [Milner 81]. Section 2.4 describes the CSA model of switch-level hardware [Hayes 81, Bryant 81] and gives a semantics to the stable CSA circuits. A translation mapping every CTA expression into a CSA circuit is then shown. In section 2.5 we work with (planar) stick diagrams, showing several examples of net algebra programming activity. A translation from CTA to sticks is briefly sketched. Section 2.6 treats grids, which are stick diagrams disposed on orthogonal lines. The algebra of grids is very important as an intermediate step in the translation of purely topological stick diagrams into geometrical layouts. An efficient stretching algorithm for grids is developed; then a translation from sticks to grids is described, which has the property of always succeeding in every admissible context (a context expresses constraints on the position of connection points on a rectangular boundary). Finally we comment that translations from grids into layouts have already been experimented with (e.g. [Mosteller 81]).

The third chapter describes the implementation of an experimental VLSI design system (constituting what is generally called a silicon assembler) where most of the geometry-related characteristics of layouts are hidden by the use of algebraic operations. In section 3.2 we introduce the basic data type of pictures (layouts), which is embedded in a general purpose programming language [Gordon 79a] allowing parameterisation and conditional assemblies of pictures. Bunches, and their use in association with an iteration construct, are described in sections 3.3 and 3.4. Section 3.5 deals with an interpretation of a net algebra operator which embodies a form of geometrical river routing. The remaining sections describe various aspects of the implementation.

The purpose of the fourth and fifth chapters is to provide a framework where formal proofs concerning the low level behaviour of hardware systems can be carried out. The fourth chapter describes a formalism in which systems developing through continuous time can be expressed. The emphasis is again algebraic, and algebraic laws are formulated which express the behaviour of such systems (section 4.3). Techniques of denotational semantics are used to provide a deterministic model (section 4.4); the attempt to extend the treatment to nondeterministic systems encounters technical difficulties and another approach is used in chapter 5. A discussion about the expressive power of this formalism is contained in sections 4.5, 4.6 and 4.7. Section 4.8 is dedicated to an example (flip-flops) which exhibits metastable behaviour.

The semantic techniques used in chapter 5 are operational, with the advantage that a semantics can be given to nondeterministic systems in a natural way. This chapter follows [Milner 81] and can be regarded as an extension of that work where a discrete time domain is replaced by a continuous one. Section 5.1 introduces the main ideas and the operational semantics methodology. After a section studying deterministic systems (5.2), nondeterminism is introduced in two orthogonal ways in section 5.3 by a choice operator and an indefinite-duration operator. Communication is treated in section 5.4 and recursion in section 5.5, where some difficulties due to the density of time have to be solved. The following three sections (5.6, 5.7 and 5.8) discuss the complex interactions between synchronous and asynchronous systems, and section 5.9 gives a way of characterising synchronous, non-synchronous and asynchronous systems.

Appendix I introduces the notation used for expressing the syntax of languages, and appendix II contains a list of the symbols used through this thesis.

1. Algebra of Networks

1.1 Introduction

A network is to a first approximation a finite graph. Our main concern is with structured network design, and we are interested in methods and notations for building and analysing networks in a hierarchical fashion. Hence the first problem we have to solve is how to express finite graphs, considered as unstructured sets of arcs, in some orderly and structured way.

The simplest way of exhibiting a graph is of course by displaying it. This kind of presentation is expressive and immediately understandable by humans, but unfortunately it also has several disadvantages.

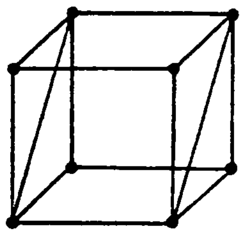


Figure 1.1 A graph

First of all the structure of the graph is not evident in its picture, i.e. we cannot tell how it was built; the mere picture of the graph hides the intended way of looking at a particular graph among the several ways in which the graph can be constructed. Hence some structure (graphical or otherwise) has to be superimposed on the graph in order to understand it in terms of its components.

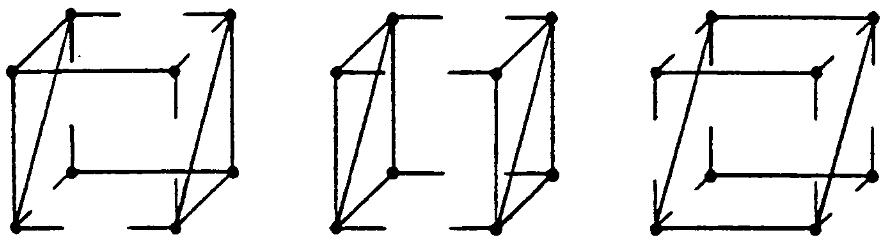


Figure 1.2 Decompositions

Second, graphical notation is not suitable for direct mathematical manipulation. Mathematical coding has to be used in order to get the benefits of formal treatment, and an effort should be made to keep the coding not too different from the intended structure of the coded object, otherwise an obscure theory will result.

Third, graphical notation does not make a good programming language; not because it is difficult to "type it in" (this can be overcome by graphical editors) but because the usual programming language control structures and parameterisation mechanisms are not easily definable on pictures.

Fourth, and finally, no matter how we express them, graphs may have to be represented in terms of data structures in a computer, and operations have to be carried out upon them; then this is just another aspect of the problem of finding a non-graphical notation for manipulating graphs in useful ways.

Our aim is then to develop a notation for structured graphs which is formally tractable, expressive enough to be used as a programming language, and easily convertible into useful data structures. The central idea is to have an abstract data type of networks over which certain operations can be performed (particularly composition of subnetworks) and which can be easily translated into different data types for different purposes. We formalise these ideas in an algebraic framework where abstract types are algebras and easy translations are different shades of algebra morphisms.

This chapter is mostly technical; the reader is advised to skim it in case of difficulties and to come back to it when needed while reading chapter 2. Sections 1.10 and 1.11 contain examples which give some motivation for the notation introduced here.

1.2 Many-sorted Algebras

An **Algebra** is a set together with some operations on its elements. Intuitively the base set of an algebra is a data type, and the operations are the basic operations allowed on that data type; other operations can be defined from the basic ones [Gratzer 79].

A **many-sorted algebra** is an extension of this idea, where we have several sets instead of one (hence several data types) and _typed_ operations which take arguments from and produce results in these sets [Goguen 78]. The extension from single-sorted to many-sorted algebras is conceptually very simple, but makes the technical treatment considerably heavier. In fact operations have to be indexed by their type, and we have to distinguish operators having the same name but belonging to different algebras. All this typing and naming information is gathered into the notions of _sort_ and _signature_.

A sort is a _data-type name_; sorts will be denoted by the letter s, sets of sorts by S and lists of sorts by $w \in S^*$ (with [] the empty list).

Definition 1.1 A **signature** $\underline{\Sigma}$ is a pair $\langle S,\Sigma\rangle$ where S is a set (of sorts) and Σ is a family of sets (of operator symbols) indexed by $S^*\times S$. An operator symbol $\alpha_{w,s} \in \Sigma_{w,s}$ has rank (or functionality) w,s arity w and sort s □

Example: Boolean

$\underline{\Sigma} = \langle\ S = \{bool\},$

$\Sigma = \{\ \Sigma_{[],bool} = \{true,false\},$

$\Sigma_{bool,bool} = \{\ \sim\ \},$

$\Sigma_{bool\ bool,bool} = \{\ \vee,\wedge\ \},$

$\Sigma_{w,s} = \emptyset$ for any other w,s } $\rangle$

We denote by $X = \{X_s | s\in S\}$ a set of sets of variables of sort s. Variables are all distinct, and they are distinct from operator symbols and punctuation.

Definition 1.2 A $\underline{\Sigma}$**(X)-expression** is a syntactic expression built from the operator symbols of $\underline{\Sigma}=\langle S,\Sigma\rangle$, the variables of $X=\{X_s|s\in S\}$ and the distinguished symbols "(", ")" and ","; more precisely, expressions are all and only the strings of symbols obtained by the following rules:

- If x is a variable of sort s, then x is an expression of sort s.
- If $e_1..e_n$ are expressions of sort $s_1..s_n$ $(n\geq 0)$

  and $\alpha_{s_1..s_n,s} \in \Sigma_{s_1..s_n,s}$ then:

  $\alpha_{s_1..s_n,s}(e_1,...,e_n)$

  is an expression of sort s

(where, for n=0, $\alpha_{[],s}()$ has sort s)

□

When there is no ambiguity subscripts are omitted, so that we simply write $\alpha(e_1,...,e_n)$.

Example: Boolean expressions

The following are $\Sigma(X)$-expressions, where Σ is the boolean signature and $X = \{X_{bool}=\{x,y\}\}$.

```
x
true()
~(∧(x,∨(false(),y)))
```

We use the following notation for cartesian products of sets:

$$A_{[\,]} = \{[\,]\}$$
$$A_w = A_{s_1 \ldots s_n} = A_{s_1} \times \ldots \times A_{s_n}$$

Definition 1.3 A Σ-algebra $\underline{A}$ (with $\Sigma = \langle S, \Sigma\rangle$) is a pair $\langle A^{\cdot}, A\uparrow\rangle$ where $A^{\cdot}$ is an S-indexed set of sets A_s and $A\uparrow$ is an $S^*\times S$-indexed set of maps $A_{w,s}$: $\Sigma_{w,s} \longrightarrow (A_w \longrightarrow A_s)$ associating a function $A_{w,s}(\alpha_{w,s})$: $A_w \longrightarrow A_s$ with each operation symbol $\alpha_{w,s} \in \Sigma_{w,s}$ □

Each A_s is called the carrier of $\underline{A}$ of sort s; each $A_{w,s}(\alpha_{w,s})$ is called the operator of $\underline{A}$ named by $\alpha_{w,s}$, and is also denoted by $\phi^A_{\alpha_{w,s}}$. When there is no ambiguity $\phi^A_{\alpha_{w,s}}$ is also written $\phi_{\alpha_{w,s}}$, ϕ^A_α, ϕ_α or even α.

Example:

```
A = < A· = { A_bool = {T,F}},
      A↑ = { A_[],bool = {<true,T>, <false,F>},
             A_bool,bool = {<~, Not={<T,F>,<F,T>}>},
             A_bool bool,bool =
               {<∧, And={<<T,T>,T>,<<T,F>,F>,
                         <<F,T>,F>,<<F,F>,F>},
                 ∨,  Or={<<T,T>,T>,<<T,F>,T>,
                         <<F,T>,T>,<<F,F>,F>}>},
             A_w,s = ∅ for any other w,s} >
```

with $\phi^A_{true_{[],bool}} = T$,

$\phi^A_{false_{[],bool}} = F$,

$\phi^A_{\sim_{bool,bool}} = Not$,

$\phi^A_{\wedge_{bool\ bool,bool}} = And$,

$\phi^A_{\vee_{bool\ bool,bool}} = Or$

Expressions are a very important example of algebras:

Definition 1.4 $T_{\underline{\Sigma}}(X)$ (where $\underline{\Sigma} = \langle S,\Sigma\rangle$) is the $\underline{\Sigma}$-algebra with:

- Carriers: the set of $\underline{\Sigma}(X)$-expressions of sort $s \varepsilon S$.
- Operations: the mappings

  $\phi_{\alpha_{s_1..s_n,s}} : e_1..e_n \mapsto \alpha_{s_1..s_n,s}(e_1,...,e_n)$ $(n \geq 0)$

  for each $\alpha_{s_1..s_n,s}$ ε $\Sigma_{\alpha_{s_1..s_n,s}}$

  and expressions $e_1..e_n$ of sort $s_1..s_n$

  (for n=0 we have $\phi_{\alpha_{[],s}} : [] \mapsto \alpha_{[],s}())$

□

It is easily verified that $T_{\underline{\Sigma}}(X)$ is really a $\underline{\Sigma}$-algebra.

We finally include the definition of homomorphism and of signature morphism which are the formal basis for the translations which we shall discuss in Chapter 2 (even if those translations will only approximate the idea of homomorphism).

Definition 1.5 A $\underline{\Sigma}$-homomorphism of $\underline{\Sigma}$-algebras

$h: \underline{A} \rightarrow \underline{B}$

is an S-indexed set of maps $h_s: A_s \rightarrow B_s$ such that

$$h_s(\phi^A_{\alpha_{w,s}}(a_1,...,a_n)) = \phi^B_{\alpha_{w,s}}(h_{s_1}(a_1),...,h_{s_n}(a_n))$$

for all $s \varepsilon S$, $w=s_1...s_n \varepsilon S^*$ and $a_1 \varepsilon A_{s_1}$, ... $,a_n \varepsilon A_{s_n}$ □

Definition 1.6 A **signature morphism** ρ from $\langle S,\Sigma\rangle$ to $\langle S',\Sigma'\rangle$ is a pair $\langle f,g\rangle$ consisting of a map $f\colon S \longrightarrow S'$ and a family of maps

$$g_{w,s}\colon \Sigma_{w,s} \longrightarrow \Sigma'_{f^*(w),f(s)} \ \Box$$

A signature morphism is a (possibly many-to-one) renaming of sorts, together with a compatible (possibly many-to-one) renaming of operator symbols.

Theorem 1.1 For every Σ-algebra A and map $h\colon X \longrightarrow A$ (i.e. family $h_s\colon X_s \longrightarrow A_s$ for $s\varepsilon S$) there is a unique Σ-homomorphism $h^*\colon T_\Sigma(X) \longrightarrow A$ such that

$$h^* \circ I_X = h$$

where $I_X\colon X \longrightarrow T_\Sigma(X) : x \mapsto x$ is the injection of generators

$\Box$

The above theorem states the existence of a unique homomorphism h^* from $\Sigma(X)$-expressions ($T_\Sigma(X)$) and environments for free variables ($h\colon X \longrightarrow A$) into any Σ-algebra A. This homomorphism is often called **evaluation** or **interpretation** of an expression e in an algebra, and $h^*(e)$ is called the (because of uniqueness) **value** of e in A (with respect to an environment).

1.3 Net Algebras

Refining our idea of network, we can say that a network is a finite graph with an interface. Interfaces are an abstraction mechanism; they contain all the information about the network which is needed and visible from "outside", while hiding the internal structure. For example, syntactic checks can be performed on network operations on the basis of the information contained in the interfaces they operate onto; operations are guaranteed to be meaningful if they satisfy these syntactic checks.

Star =

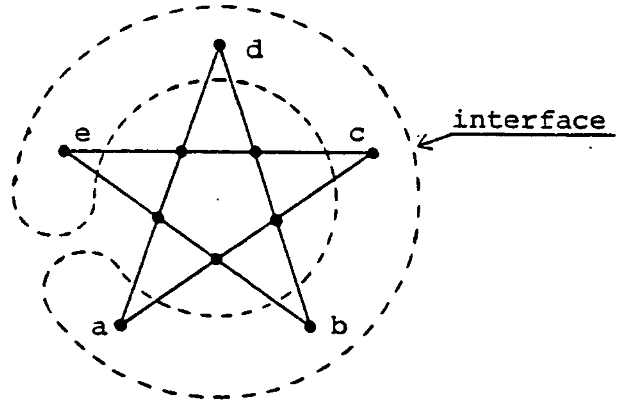


Figure 1.3 A graph with an interface

The interface of a network consists of ports which have a name and a type. Names are used to denote edges of the network (i.e. connection points), and types guarantee the consistency of certain operations. The most important use of interfaces is in joining networks together into larger networks; the join is done by naming the ports to be connected, provided that there are no name clashes and that the types of the connected ports match.

1.3.1 Sorts

Formally, an interface is a sort. Given a set Types of types and a set PortNames of (port) **names** (with a,b ranging over names and A,B,C ranging over finite sets of names), a sort is a map $s: A \longrightarrow$ Types with $\lceil s \rceil \triangleq A$; hence s(a) shows the type of the port named a. We say that two sorts s,s' are **compatible** if their common port names have the same type, i.e. if $s{\downarrow}B = s'{\downarrow}B$, where $B = \lceil s \rceil \cap \lceil s' \rceil$.

1.3.2 Signatures

Networks are built out of a given set $\mathbb{L}$ of basic components called literals (nullary operators). Every literal $l \in \mathbb{L}$ has a sort given by $\lambda(l)$.

The unary restriction operator, \a, removes the name a from the sort of a network. For every a and s we have an operator $\backslash a: s \longrightarrow s'$

where $\lceil s' \rceil = \lceil s \rceil \setminus \{a\}$ and $s'(a') = s(a')$ for a' in $\lceil s' \rceil$. Restriction is a postfix operator, and we abbreviate $x\backslash a_1 \ldots \backslash a_n$ to $x\backslash a_1 \ldots a_n$.

Star\a =

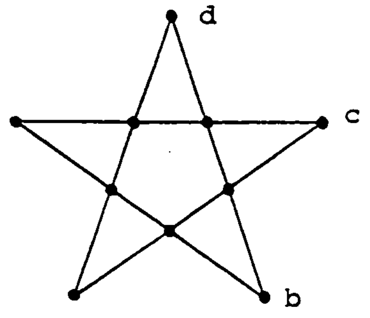


Figure 1.4 Restriction

The **unary renaming operator**, $\{r\}$, changes the names of a sort without changing the port types. For every sort s and bijection r: $\lceil s \rceil \longrightarrow A'$ we have an operator $\{r\}$: $s \longrightarrow s'$ where $s' = s \circ r^{-1}$. Restriction is postfix and we write $\{a_1\backslash b_1, \ldots, a_n\backslash b_n\}$ for r when $\{a_i\} \subseteq A$, $r(a_i)=b_i$ and $r(a)=a$ for a not in $\{a_i\}$ (hence $\{\}$ is the identity renaming).

Star{a\f, b\g} =

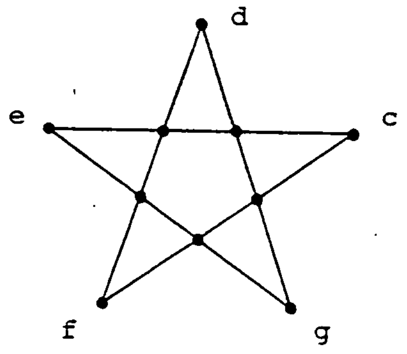


Figure 1.5 Renaming

The binary **composition** operator, $|$, composes two networks together identifying and then forgetting their common port names. For every compatible pair of sorts s,s' we have an operator $|: s,s' \longrightarrow s''$ where $s'' = s \oplus s' : A \oplus A'$ (we use $\oplus$ for symmetric difference: $A \oplus A' = (A\backslash A') \cup (A'\backslash A)$, and $s \oplus s' = s{\downarrow}(A\backslash A') \cup s'{\downarrow}(A'\backslash A)$). Composition is an infix operator associating to the left.

A useful derived operation is **explicit composition**, $[r]$, which composes two networks by linking the ports which are explicitly mentioned in a bijection r. The operator $[r]$: $s,s' \longrightarrow s''$ with $r: A \longrightarrow A'$, $B=\lceil s\rceil\backslash A$ and $B'=\lceil s'\rceil\backslash A'$, is well defined iff

(i) $A\subseteq\lceil s\rceil$ and $A'\subseteq\lceil s'\rceil$

(ii) $s(a) = s'(r(a))$ for every a in A (type restriction)

(iii) $B\cap B' = \emptyset$ (no name clashes)

Then $\lceil s''\rceil = B\cup B'$ and $s''(b)$ is $s(b)$ if $b \in B$ and $s'(b)$ otherwise.

Under these conditions we define

$$e[r]e' \triangleq e\{r\cup id_B\}|e' = e|e'\{r^{-1}\cup id_{B'}\}$$

Explicit composition is infix and left associative; $e[r]e'$ will be written as $e[a_1\text{--}b_1,\ldots,a_n\text{--}b_n]e'$ for $\langle a_i,b_i\rangle \in r$.

(Star\d) [c--e, b--a]
(Star\d) =

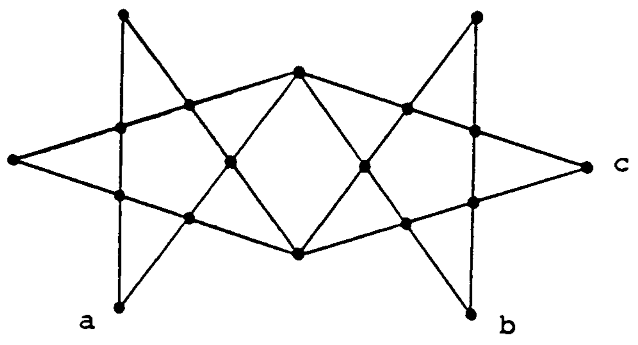


Figure 1.6 Composition

1.3.3 Net Expressions and Laws

From the signature of a net algebra, and for a given set of literals, we can construct a corresponding set of net expressions (ranged over by e):

- literals are expressions
- if e,e' are expressions

  then $(e\backslash a)$, $(e\{r\})$ and $(e|e')$ are expressions.

Parentheses will often be omitted.

The operators we have so far defined must obey a set of laws called the net laws, which complete our definition of <u>net</u> <u>algebras</u>. We write $\sigma(e)$ for the sort of e, and we require the following equations to hold whenever they are well-formed according to our previous remarks.

[|] $e \mid e' = e' \mid e$

[||] $(e \mid e') \mid e'' = e \mid (e' \mid e'')$

$\quad$ if $\lceil\sigma(e)\rceil \cap \lceil\sigma(e')\rceil \cap \lceil\sigma(e'')\rceil = \emptyset$

[\\] $e \setminus a = e \qquad$ if $a \notin \lceil\sigma(e)\rceil$

[\\\\] $(e \setminus a) \setminus b = (e \setminus b) \setminus a$

[\\|] $(e \mid e') \setminus a = (e \setminus a) \mid (e' \setminus a) \qquad$ if $a \in \lceil\sigma(e|e')\rceil$

[{}] $e\ \{id\} = e$

[{}{}] $(e\ \{r\})\ \{r'\} = e\ \{r' \circ r\}$

[{}\\] $(e\ \{r\}) \setminus (ra) = (e \setminus a)\ \{r'\}$ where $r' = r{\downarrow}(\lceil r\rceil \setminus a)$

[{}|] $(e \mid e')\ \{r \uplus r'\} = (e\ \{r \uplus r''\}) \mid (e\ \{r' \uplus r''\})$

$\quad$ where $\lceil r\rceil = \lceil\sigma(e)\rceil \setminus \lceil\sigma(e')\rceil$, $\lceil r'\rceil = \lceil\sigma(e')\rceil \setminus \lceil\sigma(e)\rceil$

$\quad$ and $\lceil r''\rceil = \lceil\sigma(e)\rceil \cap \lceil\sigma(e')\rceil$

Derived laws for explicit composition are as follows:

[[]] $e[r]e' = e'[r^{-1}]e$

[[][]] $(e[id_{A_1}]e')[id_{A_2 \uplus A_3}]e'' = e[id_{A_3 \uplus A_1}](e'[id_{A_2}]e'')$

$\quad$ whenever all the compositions are well formed

[{}[]] $e[r' \circ r]e' = e\{r \uplus id_B\}[r']e'$

[[]\\] $(e[r]e') \setminus a = (e \setminus a)[r](e' \setminus a)$ if $r\colon A_1 \longrightarrow A_2$ and $a \notin A_1 \uplus A_2$

[[]{}] $(e'[r]e'')\{r' \uplus r''\} = (e'\{r' \uplus r_1'\})[r_1'' \circ r \circ r_1'](e''\{r'' \uplus r_1''\})$

1.3.4 More on Net Expressions

Net expressions can be used as the kernel of a programming language for networks. We give some definitions which can guide the implementation of net expressions, particularly regarding their syntactic correctness. A formal syntax for net expressions is introduced, and algorithms are given for checking whether a net expression is well formed and for extracting its sort.

The formal syntax of net expressions is defined here, using the metasyntactic notation of Appendix I:

```
literal ::=  ...          (depending on the particular algebra)
exp ::= literal |
        exp '\' name |
        exp '{' {name '\' name / ','} '}' |
        exp '|' exp |
        exp '[' {name '--' name / ','} ']' exp |
        '(' exp ')'
```

Restriction and renaming bind stronger than explicit composition, which binds stronger than implicit composition. Both kinds of composition are left associative.

A sorting $\underset{\sim}{e}$ of a net expression e is an assignment of a sort to every subexpression e' of e; for example $(c_s[a\text{--}b]c'_{s'})_{s''}$ is a sorting of c[a--b]c'.

A well-sorting of e is a sorting $\underset{\sim}{e}$ such that the predicate WellSorted($\underset{\sim}{e}$) (defined below) is true. We then say that e is well-sorted if it admits a well-sorting $\underset{\sim}{e}$.

WellSorted (1_s) =

$s=\lambda(1)$

WellSorted $((e_s\backslash a)_{s'})$ =

WellSorted(e_s) and $s'=s\downarrow\lceil s\rceil\backslash\{a\}$

$\text{WellSorted}\ ((e_s\{a_i\backslash b_i\})_{s'}) =$

$\quad \text{WellSorted}(e_s) \text{ and NameBijection}(\{\langle a_i,b_i\rangle\})$

$\quad \text{and } \{a_i\} \subseteq \lceil s \rceil \text{ and } (\lceil s \rceil \backslash \{a_i\}) \cap \{b_i\} = \emptyset$

$\quad \text{and } s' = (s{\downarrow}\lceil s \rceil \backslash \{a_i\}) \cup \{\langle b_i, s(a_i)\rangle\}$

$\text{WellSorted}\ ((e_s | e'_{s'})_{s''}) =$

$\quad \text{WellSorted}(e_s) \text{ and WellSorted}(e'_{s'})$

$\quad \text{and } \forall a \in \lceil s \rceil \cap \lceil s' \rceil.\ s(a) = s'(a)$

$\quad \text{and } s'' = s{\downarrow}(\lceil s \rceil \cap \lceil s' \rceil) \cup s'{\downarrow}(\lceil s \rceil \cap \lceil s' \rceil)$

$\text{WellSorted}\ ((e_s[a_i\text{--}b_i]e'_{s'})_{s''}) =$

$\quad \text{WellSorted}(e_s) \text{ and WellSorted}(e'_{s'})$

$\quad \text{and NameBijection}(\langle\{a_i,b_i\rangle\})$

$\quad \text{and } a_i \in \lceil s \rceil \text{ and } b_i \in \lceil s' \rceil$

$\quad \text{and } s(a_i) = s'(b_i) \text{ and } (\lceil s \rceil \backslash \{a_i\}) \cap (\lceil s' \rceil \backslash \{b_i\}) = \emptyset$

$\quad \text{and } s'' = s{\downarrow}(\lceil s \rceil \backslash \{a_i\}) \cup s'{\downarrow}(\lceil s' \rceil \backslash \{b_i\})$

$\text{NameBijection}\ (\{\langle a_k,b_k\rangle\}) =$

$\quad i \neq j \Rightarrow a_i \neq a_j,\ b_i \neq b_j$

The following procedure, SortOf, computes the sort of a well-sorted net expression. It is easily verified that WellSorted($\underset{\sim}{e}$) is true, where $\underset{\sim}{e}$ is the sorting generated by applying SortOf to all the subexpression of e.

$\text{SortOf}(1) \quad = \lambda(1)$

$\text{SortOf}(e\backslash a) = \text{SortOf}(e){\downarrow}\lceil \text{SortOf}(e) \rceil \backslash a$

$\text{SortOf}(e\{a_i\backslash b_i\}) =$

$\quad (\text{SortOf}(e){\downarrow}\lceil \text{SortOf}(e) \rceil \backslash \{a_i\}) \cup \{\langle b_i, \text{SortOf}(e)(a_i)\rangle\}$

$\text{SortOf}(e|e') =$

$\quad \text{let } A = \lceil \text{SortOf}(e) \rceil \cap \lceil \text{SortOf}(e') \rceil$

$\quad \text{in SortOf}(e){\downarrow}A \cup \text{SortOf}(e'){\downarrow}A$

$\text{SortOf}(e[a_i\backslash b_i]e') =$

$\quad \text{SortOf}(e){\downarrow}\lceil \text{SortOf}(e) \rceil \backslash \{a_i\} \cup \text{SortOf}(e'){\downarrow}\lceil \text{SortOf}(e') \rceil \backslash \{b_i\}$

1.4 Net Morphisms

A **net morphism** is a homomorphism of net algebras. Given two net algebras $\underline{A}$ and $\underline{B}$ over the same signature (i.e. over the same set of literals $\mathbb{L}$), a morphism $h:\underline{A} \rightarrow \underline{B}$ is a set of maps

$$\{h_s : A_s \rightarrow B_s \mid s \in \text{NetSort}\}$$

such that:

$$h_s(1_s^A) = 1_s^B \qquad \forall\ 1 \in \mathbb{L}$$
$$h_{s'}(e_s \backslash_{s,s'}^A a) = (h_s(e_s)) \backslash_{s,s'}^B a$$
$$h_{s'}(e_s \{r\}_{s,s'}^A) = (h_s(e_s)) \{r\}_{s,s'}^B$$
$$h_{s''}(e_s \mid_{s\ s',s''}^A e'_{s'}) = h_s(e_s) \mid_{s\ s',s''}^B h_{s'}(e'_{s'})$$

1.5 Net Algebras and Flow Algebras

Net algebras are modelled on Milner's Flow Algebras [Milner 79]. The main difference is that in flow algebras many-to-many port connections are possible, while in net algebras we have one-to-one connections of ports and connected ports are forgotten in the sort of the result. One-to-one connections seem to reflect more accurately some of our intended applications, particularly in the case of connecting geometric objects. In Chapter 3 for example we define composition so that the connection of two geometric ports does not leave "space" for any other connection, and the connected ports may as well disappear from the sort of the result.

The formal treatment of net algebras shows that the theory and the set of laws we obtain are about as nice as in the case of flow algebras. However, the relationships between the two theories need some further study. On the one hand, it is easy to mimic net algebras in the flow algebra framework; for example the explicit composition e[a—b]e' (with the usual restrictions) is definable in terms of flow algebra composition, restriction and renaming as (e{c\a}|e'{c\b})\c (with c new) and the net laws are then derivable from the flow laws. On the other hand, net expressions cannot easily define flow algebra expressions because the latter may connect each of their ports to an unlimited number of other ports. A solution could be to define flow algebra composition in the net algebra framework in the following way: any time that we have to connect a port, we first "fork" it into two ports (by composition with a three-port forking literal) and then we connect one of the new ports, leaving the other one free for subsequent connections. Another solution, which might also be useful for different purposes, could be the introduction of net expressions with infinitary sorts: each flow algebra port would be represented by an infinite number of indexed net algebra ports, and composition would take care of always using the "next" available port.

1.6 The Initial Net Algebra

There is a particularly important net algebra, called the initial net algebra, for which the laws [|]..[{}|] <u>only</u> hold and which is unique up to net isomorphism. The initial net algebra is the one that we implicitely have in mind when we talk about "nets", "graphs" or "pictures" and their abstract properties. It turns out that the formalisation is not so intuitive, but it allows us to give a formal justification for our laws and to investigate their darkest details.

The initial net algebra can be built by standard algebraic techniques, quotienting the set of net expressions by the congruence relation generated by the net laws [Gratzer 81]. In this section we look for a more explicit characterisation of the initial net algebra in terms of a suitable kind of graph. The corresponding results for flow algebras can be found in [Milner 79].

We start with some preliminary definitions:

- **PortNames**, is the countable set of port names, with a,b ranging over port names and A,B ranging over finite sets of port names;
- **Types**, is the set of port types;
- $\mathbb{L}$, is the set of literals l (nullary operators);
- **Sorts**, is the set of functions s: PortNames $\longrightarrow$ Types associating a type to each port name (where $\lceil s \rceil \triangleq$ domain(s) is finite);
- λ: $\mathbb{L} \longrightarrow$ Sorts, associates a sort to each literal.

Definition 1.7

A **network** is a quintuple $\langle V,\gamma,A,\pi,E\rangle$, where:

- **V** (the set of vertices), is a non-empty finite set, with $v \in V$;
- γ: V $\longrightarrow \mathbb{L}$ (the interpretation mapping), associates a literal to each node of the network;
- A $\subseteq$ PortNames (the set of port names), is a finite set;
- P is the set of the ports $\{\langle v,a\rangle \mid v \in V$ and $a \in \lceil \lambda(\gamma(v)) \rceil\}$; where each port is a pair $\langle v,a\rangle$ (vertex-portname) such that a is a port name of the literal associated with v;
- π: A $\longrightarrow$ P (the naming mapping), is 1-1;
- **type**: P $\longrightarrow$ Types, defined as type(v,a) $\triangleq$ $\lambda(\gamma(v))(a)$
- E $\subseteq$ P X P (the edges), is a relation on ports satisfying:

1. E is symmetric and a partial function.
2. If $\langle v,a\rangle E \langle v',a'\rangle$ then $v \neq v'$ and type(v,a)=type(v',a').
3. No $\langle v,a\rangle$ is both in the domain of E and equal to $\pi(b)$ for some b.

$\square$

Condition 1. ensures that connection is symmetric and any port is connected to at most another one. Condition 2. excludes self-loops and ensures type-consistency. Condition 3. ensures that no port is both named (i.e. externally connectable) and connected.

Definition 1.8 A net isomorphism ρ: $\langle V,\gamma,A,\pi,E\rangle \simeq \langle V',\gamma',A',\pi',E'\rangle$ is an isomorphism ρ: $V \simeq V'$ such that:

$\gamma = \gamma' \circ \rho$;

$A' = A$;

$\pi' = (\rho \# id_A) \circ \pi$;

$\langle v,a\rangle E \langle w,b\rangle \Rightarrow \langle \rho(v),a\rangle E' \langle \rho(w),b\rangle$

□

where $(f\#g)\langle a,b\rangle \triangleq \langle f(a),g(b)\rangle$.

Remark: We do not distinguish between isomorphic networks.

Definition 1.9 The sort of a network $N=\langle V,\gamma,A,\pi,E\rangle$ is s: $A \longrightarrow$ Types with $s(a) \triangleq type(\pi(a))$ and $\lceil s \rceil \triangleq A$ □

Definition 1.10 The operations on networks are defined as follows:

$1 \triangleq \langle\{1\},\ 1 \mapsto 1,\ A,\ a \varepsilon A \mapsto \langle 1,a\rangle,\ \emptyset\rangle$ where $A = \lceil \lambda(1) \rceil$

$\langle V,\gamma,A,\pi,E\rangle \backslash a \triangleq \langle V,\gamma,A\backslash\{a\},\pi\backslash a,E\rangle$

$\langle V,\gamma,A,\pi,E\rangle\{r\} \triangleq \langle V,\gamma,B,\pi \circ r^{-1},E\rangle$ where r:$A \longrightarrow B$

$\langle V,\gamma,A,\pi,E\rangle \mid \langle V',\gamma',A',\pi',E'\rangle \triangleq \langle V \cup V',\gamma \cup \gamma',A \oplus A',\pi'',E''\rangle$

where $C = A \cap A'$

and $s \downarrow C = s' \downarrow C$

and $\pi'' = \pi \downarrow (A\backslash C) \cup \pi' \downarrow (A'\backslash C)$

and $E'' = E \cup E' \cup \{\langle \pi a,\pi' a\rangle,\langle \pi' a,\pi a\rangle \mid a \varepsilon C\}$

where we assume $V \cap V' = \emptyset$.

□

Theorem 1.2 The operations are well defined:

(1) $\forall l \in \mathbb{L}.\ \langle\{l\},\ l \mapsto l,\ A,\ a \in A \mapsto \langle l,a\rangle,\ \emptyset\rangle$, where $A=\lceil\lambda(l)\rceil$, is a network;

(2) if N is a network, so is N\a;

(3) if N is a network and r a bijection on ⌈s⌉, then N{r} is a network;

(4) if N,N' are networks and s,s' are compatible, then N|N' is a network;

(5) the operations have the correct type.

Proof In Appendix to this chapter ▯

Every well sorted net expression can be made to denote a network (by interpreting the operations as network operations); the converse is also true:

Theorem 1.3 (Definability)

Every network can be denoted by a well sorted net expression (up to network isomorphism).

Proof In Appendix to this chapter ▯

The net laws are verified:

Theorem 1.4 (Consistency)

Laws [|] .. [{}|] are valid up to network isomorphism.

Proof In Appendix to this chapter ▯

Definition 1.11 Let ≡ be the congruence generated by laws [|] .. [{}|] over net expressions. Two net expressions e,e' are convertible iff e≡e' ▯

Lemma 1.1 (Network Substitution Lemma)

Network isomorphism is a congruence with respect to restriction, renaming and composition ▯

Theorem 1.5 (Soundness)

If $e \equiv e'$, then e and e' denote isomorphic networks.

Proof

By induction on the proof of $e \equiv e'$, using the consistency theorem and the network substitution lemma.

$\square$

Definition 1.12 An **atom** (at) is an expression of the form

$$at = l \backslash A \backslash \{r\}$$

where l is a literal and $\backslash A$ is multiple restriction over all the $a \in A$. $\square$

Definition 1.13 A net expression is in **normal form** (nf) if it has the form

$$nf = at_1 \mid \ldots \mid at_n$$

with 1. $\forall i,j.\ s_i$ compatible with s_j

and 2. $\forall i,j,k$ all different. $\lceil s_i \rceil \cap \lceil s_j \rceil \cap \lceil s_k \rceil = \emptyset$

where $s_i = \sigma(at_i)$

$\square$

Theorem 1.6 (Normal Forms)

Every net expression is convertible to a normal form.

Proof In Appendix to this chapter $\square$

Theorem 1.7 (Completeness)

If e and e' denote isomorphic networks, then $e \equiv e'$.

Proof

(1) If nf and nf' denote isomorphic networks, then nf$\equiv$nf'.

Suppose $nf = at_1 \mid \ldots \mid at_n$ and $nf' = at'_1 \mid \ldots \mid at'_{n'}$. By [|], [||] and condition 2. on normal forms, we can reorder nf and nf' so that there is a bijection between at_i, at'_i and the nodes of the two networks (hence $n=n'$). Let us assume that nf and nf' are already properly ordered; by the properties of the isomorphism for each pair of atoms $at_i = l_i \backslash B_i\{r_i\}$ and $at'_i = l'_i \backslash B'_i\{r'_i\}$ we have $l_i = l'_i$, and $B_i = B'_i$. The renamings r_i and r'_i do not have to be exactly the same, because internal connection can be arbitrarily named. However they must agree on the visible ports, and the internal connections can be renamed as shown in the proof of the normal form theorem. Hence $nf \equiv nf'$.

(2) Let e denote N and e' denote N' with $N \simeq N'$. By the normal form theorem, e and e' have respective normal forms $nf \equiv e$ and $nf' \equiv e'$. By soundness nf denotes $\bar{N} \simeq N$ and nf' denotes $\bar{N'} \simeq N'$. Since $\bar{N} \simeq N \simeq N' \simeq \bar{N'}$, by (1) we have $nf \equiv nf'$. Hence $e \equiv nf \equiv nf' \equiv e'$.

$\square$

Definition 1.14 The net algebra $\underline{N}_{\mathbb{L}}$, (with respect to a set of literals $\mathbb{L}$) has as carriers the networks of sort s for each s, and as operations the network operations.

$\square$

Theorem 1.8 (Initiality)

For each net algebra $\underline{A}$ there is a unique net homomorphism

$$\rho_{\underline{A}}: \underline{N}_{\mathbb{L}} \rightarrow \underline{A}$$

Proof

Let $e_{\underline{B}}$ be the interpretation of the net expression e in the net algebra $\underline{B}$. There is at most one net homomorphism $\rho: \underline{N}_{\mathbb{L}} \rightarrow \underline{A}$ which is determined, because of definability, by:

- $\rho(l) = l \;\; (\forall\; l \in \mathbb{L})$
- $\rho(N\backslash a) = \rho(N)\backslash a$
- $\rho(N\{r\}) = \rho(N)\{r\}$
- $\rho(N|N') = \rho(N)|\rho(N)$

i.e. we have $\rho(e_{\underline{N}_{\mathbb{L}}}) = e_{\underline{A}}$.

We have however to show that ρ is well defined: if e and e' define N and N' with $N \simeq N'$, then we must show that $\rho(N) = \rho(N')$, i.e. that

$e_{\underline{A}} = e'_{\underline{A}}$. By completeness $e = e'$ and so, as $\underline{A}$ satisfies the net laws, $e_{\underline{A}} = e'_{\underline{A}}$.

$\square$

1.7 Planar Networks

In the next chapter we shall often be concerned with **planar networks**, i.e. with networks whose graph is planar. In those situations it is useful to be able to check syntactically whether an expression denotes a planar network, so that we can define precisely the class of meaningful expressions.

It is possible to characterise planar networks by refining our notion of interface. While an interface is just a set of ports, a planar interface is going to be a cyclically ordered set of ports, hereafter called a cycle.

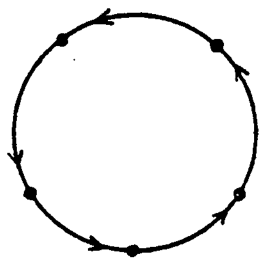


Figure 1.7 A planar interface

Suppose we have a set of planar primitives, with planar interfaces; we need a composition operation preserving planarity and cycles, i.e.

(i) Composition must take pairs of cycles into cycles.

(ii) Composition must take pairs of planar graphs into planar graphs.

A first restriction is imposed on composition in order to guarantee condition (i); the presence of cycles then helps enforcing condition (ii). We require:

(i') The ports being connected must be contiguous in both cycles, so that it is possible to form a new cycle by joining the two cycles around the connection area after having dropped the connected ports.

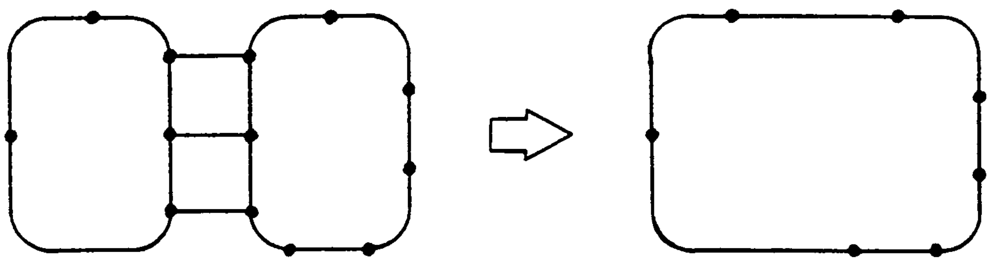


Figure 1.8 Composing cycles

(ii') (Existence) The ports being connected must be inversely ordered in their respective cycles; thus two planar graphs are connected by non-intersecting edges and the result is planar. (Uniqueness) The particular resulting planar graph is not yet completely determined:

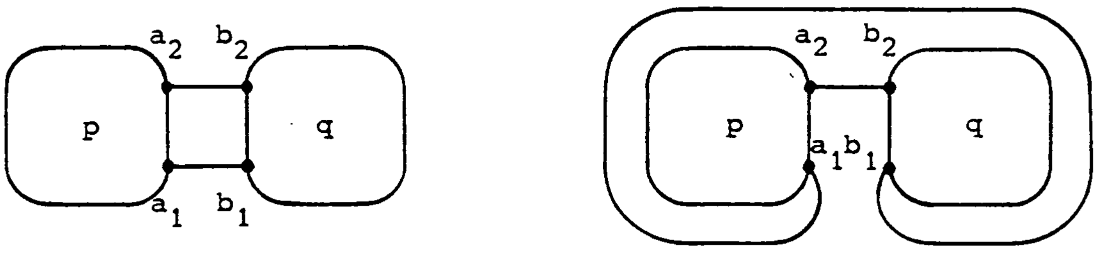


Figure 1.9 Ambiguity of $p[a_1\text{--}b_1, a_2\text{--}b_2]q$

We then impose that in a connection $[a_1\text{--}b_1; \ldots ; a_n\text{--}b_n]$ the oriented arc $a_{i+1}\text{--}b_{i+1}$ be on the "left" of the oriented arc $a_i\text{--}b_i$, with a_{i+1} adjacent to a_i and b_{i+1} adjacent to b_i $(i \in \{1..n-1\})$. Implicit composition is now "$p|_a q$" where a is the starting port of the planar composition, which then proceeds anticlockwise on the sort of p.

The sort of a Planar Network is a pair

$\langle s: A \rightarrow \text{Types}, o: A \rightarrow A\rangle$

where $s: A \rightarrow$ Types is like the sort of a non-planar network, i.e. it is a mapping from a finite set of port names A into port types. The second component of a sort is used to express planarity constraints; a cyclic ordering is imposed on A by a bijection o: A ≃ A which is a cyclic permutation of A. We say that a ε A preceeds a' ε A if o(a)=a' and that a is adjacent to a' if a preceeds a' or a' preceeds a. The ordering induced by the "preceeds" relation is taken to represent the anticlockwise ordering of ports around a graph.

Two sorts are equal if they associate the same types to the same port names, and if the cyclic ordering of ports is the same.

1.8 Bunched Networks

The number of ports contained in a sort can quickly get out of hand when arrays of networks are built. In these cases it is too cumbersome to invent different names for all the ports in a sort, but ambiguities would arise if we allowed repeated names. We therefore introduce **bunches** as a way of structuring port names.

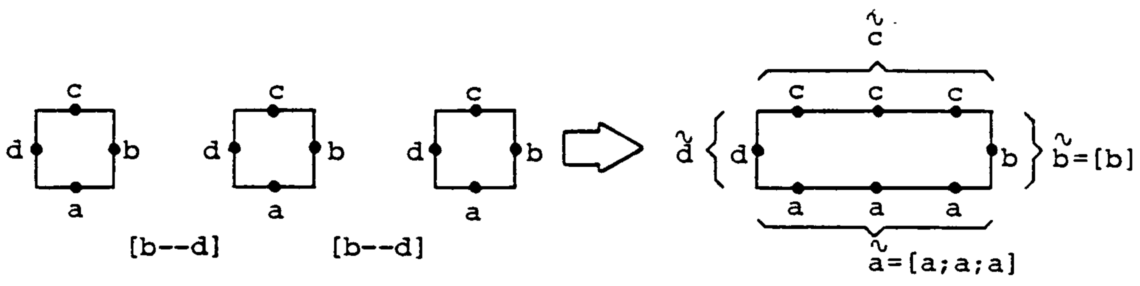


Figure 1.10 Bunches arising in composition

In a **bunched sort**, the port names are partitioned into a collection of lists, called **bunches**. Each bunch $\tilde{a}$ is a list containing several copies of the same name, a (each copy denoting a different port):

$\tilde{a}$ = [a;..;a]

All the names in a bunch must have the same type. Empty bunches $\tilde{b}$=[] are also admitted, meaning that there is no b port.

We can consider a bunched sort as an ordinary sort containing indexed names a_i (where a_i is the i-th item in the list $\tilde{a}$=[a;..;a]), the advantage of the list notation is that we obtain an automatic re-indexing on bunch operations. Lists are used instead of multisets because ports must not lose their individuality.

A bunch restriction p\a cancels the bunch $\tilde{a}$ from the sort of p.

A bunch renaming p{r} (e.g. p{a\b}) renames uniformly all the elements of the bunches specified by r (i.e. $\tilde{a}$ becomes $\tilde{b}$). Note that r must still be a bijection of port names.

A **debunching** operation gives access to the individual elements of a bunch: p{a[$\tilde{i}$]\b} renames part of the bunch a to a new bunch b, provided that b is not already in the sort of p. The list [$\tilde{n}$] is a list of indexes of $\tilde{a}$; it can be written as a list of numbers [1;2;5] or a range [3..7] or a combination of them [12;5..7;2;1]. Note that debunching can be used to reorder a bunch: for example if p has a bunch of four ports $\tilde{b}$, than p{b[4;3;2;1]\b} inverts the order of the ports in the bunch.

A **cobunching** operation is used to merge bunches: p{a;b\c} renames the concatenation of the bunches $\tilde{a}$ and $\tilde{b}$ (in that order) to a bunch $\tilde{c}$, provided that c is either a, or b, or is not already in the sort of p.

Debunching and cobunching can be generalised to more complex expressions like

p{a[3..5];b;c[1]\b, d\e} =

    p{a[3..5]\b'}{c[1]\b"}{b';b\b}{b;b"\b}{d\e}

provided that restrictions similar to the ones discussed above are observed.

The implicit bunch composition p|q connects the bunches of p to the bunches of q having the same name, and the connected bunches disappear from the sort of the result. The usual restrictions apply to bunch composition. Moreover the connection of two bunches is legal only if the bunches have the same number of elements; then the first element of one bunch is connected to the last element of the other bunch, and similarly for all the other elements (this convention turns out to be natural on several occasions, e.g. to connect a bunch on the "west" of a net to a bunch on the "east" of another net, and expecially in the case of planar bunches).

A more general kind of composition is the **partial implicit** composition $p|_A q$, where A is a subset of the common bunches of p and q. Only the bunches contained in A are connected as described above; the remaining bunches common to p and q are cobunched in pairs of the same name (the ones of p to the left). For example if we imagine to have nets of rectangular shape, we can connect the east bunches of one net to the west bunches of another net, while the south and north bunches of both copies are bunched together.

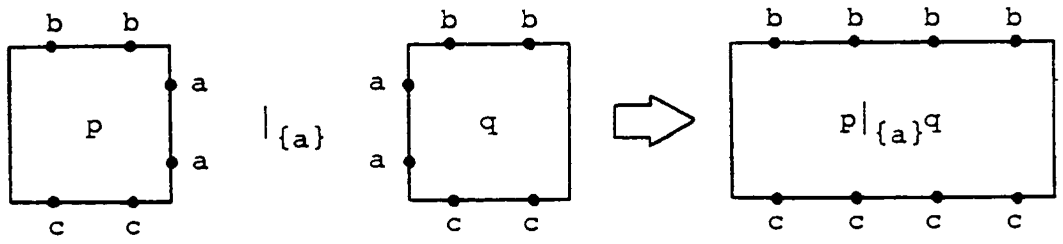


Figure 1.11 Partial implicit composition

The explicit bunch composition p[r]q connects the bunches of p to the bunches of q according to [r] as with partial implicit composition: we can define [r] as

$$p[r]q \triangleq p|_{\lceil r \rceil}(q\{r^{-1}\})$$

Hence the connected bunches disappear from the sort of the result, and if p has a bunch $\tilde{a}_p$ and q has a bunch $\tilde{a}_q$, then the cobunching

$\tilde{a}_p;\tilde{a}_q$ belongs to the result. This automatic cobunching turns out to be very useful.

Formally, the set PortNames' of a bunched sort is the set PortNames $\times$ $\mathbb{N}$, where n $\in$ $\mathbb{N}$ is the length of each bunch. Let #: A $\longrightarrow$ $\mathbb{N}$ be the function returning the length of each bunch of a sort based on A, and returning 0 for each port name not contained in A. Moreover, let $\langle n\rangle \triangleq \{1,..,n\}$. Then bunch restriction is defined as:

$$p\backslash b \triangleq p\backslash\{b_i \mid i\in\langle\#b\rangle\}$$

and bunch renaming as:

$$p\{r:A\simeq B\} \triangleq p\{\cup_{a\in A}\{a_i\backslash r(a)_i \mid i\in\langle\#a\rangle\}\}$$

For debunching we need to introduce the operation $\partial(i,l)$ which returns the position of the number i in the list of numbers l.

$$p\{a[l]\backslash b\} \triangleq p\ \{a_i\backslash b_{\partial(i,l)} \mid i\in l\} \cup \{a_i\backslash a_{\partial(i,\langle\#a\rangle\backslash l)} \mid i\notin l\}$$

Cobunching is defined as:

$$p\{a;b\backslash c\} \triangleq p\ \{a_i\backslash c_i \mid i\in\langle\#a\rangle\} \cup \{b_i\backslash c_{\#a+i} \mid i\in\langle\#b\rangle\}$$

Finally, partial implicit composition (from which the other compositions can be derived) is defined in terms of the previously described bunch operators and of normal composition:

$$\begin{aligned} p\,|_A q \ &\triangleq \\ &(p\{b\backslash b' \mid b\in B\}\cup\{a_{i_a}\backslash a_{\#a-i_a+1} \mid a\in A, i_a\in\langle\#a\rangle\}\ | \\ &\ q\{b\backslash b'' \mid b\in B\}) \\ &\{b';b''\backslash b \mid b\in B\}\end{aligned}$$

where $B = (\lceil\sigma(p)\rceil\cap\lceil\sigma(q)\rceil)\backslash A$ and b',b'' do not occur in the sorts of p and q.

1.9 Planar Bunched Networks

A Planar bunched sort is a planar sort with planar bunches; a planar bunch is a bunch $\tilde{a}=[a_1;..;a_n]$ where the a_i respect the cyclic order of their sort. Planar bunch operations are similar to their nonplanar versions, except that they must make sense in a planar framework.

Planar bunch restriction and renaming present no further problems.

Debunching is valid only if the extracted sub-bunch respects the cyclic ordering; note that we can rotate bunches this way, like in $b[2;3;4;1]\backslash\tilde{b}$ for a 4-bunch $\tilde{b}$.

Cobunching needs some further explanation; the planar cobunching of two planar bunches $\tilde{a};\tilde{b}\backslash\tilde{c}$ is the bunch $\tilde{c}$ starting with the first port of $\tilde{a}$, containing $\tilde{a}$ and $\tilde{b}$, and respecting the order of the planar sort. Note that if $\tilde{a}$ and $\tilde{b}$ are interleaved then $\tilde{c}$ respects the interleaving, and $\tilde{c}$ can be rotated in order to start with the desired port of $\tilde{a}$ or $\tilde{b}$.

The various kinds of compositions work much as before. Again the connection of (interleaved) bunches must respect the cyclic ordering and the first port of each bunch connects to the last port of the respective matching bunch. Note that this first-to-last conventions allows us, in most cases, to connect planar bunches without having to rearrange them in order to respect planarity constraints.

1.10 Molecules, Hypercubes, Mosaics and Klein Bottles

This section shows some examples of use of net algebras, especially concerning recursive definitions and bunches. The examples suggest some interesting extensions of our notation which are left as open problems.

The first example is a attempt to describe molecules by their chemical bond structure. Chemical elements of valency n are represented by literals with n ports, for example:

H:{h} (hydrogen)

O:$\{o_1,o_2\}$ (oxygen)

C:{c_1,c_2,c_3,c_4} (carbon)

We can easily compose simple molecules:

Methane $\triangleq$

C [c_1—h] H

[c_2—h] H

[c_3—h] H

[c_4—h] H

CarbonDioxide $\triangleq$

C [c_1—o_1, c_2—o_2] O

[c_3—o_1, c_4—o_2] O

CH $\triangleq$

C [c_4—h] H

C_2H_2 $\triangleq$

CH [c_2—c_1, c_3—c_3] CH

Benzene $\triangleq$

C_2H_2 [c_2—c_1] C_2H_2

[c_2—c_1, c_1—c_2] C_2H_2

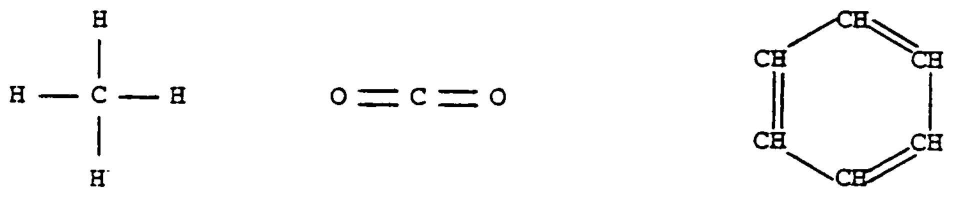


Figure 1.12 Methane, carbon dioxide and benzene

Two molecules are isomers if they have the same number of atoms of each kind, but "behave" differently. Hence isomerism implies structural difference, which can be expressed in our notation as well as by chemical diagrams. For various reasons, it does not matter which valencies of an atom are connected to another atom, so that the simple interchanges of bonds of a single atom does not produce isomers.

In general we might want to talk about the spacial orientation of valencies, which is important in stereochemistry and cristallography. This suggests a generalisation of planar cycles and sorts to three dimensions, producing what we might call envelopes, i.e. arrangements of ports on the vertices of a polyedrum. Envelopes should characterise legal compositions of polyedra in 3-D space, forbidding copenetration in the same way as cycles forbid over-crossings. We could then equally well describe crystalline structures, mechanical parts or the architecture of buildings in a safe and unambiguous way. This is left as an open problem which might have very interesting applications, but which has little relevance here.

As a second example, let us build an n-dimensional cube starting from a single literal v (vertex) with ports $e^1..e^n$. In order to avoid name clashes we index the port names e^i by lists of binary digits (e.g. $e^3_{[0;1;1;0]}$):

$$
\begin{aligned}
c_0 &\triangleq v\{e^1\backslash e^1_{[]},\ ..\ ,e^n\backslash e^n_{[]}\} \\
c_{i+1} &\triangleq \\
&c_i\{e^{i+1}_{[0..0]}\backslash e^{i+1}_{[0,0..0]},\ ..\ ,e^{i+1}_{[1..1]}\backslash e^{i+1}_{[0,1..1]},\ ..\ , \\
&\quad e^n_{[0..0]}\backslash e^n_{[0,0..0]},\ ..\ ,e^n_{[1..1]}\backslash e^n_{[0,1..1]}\} \\
&[e^{i+1}_{[0,0..0]}\text{—}e^{i+1}_{[1,0..0]},\ ..\ ,e^{i+1}_{[0,1..1]}\text{—}e^{i+1}_{[1,1..1]}] \\
&c_i\{e^{i+1}_{[0..0]}\backslash e^{i+1}_{[1,0..0]},\ ..\ ,e^{i+1}_{[1..1]}\backslash e^{i+1}_{[1,1..1]},\ ..\ , \\
&\quad e^n_{[0..0]}\backslash e^n_{[1,0..0]},\ ..\ ,e^n_{[1..1]}\backslash e^n_{[1,1..1]}\}
\end{aligned}
$$

The first three steps in the construction of a three-dimensional cube are illustrated in the next figure:

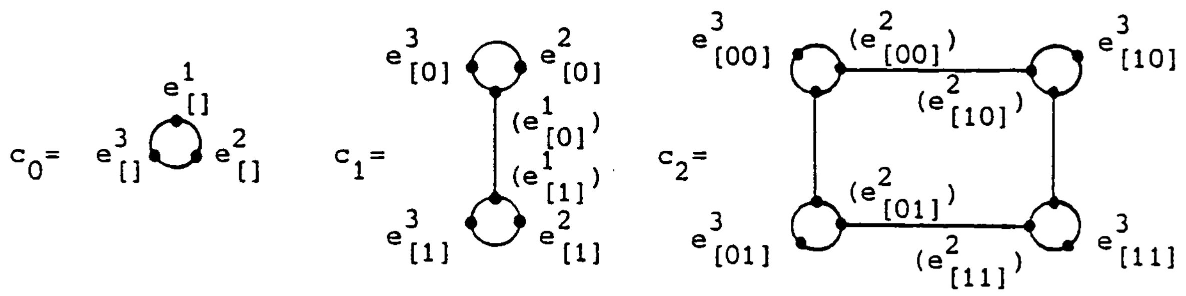


Figure 1.13 Building a cube

This is a situation where the advantages of bunches are particularly clear, indeed by using bunched sorts and compositions we need only write:

$$
\begin{aligned}
c_0 &\triangleq v \\
c_{i+1} &\triangleq c_i[e^{i+1}\text{—}e^{i+1}]c_i
\end{aligned}
$$

Note that the result is really an hypercube, and not a "twisted" version of it (remember that two bunches $a_1..a_n$ and $b_1..b_n$ are connected as $a_1\text{—}b_n..a_n\text{—}b_1$).

Suppose now that we only have a literal v with three ports f,b,e (forward, backward and external) and we still want to make a hypercube [Preparata 79]. All we have to do is to build up n-ary vertices from ternary ones (for n>1):

$$v_n \triangleq (v\{e\backslash e^1\}[f\text{--}b]\dots[f\text{--}b]v\{e\backslash e^{n-1}\})$$
$$[f\text{--}b,b\text{--}f]\ v\{e\backslash e^n\}$$

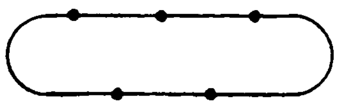


Figure 1.14 An n-ary vertex

and we can then apply our previous definitions.

The third example concerns mosaics on the plane. Suppose we have a literal t (equilateral triangle) with ports b,r,l (base, right and left) organised in this order in a planar sort. The following definition builds a mosaic of triangles which at every steps retains the form of an equilateral triangle. Bunches are used.

$$m_0 \triangleq t$$
$$m_{n+1} \triangleq$$
$$m_n\{b\backslash b', l\backslash l', r\backslash r'\}$$
$$[b'\text{--}b]\ m_n\ [l'\text{--}l]\ m_n\ [r'\text{--}r]\ m_n$$

The next figure shows the first three steps in the construction of a mosaic:

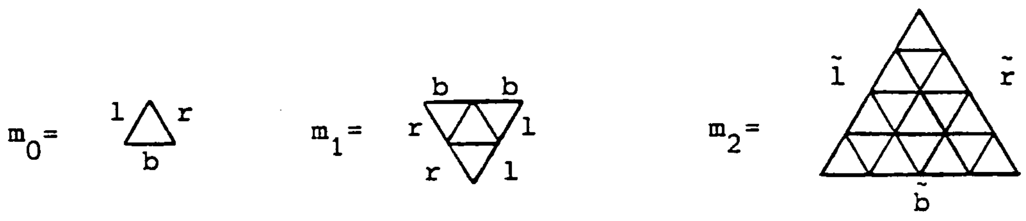


Figure 1.15 Building a mosaic

Note that if we do not use planar sorts triangles are allowed to flip around their connection points, and the result can be a rather complicated three-dimensional graph instead of a planar mosaic.

The fourth example concerns sorts with an infinite number of ports. We can consider a segment s_1 of length 1 in 3-D space, as a literal with uncountably many ports p_x for $0 \le x \le 1$. We can obtain a v-shape by joining two segments at their end point:

$$v \triangleq s_1[p_0 -- p_0]s_2$$

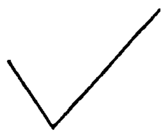


Figure 1.16 Joining two segments

We can then join two v-shapes by connecting the middle points of the first v-shape to the 0.3-points of the second v-shape. Note the effect of bunches in this case.

$$w \triangleq v[p_{0.5} -- p_{0.3}]v$$

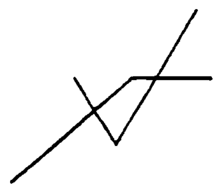


Figure 1.17 Joining two v-shapes

We can produce more interesting examples with a literal r (representing a "flexible" rectangle in 3-D space) with ports n_x, s_x for $0 \leq x \leq 1$ ranging from left to right, and e_y, w_y for $0 \leq y \leq 1$ ranging from bottom to top:

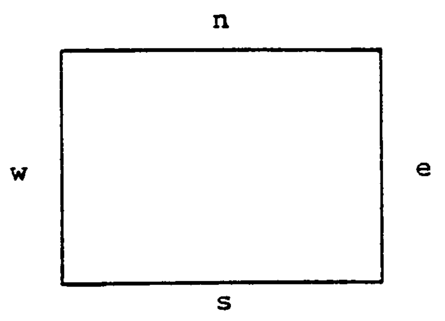


Figure 1.18 A flexible rectangle

Here are some interesting objects which can be obtained:

ring $\triangleq$ $r\ [e_x \text{—} w_x, w_x \text{—} e_x\ \forall x]\ r$

punched-ring $\triangleq$ $r\ [e_x \text{--} w_x, w_x \text{--} e_x\ \forall x \varepsilon [0..0.1] \cup [0.9..1]]\ r$

Note that we do not capture the class of 3-D surfaces modulo continuous transformations; for example we have no way of distinguishing a straight ring from a double-twisted one.

An alternative flexible rectangle may be defined to have four ports n,s,e,w which are uncountable bunches disposed anticlockwise around the perimeter (when bunches are concatenated, they are renormalised to the interval 0..1). This case is particularly similar to the treatment of ports in Chapter 3.

ring	$\triangleq$	r [e--w,w--e] r
punched-ring	$\triangleq$	r [e--w,w[0..0.1;0.9..1]—e[0..0.1;0.9..1]] r
moebius-strip	$\triangleq$	r [e--w,w[0..1]—e[1..0]] r
sphere	$\triangleq$	r [e--w,w--e,n--n,s--s] r
klein-bottle	$\triangleq$	r [e--w,w--e,n--s,s--n] r
torus1	$\triangleq$	ring [n--s,s--n] ring
torus2	$\triangleq$	ring [n[0..1]—s[1..0],s[0..1]—n[1—0] ring

Note that torus1 is obtained by inserting one ring in parallel inside the other, and then joining the edges, while in torus2 the two rings are composed into a thicker ring which is then bent around to connect its two edges.

1.11 Main Example: The Foster & Kung Pattern Matcher

The Foster & Kung pattern matching hardware algorithm [Foster 80] will be used as an example through chapters 1 and 2. It has a very simple and regular structure, deriving from the systematic hierarchical decomposition of a pattern matching problem, while its behaviour involves flow of data in a double pipeline configuration and is far from obvious.

The problem is to find all the occurrences of a pattern p in a text (string of characters) s, where p may contain the distinguished character '*'. A pattern p matches a subtext s' of s if p and s': (i) have the same length and, (ii) either the corresponding characters are equal or the pattern character is '*'.

The pattern matcher is implemented as a separate processor, communicating with some host computer; here is the general plan.

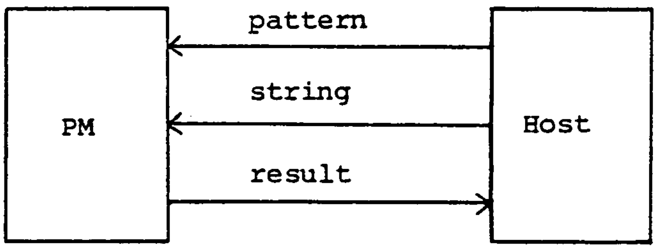


Figure 1.19 The pattern matcher as a processor

Instead of storing the pattern into PM and then supplying the string, it is simpler to implement a on-the-fly pattern matching where the pattern is repeatedly transmitted by the host H together with an indication of the end of the pattern which is encoded in the last character of the pattern.

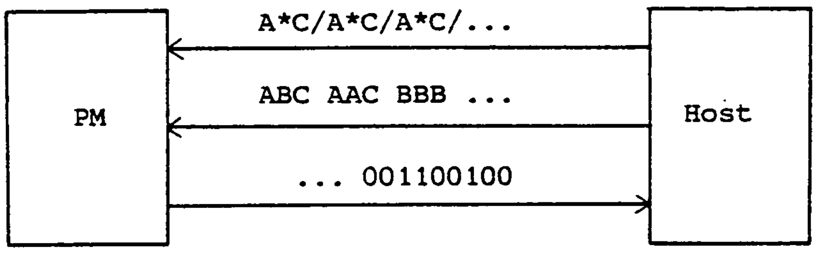


Figure 1.20 The pattern matcher protocol

The result of the matching is returned as a binary string containing a 1 for each successful match; the position of each 1 corresponds to the position of the last character of a matching subtext.

The key architectural idea is the use of a pipeline where text and (repeated) pattern meet head-on.

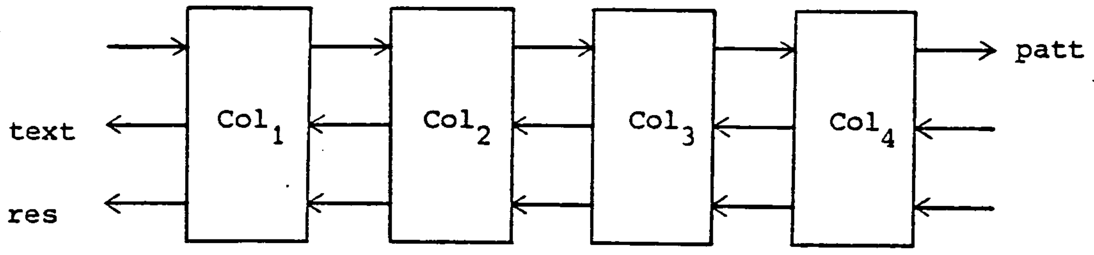


Figure 1.21 Architecture

The pipeline should be at least as long as the pattern. This structure is very convenient because makes the matching process time-linear in the text length and space-linear in the pattern length. Moreover, if we want to match a very long pattern we can simply connect several PM processors in a row and all works well.

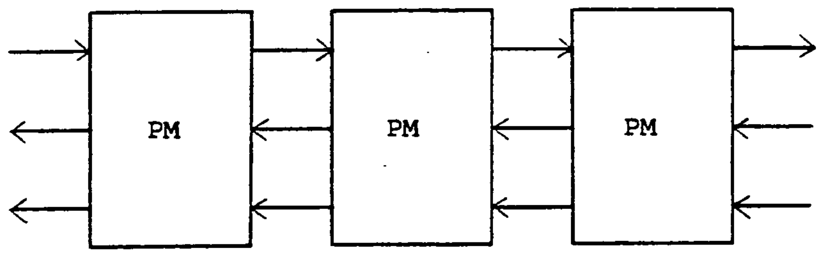


Figure 1.22 Matching long patterns

Every stage of the pipeline matches a single character of text to a single character of pattern. The stage produces an output whenever it matches the last char of the pattern, otherwise it transmits forward the output coming from the previous stage.

Consider a single stage: it receives in turn the pattern from the left and the text from the right (retransmitting them unaltered) and it has to remember whether all the previous characters matched, so that at the end of the pattern it can tell whether the pattern as a whole matches the subtext.

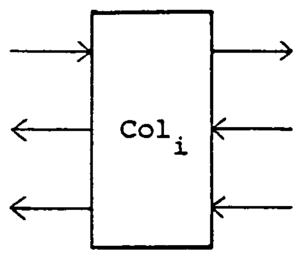


Figure 1.23 A stage of the pattern matcher

If there are n_p characters in the pattern, a single stage will consider all the substrings of length n_p starting at multiples of n_p in the text, ignoring all the other substrings. The other substrings of length n_p will be considered by the adjacent stages, so that if we have n_p stages we consider all the substrings of length n_p. More than n_p stages will do no harm: the result will simply be overwritten one or more times, but it will still be correct.

We can further decompose the structure of a single stage by distinguishing a comparator part and an accumulator part.

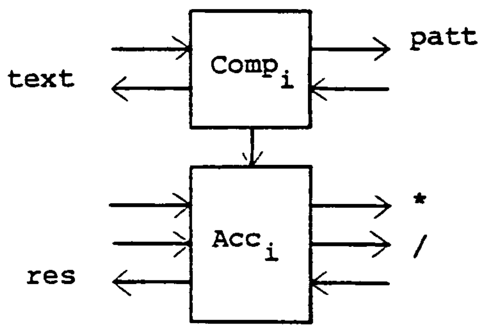


Figure 1.24 Inner structure of a stage

The comparator takes a string character and a pattern character, and compares them outputting the result to the accumulator. The accumulator accumulates the successive results of the comparator, and when the pattern is complete it produces the final result.

The pattern information is split between comparator and accumulator. The comparator receives the proper pattern characters, and the accumulator receives: (i) the information that the current pattern character is actually the wild card character (so that it can ignore the result of the comparator) and: (ii) the information that the current character is the last of the pattern (so that it can output the result and reinitialise itself).

Each character is assumed to be a parallel vector of bits, let us say 4 bits. We can further decompose the comparator into a series of bit comparators, each of them matching a bit of pattern against a bit of string. Having done this, we might just take the boolean and of the results of all the bit comparator and feed it to the accumulator. However, there is a different solution which gives us the opportunity of studying a more interesting kind of architecture, as well as being more elegant for VLSI implementations. We can organise the bit comparators into a pipeline which runs orthogonally to the main string-pattern pipeline; this assumes that the bits constituting the characters are shifted at the input of the pattern matcher and realigned at the output. The net effect is that although a byte comparison takes 4 cycles, the accumulator receives a result at each cycle.

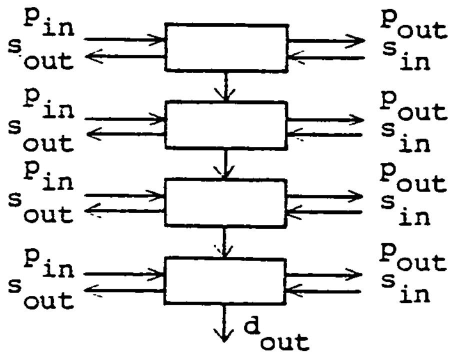


Figure 1.25 Bit comparators

The first bit comparator at the top is connected to "true", and each bit comparator outputs the boolean and of its comparation with the previous result coming from above.

There is a final optimisation to be made. It is convenient to implement each bit comparator by a single inverting stage; this implies that all the outputs will be inverted, and the next comparator (both below and to the left) must be ready to accept an inverted output. This leads to differentiating "positive" and "negative" comparators and accumulators, arranging them into a chess-board pattern. The behaviour of the pattern matcher will not be affected, provided that there are both an even number of stages and an even number of bits in each character.

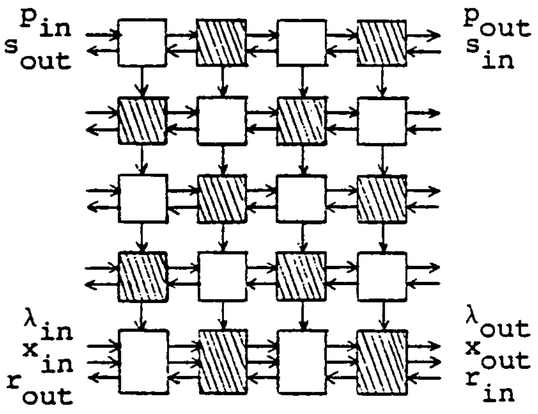


Figure 1.26 Positive and negative devices

We now show that the structure of the pattern matcher can be expressed as a network. We take the bit comparators and accumulators as black boxes (to be denoted by literals) and we compose them together into the complete system using our network operations. In the next chapter, more refined net algebras will be used to specify the contents of these black boxes according to the descriptive model or technology we want to implement them in. Here we use the following primitives (i.e. literals):

True: {true: match}

False: {false: match}

PosBitComp: {p_{in},p_{out}: pattern, s_{in},s_{out}: string, d_{in},d_{out}: match}

NegBitComp: {p_{in},p_{out}: pattern, s_{in},s_{out}: string, d_{in},d_{out}: match}

PosAccum: {λ_{in},λ_{out}: endpattern, x_{in},x_{out}: wildcard,
r_{in},r_{out}: result, d_{in}: match}

NegAccum: {λ_{in},λ_{out}: endpattern, x_{in},x_{out}: wildcard,
r_{in},r_{out}: result, d_{in}: match}

(The choice of types (pattern,string,match, etc.) is a pure matter of taste: we might have defined all the ports to have type bool, or we might have introduced enough type structure to syntactically forbid the direct composition of BitComp's of the same sign.)

In order to parameterise the pattern matcher with respect to its dimensions we introduce a simple iteration construct:

n times p with [r]

which uses n-1 times the connection [r] to connect n copies of p, for example:

3 times p with [r] = p[r]p[r]p

We can now program the pattern matcher, using bunches, iteration and parameterisation.

```
PosByteComp n =
  n times PosBitComp [d_out—d_in] NegBitComp
  with [d_out—d_in]

NegByteComp n =
  n times NegBitComp [d_out--d_in] PosBitComp
  with [d_out—d_in]
```

```
PosColumn n =
  True [true--$d_{in}$]
  PosByteComp n [$d_{out}$--$d_{in}$]
  PosAccum

NegColumn n =
  False [false--$d_{in}$]
  NegByteComp n [$d_{out}$--$d_{in}$]
  NegAccum

PatternMatcher m n =
  m times
    PosColumn n
      [$p_{out}$--$p_{in}$, $s_{in}$--$s_{out}$,
       $\lambda_{out}$--$\lambda_{in}$, $x_{out}$--$x_{in}$, $r_{in}$--$r_{out}$]
    NegColumn n
  with [$p_{out}$--$p_{in}$, $s_{in}$--$s_{out}$,
        $\lambda_{out}$--$\lambda_{in}$, $x_{out}$--$x_{in}$, $r_{in}$--$r_{out}$]
```

What we are doing here from an algebraic point of view is to introduce a set of derived operators; for example for every n and r we have a unary operator n times p with [r]; again for every n we have a nullary derived operator PosByteComp n, etc. Similar kind of programming will be done in Chapter 2. All these ideas will finally be incorporated into a real programming language in Chapter 3.

1.12 Appendix: Some Proofs Needed for the Initiality Theorem

Theorem The operations are well defined:

(1) $\forall l \in \mathbb{L}.\ \langle \{1\},\ 1 \mapsto l,\ A,\ a \in A \mapsto \langle 1,a \rangle,\ \emptyset \rangle$, where $A = \lceil \lambda(l) \rceil$, is a network;

(2) if N is a network, so is $N \backslash a$;

(3) if N is a network and r a bijection on $\lceil s \rceil$, then $N\{r\}$ is a network;

(4) if N, N' are networks and s, s' are compatible, then $N | N'$ is a network;

(5) the operations have the correct type.

Proof

(1) $l = \langle \{1\},\ 1 \mapsto l,\ A,\ a \in A \mapsto \langle 1,a \rangle,\ \emptyset \rangle$ is clearly a network.

(2) $N = \langle V,\gamma,A,\pi,E \rangle$ and $N \backslash a = \langle V',\gamma',A',\pi',E' \rangle$:

we have $V' = V$; $\gamma' = \gamma$; $A' = A \backslash \{a\}$;

$P' = \{\langle v',b \rangle \mid v' \in V' \text{ and } b \in \lceil \lambda(\gamma'(b)) \rceil\} = \{\langle v,b \rangle \mid v \in V \text{ and } b \in \lceil \lambda(\gamma(b)) \rceil\} = P$; $\pi' : A' \rightarrow P' = \pi \backslash a$ is 1-1; type$'$ = type;

$E' = E \subseteq P \times P = P' \times P'$; E' satisfies 1. and 2. because E does, and it satisfies 3. as $\pi'(A') \cap \lceil E' \rceil = \pi \backslash a (A \backslash a) \cap \lceil E \rceil = \pi(A) \cap \lceil E \rceil = \emptyset$.

(3) $N = \langle V,\gamma,A,\pi,E \rangle$ and $N\{r\} = \langle V',\gamma',A',\pi',E' \rangle$:

we have $V' = V$; $\gamma' = \gamma$; $A' = r(A)$; type$'$ = type;

$P' = \{\langle v',b \rangle \mid v' \in V' \text{ and } b \in \lceil \lambda(\gamma'(b)) \rceil\}$

$= \{\langle v,b \rangle \mid v \in V \text{ and } b \in \lceil \lambda(\gamma(b)) \rceil\} = P$;

$\pi' : A' \rightarrow P' = \pi \circ r^{-1}$ is 1-1 as π is 1-1 and r is a bijection;

$E' = E \subseteq P \times P = P' \times P'$; E' satisfies 1. and 2. because E does, and it satisfies 3. as $\pi'(A') \cap \lceil E' \rceil = \pi(r^{-1}(r(A))) \cap \lceil E \rceil = \pi(A) \cap \lceil E \rceil = \emptyset$.

(4) $N = \langle V,\gamma,A,\pi,E \rangle$, $N' = \langle V',\gamma',A',\pi',E' \rangle$ and $N | N' = \langle V'',\gamma'',A'',\pi'',E'' \rangle$:

we have $V'' = V \uplus V'$; $\gamma'' = \gamma \uplus \gamma'$; $A'' = A \oplus A'$;

$P'' = \{\langle v'',b \rangle \mid v'' \in V'' \text{ and } b \in \lceil \lambda(\gamma''(b)) \rceil\} = P \uplus P'$;

$\pi'':A''\longrightarrow P'' = \pi{\downarrow}(A\backslash C) \cup \pi'{\downarrow}(A'\backslash C)$ with $C=A\cap A'$;

$type''(v,a) = \lambda(\gamma''(v))(a) =$ if $(v,a)\varepsilon P$ then $type(v,a)$ else $type'(v,a)$;

$E'' = E\cup E'\cup\{\langle\pi a,\pi' a\rangle\langle\pi' a,\pi a\rangle \mid a\varepsilon C\} \subseteq P''\times P''$; E'' is symmetric because so are E and E'; E'' is a partial function because E,E' are partial functions, π,π' are 1-1 and condition 3. holds for N and N'. E'' satisfies 2. because E' and E'' do, $V\cap V'=\emptyset$ and s,s' are compatible; E'' satisfies 3. because:

$\pi''(A'')\cap\lceil E''\rceil = ((\pi{\downarrow}(A\backslash C) \cup \pi'{\downarrow}(A'\backslash C))(A\oplus A')\cap\lceil E''\rceil$

$= ((\pi{\downarrow}(A\backslash C))(A\backslash C) \cup (\pi'{\downarrow}(A'\backslash C))(A'\backslash C)) \cap \lceil E''\rceil$

$= (\pi(A\backslash C)\cap\lceil E\rceil) \cup (\pi(A'\backslash C)\cap\lceil E'\rceil) = \emptyset$.

(5) The sort of 1 is $s = \lambda(1)$;

the sort of $N' = N\backslash a$ is $s' = type'\circ\pi' = type\circ\pi\backslash a = s\backslash a$;

the sort of $N' = N\{r\}$ is $s' = type'\circ\pi' = type\circ\pi\circ r^{-1} = s\circ r^{-1}$;

the sort of $N'' = N|N'$ is:

$s'' = type''\circ\pi'' = (type\cup type')\circ(\pi{\downarrow}(A\backslash C)\cup\pi'{\downarrow}(A'\backslash C))$

$= (type\circ\pi{\downarrow}(A\backslash C)) \cup (type'\circ\pi'{\downarrow}(A'\backslash C)) = s{\downarrow}(A\backslash A') \cup s'{\downarrow}(A'\backslash A)$.

□

Theorem (Definability)

Every network can be denoted by a well sorted net expression (up to network isomorphism).

Proof The proof is by induction on the size of V; since $V\neq\emptyset$, we consider the cases where V is a singleton and where it has at least two elements.

(1) $V = \{v\}$, $N = \langle V,\gamma,A,\pi,E\rangle$:

then $\gamma = v\mapsto 1$; $P = \{\langle v,a\rangle \mid a\varepsilon\lceil\lambda(1)\rceil\}$; $E = \emptyset$ as there is only one vertex and self-loops are not allowed;

define $f:A\longrightarrow B$ where $B=\lceil\lambda(1)\rceil$ as $f(b) = f = {\downarrow}_2\circ\pi$;

define $r:f(A)\longrightarrow A$ as $r(a) = f^{-1}(a)$.

N is defined by $1\backslash(B\backslash f(A))\{r\}$, in fact:

$1\backslash(B\backslash f(A))\{r\} = \langle\{1\}, 1\mapsto 1, \ulcorner\lambda(1)\urcorner, a\mapsto\langle 1,a\rangle, \emptyset\rangle \backslash(B\backslash f(A))\ \{r\}$

$= \langle\{1\}, 1\mapsto 1, B, a\varepsilon B\mapsto\langle 1,a\rangle, \emptyset\rangle \backslash(B\backslash f(A))\ \{r\}$

$= \langle\{1\}, 1\mapsto 1, f(A), a\varepsilon B\mapsto\langle 1,a\rangle, \emptyset\rangle\ \{r\}$

$= \langle\{1\}, 1\mapsto 1, f(A), a\varepsilon f(A)\mapsto\langle 1,a\rangle, \emptyset\rangle\ \{r\}$

$= \langle\{1\}, 1\mapsto 1, r(f(A)), (a\varepsilon f(A)\mapsto\langle 1,a\rangle)\circ r^{-1}, \emptyset\rangle$

$= \langle\{1\}, 1\mapsto 1, A, \pi', \emptyset\rangle \simeq N$, where $\pi'=\{\langle a,\langle 1,b\rangle\rangle | \langle a,\langle v,b\rangle\rangle\varepsilon\pi\}$

(2) $V = V'\cup V''$ with $\#V',\#V''\geq 1$ and $V'\cap V''=\emptyset$; $N = \langle V,\gamma,A,\pi,E\rangle$:

let $\gamma' = \gamma'\downarrow V'$; $P' = \{\langle v,a\rangle \mid v\varepsilon V'$ and $a\varepsilon\ulcorner\lambda(\gamma'(v))\urcorner\}$; $E' = E\cap P'\chi P'$;

$A' = \{a\varepsilon A \mid \pi(a)\varepsilon P'\}$; $\pi_0' = \pi\downarrow A'$;

let $\gamma'' = \gamma\downarrow V''$; $P'' = \{\langle v,a\rangle \mid v\varepsilon V''$ and $a\varepsilon\ulcorner\lambda(\gamma''(v))\urcorner\}$; $E'' = E\cap P''\chi P''$;

$A'' = \{a\varepsilon A \mid \pi(a)\varepsilon P''\}$; $\pi_0'' = \pi\downarrow A''$;

let $E''' = E\backslash(E'\cup E'') \subseteq (P'\chi P'')\cup(P''\chi P')$;

let $\alpha:C\rightarrow(E'''\cap(P'\chi P''))$ be a bijection such that $C\cap(A'\cup A'')=\emptyset$ (α assigns a new name to each connection between V' and V'' in N);

let $N' = \langle V',\gamma',A'\cup C,\pi',E'\rangle$ where $\pi' = \pi_0'\cup(\downarrow_1\circ\alpha)$

and $N'' = \langle V'',\gamma'',A''\cup C,\pi'',E''\rangle$ where $\pi'' = \pi_0''\cup(\downarrow_2\circ\alpha)$

It is easily verified that N' and N'' are networks, moreover:

$\gamma = \gamma'\cup\gamma''$; $P = P'\cup P''$; $P'\cap P'' = \emptyset$; $A = A'\cup A''$; $A'\cap A'' = \emptyset$; $\pi = \pi_0'\cup\pi_0''$;

$E \supseteq E'\cup E''$; $E'\cap E'' = \emptyset$.

By induction hypothesis there are net expressions e' and e'' defining N' and N''. We now show that $e'|e''$ defines N.

First, $N'|N''$ is well defined; in fact $\forall a\varepsilon C.\ s'(a) = type'(\pi'(a)) = type'(\downarrow_1\alpha(a))$ and $s''(a) = type''(\pi''(a)) = type''(\downarrow_2\alpha(a))$, which are the same because of condition 2. on N.

We have to verify that $N'|N'' = \langle V,\gamma,A,\pi,E\rangle$, in fact:

$V'\cup V'' = V$; $\gamma'\cup\gamma'' = \gamma$,

$(A'\cup A''\cup C)\backslash((A'\cup C)\cap(A''\cup C)) = (A'\cup A''\cup C)\backslash C = A'\cup A'' = A$;

$\pi'\downarrow(A'\cup C)\backslash C \cup \pi''\downarrow(A''\cup C)\backslash C = \pi'\downarrow A' \cup \pi''\downarrow A''$

$= \pi_0'\cup(\downarrow_1\circ\alpha)\downarrow A' \cup \pi_0''\cup(\downarrow_2\circ\alpha)\downarrow A'' = \pi_0' \cup \pi_0'' = \pi$;

$E'\cup E''\cup\{\langle\pi'a,\pi''a\rangle,\langle\pi''a,\pi'a\rangle \mid a\varepsilon C\}$

$= E'\cup E''\cup\{\langle\downarrow_1(a),\downarrow_2(a)\rangle,\langle\downarrow_2(a),\downarrow_1(a)\rangle \mid a\varepsilon C\}$

$= E' \cup E'' \cup \{\langle v',v''\rangle,\langle v'',v'\rangle \mid a \varepsilon C \text{ and } \langle v',v''\rangle=\alpha(a)\}$

$= E' \cup E'' \cup E''' = E$

$\square$

Theorem (Consistency)

Laws [|] .. [{}|] are valid up to network isomorphism.

Proof

[\]: $e\backslash a = \langle V,\gamma,A,\pi,E\rangle\backslash a = \langle V,\gamma,A\backslash a,\pi\backslash a,E\rangle = \langle V,\gamma,A,\pi,E\rangle$

as $a \not\varepsilon \lceil\sigma(e)\rceil = A$.

[\\]: $e\backslash a\backslash b = \langle V,\gamma,A,\pi,E\rangle\backslash a\backslash b$

$= \langle V,\gamma,A\backslash a\backslash b,\pi\backslash a\backslash b,E\rangle = \langle V,\gamma,A\backslash b\backslash a,\pi\backslash b\backslash a,E\rangle = e\backslash b\backslash a$

[{}]: $e\{id\} = \langle V,\gamma,A,\pi,E\rangle\{id\}$

$= \langle V,\gamma,id(A),\pi\circ id^{-1},E\rangle = \langle V,\gamma,A,\pi,E\rangle = e.$

[{}{}]: $e\{r\}\{r'\} = \langle V,\gamma,A,\pi,E\rangle\{r\}\{r'\}$

$= \langle V,\gamma,r'(r(A)),\pi\circ r^{-1}\circ r'^{-1},E\rangle$

$= \langle V,\gamma,(r'\circ r)(A)),\pi\circ(r'\circ r)^{-1},E\rangle = e\{r'\circ r\}.$

[{}\]: $e\{r\}\backslash r(a) = \langle V,\gamma,A,\pi,E\rangle\{r\}\backslash r(a)$

$= \langle V,\ \gamma,\ r(A)\backslash r(a),\ (\pi\circ r^{-1})\backslash r(a),\ E\rangle$

$= \langle V,\ \gamma,\ r(A)\backslash r(a),\ (\pi\backslash a)\circ(r^{-1})\backslash r(a),\ E\rangle$

$= \langle V,\ \gamma,\ (r\backslash a)(A\backslash a),\ (\pi\backslash a)\circ(r\backslash a)^{-1},\ E\rangle\ e\backslash a\{r\backslash a\}.$

[|]: $e|e' = \langle V,\gamma,A,\pi,E\rangle|\langle V',\gamma',A',\pi',E'\rangle$

$= \langle V\cup V',\ \gamma\cup\gamma',\ A\oplus A',\ \pi\downarrow(A\backslash C)\ \cup\ \pi'\downarrow(A'\backslash C),\ E\cup E'\cup E''\rangle$

$= \langle V',\gamma',A',\pi',E'\rangle|\langle V,\gamma,A,\pi,E\rangle = e'|e.$

[\|]: $(e|e')\backslash a = (\langle V,\gamma,A,\pi,E\rangle|\langle V',\gamma',A',\pi',E'\rangle)\backslash a$

$= \langle V\cup V',\ \gamma\cup\gamma',\ (A\oplus A')\backslash a,\ (\pi\downarrow(A\backslash C)\ \cup\ \pi'\downarrow(A'\backslash C))\backslash a,\ E\cup E'\cup E''\rangle$

$= \langle V\cup V',\ \gamma\cup\gamma',\ (A\backslash a)\oplus(A'\backslash a),\ \pi\downarrow(A\backslash a\backslash D)\ \cup\ \pi'\downarrow(A'\backslash a\backslash D),\ E\cup E'\cup F''\rangle$

$= \langle V,\gamma,A\backslash a,\pi\backslash a,E\rangle|\langle V',\gamma',A'\backslash a,\pi'\backslash a,E'\rangle$

$= (\langle V,\gamma,A,\pi,E\rangle)\backslash a|(\langle V',\gamma',A',\pi',E'\rangle)\backslash a = (e\backslash a)|(e\backslash a)$

where $C = A\cap A'$

and $D = A\backslash a\cap A'\backslash a = (A\cap A')\backslash a = C\backslash a = C$ because $a\varepsilon\lceil\sigma(e|e')\rceil$.

$(A\oplus A')\backslash a = (A\backslash a)\oplus(A'\backslash a);$

$(\pi\downarrow(A\backslash C)\ \cup\ \pi'\downarrow(A'\backslash C))\backslash a = (\pi\downarrow(A\backslash C))\backslash a\ \cup\ (\pi'\downarrow(A'\backslash C))\backslash a$

$= (\pi\downarrow(A\backslash C)\backslash a)\ \cup\ (\pi'\downarrow(A'\backslash C)\backslash a) = \pi\downarrow((A\backslash a)\backslash D)\ \cup\ \pi'\downarrow((A'\backslash a)\backslash D);$

$E'' = \{\langle\pi a,\pi' a\rangle,\langle\pi' a,\pi a\rangle \mid a\varepsilon D\}$

$= \{\langle\pi a,\pi' a\rangle,\langle\pi' a,\pi a\rangle \mid a\varepsilon C\} = F''.$

$[\{\}|]$: $(e|e')\{r\cup r'\} = (\langle V,\gamma,A,\pi,E\rangle|\langle V',\gamma',A',\pi',E'\rangle)\{r\cup r'\}$

$= \langle V\cup V',\gamma\cup\gamma',A_0,P_0,E_0\rangle$

$= \langle V\cup V',\gamma\cup\gamma',A_1,\pi_1,E_1\rangle$

$= \langle V,\gamma,(r\cup r'')A,\pi\circ(r\cup r'')^{-1},E\rangle|\langle V',\gamma',(r'\cup r'')A',\pi'\circ(r'\cup r'')^{-1},E'\rangle$

$= \langle V,\gamma,A,\pi,E\rangle\{r\cup r''\}|\langle V',\gamma',A',\pi',E'\rangle\{r'\cup r''\}$

$= e\{r\cup r''\}|e'\{r'\cup r''\}$

where:

$C = A\cap A';$

$D = (r\cup r'')A\ \cap\ (r'\cup r'')A' = r''(A\cap A');$

$A_0 = (r\cup r')(A\oplus A') = r(A)\oplus r'(A') = (r\cup r'')A\ \oplus\ (r'\cup r'')A' = A_1;$

$P_0 = (\pi\downarrow(A\backslash C)\ \cup\ \pi'\downarrow(A'\backslash C))\circ(r\cup r')^{-1}$

$= (\pi\downarrow(A\backslash A')\ \cup\ \pi'\downarrow(A'\backslash A))\circ(r\cup r')^{-1}$

$= (\pi\downarrow(A\backslash A')\circ r^{-1})\ \cup\ (\pi'\downarrow(A'\backslash A)\circ r'^{-1})$

$= (\pi\circ r^{-1})\ \cup\ (\pi'\circ r'^{-1})$

$= ((\pi\circ(r\cup r'')^{-1})\downarrow r(A)\ \cup\ (\pi'\circ(r'\cup r'')^{-1})\downarrow r'(A')$

$= ((\pi\circ(r\cup r'')^{-1})\downarrow((r\cup r'')A\backslash(r'\cup r'')A')$

$\cup\ (\pi'\circ(r'\cup r'')^{-1})\downarrow((r'\cup r'')A'\backslash(r\cup r')A)$

$= ((\pi\circ(r\cup r'')^{-1})\downarrow(((r\cup r'')A)\backslash D)$

$\cup\ (\pi'\circ(r'\cup r'')^{-1})\downarrow(((r'\cup r'')A')\backslash D)$

$= P_1;$

$E_0 = E\cup E'\cup\{\langle\pi a,\pi' a\rangle,\langle\pi' a,\pi a\rangle \mid a\varepsilon A\cap A'\}$

$= E\cup E'\cup\{\langle\pi(r''^{-1}a),\pi'(r''^{-1}a)\rangle,\langle\pi'(r''^{-1}a),\pi(r''^{-1}a)\rangle \mid a\varepsilon r''(A\cap A')\}$

$= E\cup E'\cup\{\langle\pi((r\cup r'')^{-1}a),\pi'((r'\cup r'')^{-1}a)\rangle,$

$\langle\pi'((r'\cup r'')^{-1}a),\pi((r\cup r'')^{-1}a)\rangle \mid a\varepsilon r''(A\cap A')\}$

$= E_1.$

$[||]$: $(e|e')|e'' = (\langle V,\gamma,A,\pi,E\rangle|\langle V',\gamma',A',\pi',E'\rangle)|\langle V'',\gamma'',A'',\pi'',E''\rangle$

$= \langle V\cup V',\ \gamma\cup\gamma',\ A\oplus A',\ \pi\pi',\ EE'\rangle\ |\ \langle V'',\gamma'',A'',\pi'',E''\rangle$

$= \langle (V \cup V') \cup V'', (\gamma \cup \gamma') \cup \gamma'', (A \oplus A') \oplus A'', \pi\pi'-\pi'', EE'-E'' \rangle$

$= \langle V \cup (V' \cup V''), \gamma \cup (\gamma' \cup \gamma''), A \oplus (A' \oplus A''), \pi-\pi'\pi'', E-E'E'' \rangle$

$= \langle V,\gamma,A,\pi,E \rangle | \langle V' \cup V'',\gamma' \cup \gamma'',A' \oplus A'',\pi'\pi'',E'E'' \rangle$

$= \langle V,\gamma,A,\pi,E \rangle | (\langle V',\gamma',A',\pi',E' \rangle | \langle V'',\gamma'',A'',\pi'',E'' \rangle)$

$= e|(e'|e'')$

where:

$AA' = A \oplus A'$;

$A'A'' = A' \oplus A''$;

$\pi\pi' = \pi{\downarrow}(A \backslash A') \cup \pi'{\downarrow}(A' \backslash A)$;

$\pi'\pi'' = \pi'{\downarrow}(A' \backslash A'') \cup \pi''{\downarrow}(A'' \backslash A')$;

$\pi\pi'-\pi'' = \pi\pi'{\downarrow}(AA' \backslash A'') \cup \pi''{\downarrow}(A'' \backslash AA')$

$= (\pi{\downarrow}(A \backslash A') \cup \pi'{\downarrow}(A' \backslash A)){\downarrow}(A \oplus A' \backslash A'') \cup \pi''{\downarrow}(A'' \backslash A \oplus A')$

$= \pi{\downarrow}(A \backslash A' \backslash A'') \cup \pi'{\downarrow}(A' \backslash A \backslash A'') \cup \pi''{\downarrow}(A'' \backslash A \oplus A')$

$= \pi{\downarrow}(A \backslash A' \oplus A'') \cup \pi'{\downarrow}(A' \backslash A \backslash A'') \cup \pi''{\downarrow}(A'' \backslash A \backslash A')$ as $A \cap A' \cap A'' = \emptyset$

$= \pi{\downarrow}(A \backslash A'A'') \cup \pi'\pi''{\downarrow}(A'A'' \backslash A)$

$= \pi-\pi'\pi''$;

$EE' = E \cup E' \cup \{\langle \pi a,\pi' a \rangle,\langle \pi' a,\pi a \rangle \mid a \in A \cap A'\}$;

$E'E'' = E' \cup E'' \cup \{\langle \pi' a,\pi'' a \rangle,\langle \pi'' a,\pi' a \rangle \mid a \in A' \cap A''\}$;

$EE'-E'' = EE' \cup E'' \cup \{\langle \pi\pi' a,\pi'' a \rangle,\langle \pi'' a,\pi\pi' a \rangle \mid a \in AA' \cap A''\}$

$= E' \cup E' \cup E'' \cup \{\langle \pi a,\pi' a \rangle,\langle \pi' a,\pi a \rangle \mid a \in A \cap A'\}$

$\cup \{\langle \pi a,\pi'' a \rangle,\langle \pi'' a,\pi a \rangle \mid a \in A \cap A''\} \cup \{\langle \pi' a,\pi'' a \rangle,\langle \pi'' a,\pi' a \rangle \mid a \in A' \cap A''\}$

as $A \cap A' \cap A'' = \emptyset$

$= E \cup E'E'' \cup \{\langle \pi a,\pi'\pi'' a \rangle,\langle \pi'\pi'' a,\pi a \rangle \mid a \in A \cap A'A''\}$

$= E-E'E''$

$\square$

2. Hardware Networks for VLSI

2.1 Introduction

This chapter presents case studies of several classes of networks and their relationships. Each class is meant to represent an aspect of VLSI (Very Large Scale Integration) circuit design. In general we will follow a top-down approach, starting with very abstract networks (describing behaviours) and descending to very concrete ones (describing geometric patterns). However, just to keep our aim in mind, the first part of the chapter is dedicated to the lowest and most concrete level: VLSI layouts.

We shall not and cannot go so far as to show a complete set of algorithmic translations from behavioural descriptions to layouts; this is a very complex problem with a large number of possible options yet to explore. A translation process of this kind inevitably requires a considerable amount of heuristics which could only be tested by large-scale experiments. However we shall show how to break the problem down into finding a set of translations between intermediate levels of description. These levels are chosen to be self-justifying, in the sense that they all occur naturally in VLSI design and could be independently used as a basis for design tools; indeed some of them are already used in this way. Moreover we shall show the intermediate translations which are already known, sketch some which are conceivable and discuss some new ones. All the translations are modelled on the algebraic concept of homomorphism; they preserve the structure of descriptions and are meant to be mainly algorithmic, with few well localised heuristics.

Here is a picture of the general plan, which will become clearer while reading this chapter; blobs are levels of description and arrows are translations:

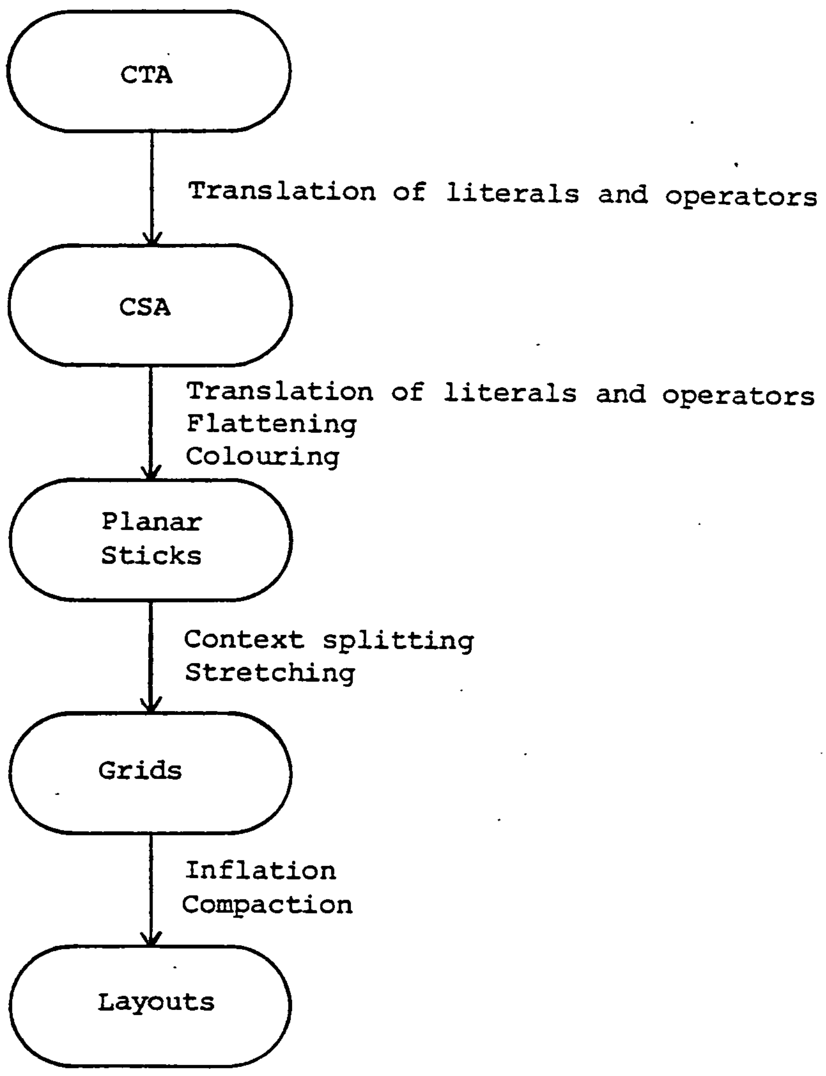


Figure 2.1 Description levels and translations

Briefly, CTA (Clocked Transition Algebra) is a behavioural description level, CSA (Connector Switch Attenuator) are switch-level diagrams, then there are stick diagrams, grids (a metric version of stick diagrams) and finally layouts.

2.2 Layouts

VLSI technology implements a computational model which radically diverges both from theoretical constructs such as sequential machines and their languages, and from practical realities like logic-gate hardware and its methodologies. It is probably no more attractive to know what kinds of transistors are available in some technology than it is to learn the instruction set of some particular machine, but the two kinds of primitives are fundamentally different, and their difference is bound to be reflected in any high level tool or formalism devised to deal with them.

It is therefore necessary to investigate some aspects of VLSI technology, because they effectively create a new computational paradigm [Kung 80, Chazelle 81]. Fortunately, a rather clean interface can be drawn between the design and fabrication activities, thanks largely to the work of Carver Mead and Lynn Conway [Mead 80]. In the case of digital systems this interface can on the one hand permit the designer to ignore most of the fabrication parameters, and on the other hand permits the fabrication process to ignore the meaning of the systems being built. An effort is currently being made on both sides to adapt the design and fabrication activities to this end.

Hence we take the view that the result of any VLSI design activity is a layout, which is our interface to the physical world of VLSI circuits. A layout is a set of geometric figures (generally rectangles) describing the geometrical structure of the devices to be fabricated. The rectangles are distributed over several planes, to indicate the different materials and phases of the fabrication processes. The position, size and overlapping of rectangles determines the electrical characteristics of the devices being fabricated, which are generally digital switches, resistors and conductors. In the most desirable situation, the fabrication process should act as a black box receiving layouts (pure geometric information, free from fabrication details like electrical parameters) and producing working chips at the other end. Here "working" means corresponding to the layout specification, which is purely syntactical and does not involve any knowledge of the intended behaviour of the devices.

Layouts specify the physical dimensions of wires, transistors, contacts etc. These specifications must obey some rules, which fall into two general classes: minimal size of devices and minimal separation between devices. Sizes and distances are expressed in an abstract unit called λ, which can be scaled up or (preferably) down according to the particular fabrication process. Good values for λ in 1981 are around 2-3 microns; by 1990 $\lambda=0.25$ microns (corresponding to 10^9 devices per silicon wafer) might be widespread. At that point nMOS technologies encounter foundamental physical limits.

Geometrical design rules generally say that wires and transistor must be at least 2λ wide, and must be separated by at least 2λ. Similar constraints are imposed on contacts etc. The most standard design rules in university environments are the ones described in [Mead 80]; a recent proposal for making them both more regular and technology independent is discussed in [Sequin 81]. Several example layouts are given in chapter 3.

2.3 Clocked Networks

A clocked system is one where events only happen at discrete time instants. The flow of time is governed by a global clock and events are only observed during clock activity. Clocked systems attempt to make the clock appear instantaneous, so that events are fully determined at the clock instant. In practice the clock is active for a finite and positive interval of time, and during this interval events can very well be unstable.

Constraints must then be imposed on the clocking scheme and on the structure of the systems so that events are stable during the active-clock period. First of all, there should be no asynchronous loop; if this condition is met, then it is possible to slow down the clock rate until all the events have had time to stabilise. Secondly, the system should be in isolation: there is no way to guarantee the correct operation of a rigidly clocked system in the presence of asynchronous input signals.

In spite of these problems, clocked systems are simple both to model and to reason about because of the discrete timing assumption. They are also simpler to implement, and most of the hardware systems today are clocked (but see [Seitz 80, Barton 81] for arguments in favour of self-timed systems). Our first example of a net algebra will allow us the expression of the structure and the behaviour of clocked systems; it is called Clocked Transition Algebra (CTA for short).

2.3.1 Clocked Transition Algebra (CTA) Expressions

A CTA Expression is a net expression over a particular set of literals. These are called clocked transition literals. As a simple example, the clocked transition:

$$a_{in} \rightsquigarrow b_{out}$$

means that the input from port a_{in} is transmitted to port b_{out} after one clock period. The sort of this expression is:

$\{a_{in}: T, b_{out}: T\}$

for any type T. Here we take the brute force approach of considering transitions as literals. Alternatively we might define an algebra of transitions (expressions describing input values) and then combine this algebra with the surrounding net algebra to obtain a bigger derived algebra [Burstall 77].

Successive attempts at an exact interpretation of "$\leadsto$" will be discussed, culminating with a formal semantics. For a first attempt, every arrow "$\leadsto$" is taken to indicate a restoring stage where the input from a is stored at time t to make it available on b at time t+1. Every restoring stage is clocked. The situation can be displayed as follows:

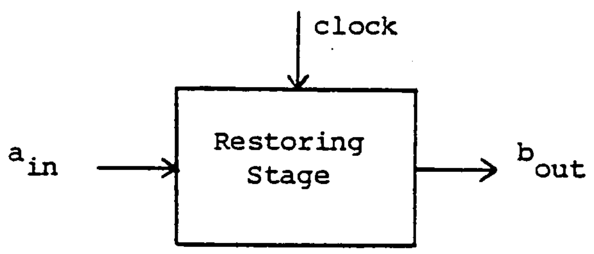


Figure 2.2 A restoring stage

When the clock is active, the input from a is stored inside the box, and the stored value is immediately available on b. This implies that while the clock is active, a and b are physically connected. Now consider the following loop:

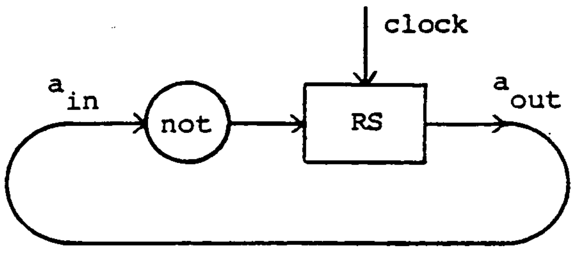


Figure 2.3 An asynchronous loop

Here we have a problem: when the clock is active, $\sim a_{in}$ is available on a_{out}. If the signal can go around the loop before the clock is deactivated, an attempt will be made to store $\sim\sim a_{in}$ (instead of $\sim a_{in}$) in the box. This situation is called an unstable **asynchronous** loop (stable asynchronous loops are also possible). The actual value stored in the restoring stage will depend on the number of loops the signal manages to perform before the clock is shut down, or even worse in concrete situations the value stored will be somewhere between a_{in} and $\sim a_{in}$.

To repair this problem we might try to make our active-clock interval very short, but then it is difficult to reliably store the value of "a_{in}" in such a short time. The situation is so uncomfortable that we switch to a different clocking scheme: instead of a single-phase clock we adopt a **two-phase** non-overlapping clock, ϕ_1 and ϕ_2:

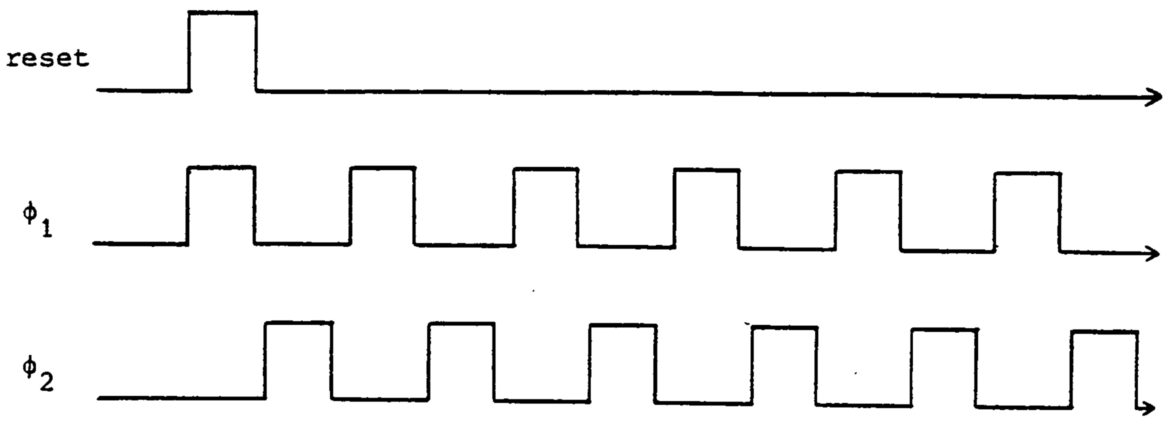


Figure 2.4 Two-phase non-overlapping clock scheme

As shown in figure above, the first ϕ_1 pulse is also carried by a special reset line which is used to obtain well defined starting conditions.

For a second attempt, our loop can be reinterpreted as:

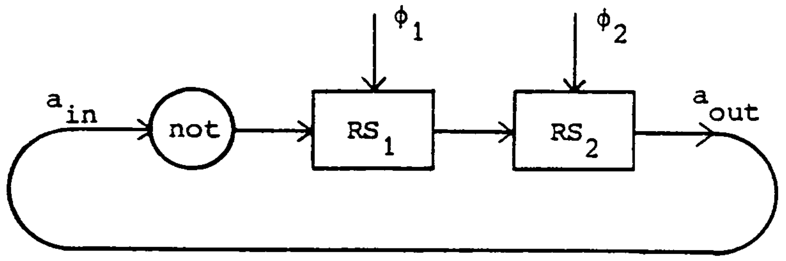


Figure 2.5 A different loop

During phase ϕ_1 $\sim a_{in}$ is stored in the first box, and during ϕ_2 the content of the first box is stored in the second one. Because the two phases are not overlapping, input and output are never in contact and the asynchronous loop is broken.

This scheme is not yet entirely satisfactory. For efficiency we may want to insert some useful circuitry between the two above restoring stages. In fact all is well so long as we correctly alternate the two clock phases.

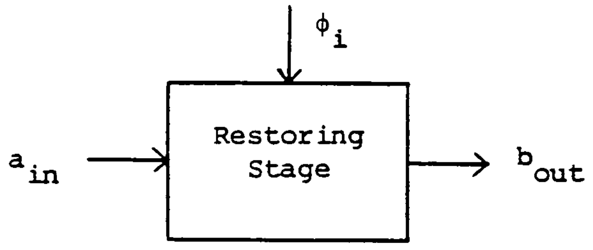


Figure 2.6 Final restoring stage

Our final interpretation for transitions is then that "$\rightsquigarrow$" is thought of as a single restoring block clocked by a single phase. To avoid ambiguities, the arrows are sometimes subscripted by the particular clock phase: "$\rightsquigarrow_1$" for ϕ_1 and "$\rightsquigarrow_2$" for ϕ_2. It is possible to check syntactically that an assignment of phases is correct; phases must alternate along every path in a net, and loops must have an even number of arrows.

Very frequently we need to share an input between several transitions and an implicit forking of the inputs is therefore required (conversely, we insist that any mergeing of the outputs is explicit). A special "," operator is used to indicate sharing, and we write:

$a_{in} \rightsquigarrow_1 b_{out}, \ a_{in} \rightsquigarrow_1 c_{out}$

meaning that the input a_{in} is shared by two transitions.

We can then consider clusters of transitions separated by "," as literals in our Clocked Transition Algebra, and use net operations to combine them. Furthermore we shall only allow transitions of the same clock phase to be clustered together. We have 1-clusters which input during phase 1 and output stably during phase 2, and 2-clusters which input during phase 2 and output stably during phase 1. For 2-clusters we also have to specify what their output will be during the <u>first</u> ϕ_1 clock pulse (i.e. on power-up or reset) because they start producing an output before receiving any input.

Our final notation is as follows. Clusters of phase-1 transitions will be denoted by:

$\langle 1{:}t_1 \rightsquigarrow b_1, \ \ldots \ ,t_n \rightsquigarrow b_n\rangle$

Clusters of phase-2 transitions will be written:

$\langle 2{:}t_1 \rightsquigarrow b_1 v_1, \ \ldots \ ,t_n \rightsquigarrow b_n v_n\rangle$

where $v_1 \ldots v_n$ are non-empty sets of admissible power-up values; in the boolean case they can each be {tt}, {ff} or {tt,ff}. The power-up values may be omitted, and they are then assumed to be "don't care", i.e. {tt,ff} (nondeterministically true or false).

In examples we shall only use boolean transitions. The left hand side of a boolean transition can consist of boolean constants (tt and ff), boolean operations ($\sim$, $\wedge$, $\vee$) and boolean-valued conditionals ("a$\Rightarrow$b;c", i.e. "if a then b else c").

2.3.2 Main Example

We can now give a specification of the black boxes described in the previous chapter, i.e. the positive and negative bit comparators and accumulators. Positive boxes are clocked by ϕ_1 and negative ones by ϕ_2:

True =

$\langle 2\text{: } tt \rightsquigarrow true\{tt\}\rangle$

False =

$\langle 1\text{: } ff \rightsquigarrow false\rangle$

PosBitComp =

$\langle 1\text{:}\sim p_{in} \rightsquigarrow p_{out},$
$\sim s_{in} \rightsquigarrow s_{out},$
$\sim(d_{in} \wedge (p_{in}=s_{in})) \rightsquigarrow d_{out}\rangle$

NegBitComp =

$\langle 2\text{:}\sim p_{in} \rightsquigarrow p_{out},$
$\sim s_{in} \rightsquigarrow s_{out},$
$\sim d_{in} \wedge (p_{in}=s_{in}) \rightsquigarrow d_{out}\rangle$

PosAccum =

$\langle 1\text{:}\sim \lambda_{in} \rightsquigarrow \lambda_{out},$
$\sim x_{in} \rightsquigarrow x_{out},$
$\sim(\lambda_{in} \Rightarrow t_{in}; r_{in}) \rightsquigarrow r_{out},$
$\sim(\lambda_{in} \Rightarrow tt; (t_{in} \wedge (x_{in} \vee d_{in}))) \rightsquigarrow t_{out}\rangle$
$[t_{out} -- t_{in},\ t_{in} -- t_{out}]$
$\langle 2\text{:}\sim t_{in} \rightsquigarrow t_{out}\rangle$

NegAccum =

$\langle 2{:}\sim\lambda_{in} \rightsquigarrow \lambda_{out},$

$\sim x_{in} \rightsquigarrow x_{out},$

$\sim\lambda_{in} \Rightarrow \sim t_{in}; \sim r_{in} \rightsquigarrow r_{out},$

$\sim\lambda_{in} \Rightarrow tt; (t_{in} \wedge (\sim x_{in} \vee \sim d_{in})) \rightsquigarrow t_{out}\rangle$

$[t_{out} \text{--} t_{in},\ t_{in} \text{--} t_{out}]$

$\langle 1{:}\sim t_{in} \rightsquigarrow t_{out}\rangle$

The same program that was shown in the previous chapter can be used to put these pieces together. Note that PosAccum and NegAccum are formed by the composition of two transition of opposite phase feeding each other. This configuration produces the bit of storage needed to remember the result of previous matchings.

2.3.3 Formal semantics of CTA

In this section we give a semantics to CTA expressions via a translation to the Synchronous Calculus of Communicating Systems (SCCS) [Milner 81]. We do not offer an introduction to SCCS but Chapter 5 can provide enough background, especially because, as noted in Section 5.8, SCCS can be considered as the class of 1-synchronous real time agents.

First we introduce some notation in SCCS which allows us to simulate value passing simply by indexing the port names; "$\Sigma_x p[x]$", for $x \in \{tt,ff\}$ is an abbreviation for "$p[tt/x] + p[ff/x]$"; "a?x:p" is an abbreviation for "$\Sigma_x\ a_x{:}p$", (representing the act of inputting the boolean value x from port a); "b!v:p" is an abbreviation for "$b_v{:}p$" (representing the act of outputting the boolean value v from port b). Several input and output ports can be mixed together in the same prefix, moreover the name of the indexes may be the same as the ports they index. For example:

$a?a,\ b?b,\ c!a\vee b:\ p\ =\ \Sigma_{a,b}\ a_a\ b_b\ c_{a\vee b}:\ p$

inputs two values on the ports a,b and outputs their boolean or (without any delay) on port c.

In the definition of clusters there is an implicit forking of input signals. This can be modelled by the following combinator:

$$\mathrm{fork}(a,n) \Leftarrow a?x, (\bar{a}'!x)^n\text{: } 1\text{: } \mathrm{fork}(a,n)$$

Where $a^n \triangleq a,\ldots,a$ (n times).

The semantics is given by a translation function $[\![\cdot]\!]$ from CTA to SCCS. We indicate by $[\![\ldots]\!][a_i]$ a set of transitions "..." where the set $\{a_i\}$ contains all the input ports used in the transitions, and by $[\![\ldots]\!][b_i/a_i]$ the substitution of the input ports b_i for a_i in "...". We omit the obvious definition of the translation for boolean expressions.

$[\![<1:t_1 \rightsquigarrow b_1,\ldots,t_n \rightsquigarrow b_n>]\!][a_1..a_m] =$

$(\mathrm{fork}(a_1,n) \times \ldots \times \mathrm{fork}(a_m,n) \times$

$\mu P.\ [\![t_1 \rightsquigarrow b_1]\!][a_i'/a_i] \times \ldots \times [\![t_n \rightsquigarrow b_n]\!][a_i'/a_i] \times 1{:}1{:}P)$

$\backslash a_1' \ldots \backslash a_m'$

where $a_1'\ldots a_m'$ are not in the sort of the transitions

$[\![<2:t_1 \rightsquigarrow b_1 v_1,\ldots,t_n \rightsquigarrow b_n v_n>]\!][a_1..a_m] =$

$\Sigma_{b_1..b_n \varepsilon v_1 \times..\times v_n}\ b_1!b_1,\ldots,b_n!b_n:$

$[\![<1:t_1 \rightsquigarrow b_1,\ldots,t_n \rightsquigarrow b_n>]\!][a_1..a_m]$

$[\![t \rightsquigarrow b]\!][a_1..a_n] =$

$a_1?x_1,\ldots,a_n?x_n:\ b![\![t]\!][x_1/a_1,\ldots,x_n/a_n]:\ 1$

$[\![p\backslash a]\!] = [\![p]\!]\backslash a$

$[\![p\{r\}]\!] = [\![p]\!]\{r\}$

$[\![p[a_i\text{--}b_i]q]\!]$

$\quad([\![p]\!]\{c_i/a_i\} \times [\![q]\!]\{\bar{c}_i/b_i\})\backslash c_i$

$\quad$where c_i are not in the sort of p and q

as an example, let us compute the semantics of $\langle 1\text{:}a \leadsto b\rangle$, where we use "=" for SCCS's "strong congruence" (the same derivations are valid for the "smooth congruence" of Chapter 5).

$[\![\langle 1\text{:}a \leadsto b\rangle]\!][a]$

$\quad = (\text{fork}(a,1) \times (\mu P.\ [\![a \leadsto b]\!][a'/a] \times 1\text{:}1\text{:}P))\backslash a'$

$\quad = (\text{fork}(a,1) \times (a'?y\text{: } b!y\text{: } 1) \times 1\text{:}1\text{:}\mu P.\ \ldots\ )\backslash a'$

$\quad = (a?x,\bar{a}'!x\text{: } 1\text{: fork}(a,1) \times (a'?y\text{: } b!y\text{: } \mu P.\ \ldots\ )\backslash a'$

$\quad = a?x\text{: } b!x\text{: } (\text{fork}(a,1) \times \mu P.\ \ldots\ )\backslash a'$

$\quad = \mu P.\ a?x\text{: } b!x\text{: } P$

Hence $\langle 1\text{:}a \leadsto b\rangle$ repeatedly accepts an input x during phase 1 and produces the same output x during phase 2. Note that according to this semantics no output is generated from b during phase 1 and no input is accepted by a during phase 2. A more detailed semantics might allow b to output nondeterministically tt or ff during phase 1 and a to input an unused value during phase 2; it seems sound to regard these two semantics as essentially equivalent.

2.3.4 Semantics of the main example

We can now compute the semantics of the pattern matcher:

True =

$\quad \mu P.\ \text{true}!\text{tt: } 1\text{: } P$

False =

$\mu P.\ 1\text{: false!ff: } P$

PosBitComp =

$\mu P.\ p_{in}?p_{in}, s_{in}?s_{in}, d_{in}?d_{in}:$

$\quad p_{out}!\sim p_{in}, s_{out}!\sim s_{in}, d_{out}!\sim(d_{in} \wedge (p_{in}=s_{in})):\ P$

NegBitComp' =

$\Sigma_{p_{out}, s_{out}, d_{out}}\ \text{NegBitComp}(p_{out}, s_{out}, d_{out})$

NegBitComp$(p_{out0}, s_{out0}, d_{out0})$ =

$p_{out}!p_{out0}, s_{out}!s_{out0}, d_{out}!d_{out0}:$

$\mu P.\ p_{in}?p_{in}, s_{in}?s_{in}, d_{in}?d_{in}:$

$\quad p_{out}!\sim p_{in}, s_{out}!\sim s_{in}, d_{out}!\sim d_{in} \wedge (p_{in}=s_{in}):\ P$

$= p_{out}!p_{out0}, s_{out}!s_{out0}, d_{out}!d_{out0}:$

$\quad p_{in}?p_{in}, s_{in}?s_{in}, d_{in}?d_{in}:$

$\quad \text{NegBitComp}(\sim p_{in}, \sim s_{in}, \sim d_{in} \wedge (p_{in}=s_{in}))$

PosAccum' =

$\Sigma_t\ \text{PosAccum}(t)$

PosAccum(t_{out0}) =

$((\mu P.\ \lambda_{in}?\lambda_{in}, x_{in}?x_{in}, r_{in}?r_{in}, d_{in}?d_{in}, t_{in}?t_{in}:$

$\quad \lambda_{out}!\sim\lambda_{in}, x_{out}!\sim x_{in}, r_{out}!\sim(\lambda_{in} \Rightarrow t_{in}; r_{in}),$

$\quad t_{out}!\sim(\lambda_{in} \Rightarrow tt; (t_{in} \wedge (x_{in} \vee d_{in}))):\ P)$

$\quad \{\bar{u}/t_{out}, v/t_{in}\}\ \chi$

$(t_{out}(t_{out0}):\ \mu Q.\ t_{in}?t_{in}:\ t_{out}!\sim t_{in}:\ Q)$

$\quad \{u/t_{in}, \bar{v}/t_{out}\})\backslash u \backslash v$

NegAccum' =

$$\Sigma_{\lambda_{out},x_{out},r_{out},t_{out}}\ \mathrm{NegAccum}(\lambda_{out},x_{out},r_{out},t_{out})$$

$$\begin{aligned}
&\mathrm{NegAccum}(\lambda_{out0},x_{out0},r_{out0},t_{out0}) = \\
&\quad(\lambda_{out}!\lambda_{out0},x_{out}!x_{out0},r_{out}!r_{out0},t_{out}!t_{out0}: \\
&\quad\ \mu P.\ \lambda_{in}?\lambda_{in},x_{in}?x_{in},r_{in}?r_{in},d_{in}?d_{in},t_{in}?t_{in}: \\
&\qquad\quad \lambda_{out}!{\sim}\lambda_{in},x_{out}!{\sim}x_{in},r_{out}!{\sim}\lambda_{in}{\Rightarrow}{\sim}t_{in};{\sim}r_{in}, \\
&\qquad\quad t_{out}!{\sim}\lambda_{in}{\Rightarrow}tt;t_{in}\wedge({\sim}x_{in}\vee{\sim}d_{in}):\ P) \\
&\qquad [\bar{u}/t_{out},v/t_{in}]\ \times \\
&\quad(\mu Q.\ t_{in}?t_{in}:\ t_{out}!{\sim}t_{in}:\ Q) \\
&\qquad [u/t_{in},\bar{v}/t_{out}])\backslash u\backslash v
\end{aligned}$$

Simplifying the expressions for PosAccum and NegAccum according to the SCCS laws and recursion theorem we get:

$$\begin{aligned}
&\mathrm{PosAccum}(t) = \\
&\qquad \lambda_{in}?\lambda_{in},x_{in}?x_{in},r_{in}?r_{in},d_{in}?d_{in}: \\
&\qquad \lambda_{out}!{\sim}\lambda_{in},x_{out}!{\sim}x_{in},r_{out}!{\sim}(\lambda_{in}{\Rightarrow}t;r_{in}): \\
&\qquad \mathrm{PosAccum}(\lambda_{in}{\Rightarrow}tt;(t\wedge(x_{in}\vee d_{in})))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{NegAccum}(\lambda_{out0},x_{out0},r_{out0},t) = \\
&\quad \lambda_{out}!\lambda_{out0},x_{out}!x_{out0},r_{out}!r_{out0}: \\
&\quad \lambda_{in}?\lambda_{in},x_{in}?x_{in},r_{in}?r_{in},d_{in}?d_{in}: \\
&\qquad \mathrm{NegAccum}({\sim}\lambda_{in},{\sim}x_{in},{\sim}\lambda_{in}{\Rightarrow}t;{\sim}r_{in},{\sim}\lambda_{in}{\Rightarrow}tt;{\sim}t\wedge({\sim}x_{in}\vee{\sim}d_{in})
\end{aligned}$$

At this point we might try to proceed to compose the various subcomponents in order to get the semantics of the whole circuit. This is in principle no different from the manipulations with have done so far, but the practical difficulties are overwhelming. A ah! ah! }

proof of correctness would require induction on the size nχm of the pattern matcher. This seems to be very lengthy to do by hand, and without machine assistance the confidence in the accuracy of the result would be very low. We try to show the practical difficulties which arise, by composing a little 2χ2 pattern matcher. We start building up a positive column:

$$\text{let } \{\text{Pren}\} = \{p^1_{in}/p_{in}, p^1_{out}/p_{out}, s^1_{in}/s_{in}, s^1_{out}/s_{out}, \bar{d}/d_{out}\}$$
$$\text{and } \{\text{Nren}\} = \{p^2_{in}/p_{in}, p^2_{out}/p_{out}, s^2_{in}/s_{in}, s^2_{out}/s_{out}, d/d_{in}\}$$

$$\text{PN} = (\text{PosBitComp}\{\text{Pren}\} \ \chi \ \text{NegBitComp}'\{\text{Nren}\})\backslash d$$
$$= \Sigma_{p^2_{out}, s^2_{out}, d_{out}} \ \text{PN}(p^2_{out}, s^2_{out}, d_{out})$$

$$\text{PN}(p^2_{out0}, s^2_{out0}, d_{out0}) =$$
$$(\text{PosBitComp}\{\text{Pren}\} \ \chi \ \text{NegBitComp}(p^2_{out0}, s^2_{out0}, d_{out0})\{\text{Nren}\})\backslash d$$
$$= p^1_{in}?p^1_{in}, s^1_{in}?s^1_{in}, d_{in}?d_{in}, p^2_{out}!p^2_{out0}, s^2_{out}!s^2_{out0}, d_{out}!d_{out0}:$$
$$p^2_{in}?p^2_{in}, s^2_{in}?s^2_{in}, p^1_{out}!\sim p^1_{in}, s^1_{out}!\sim s^1_{in}:$$
$$\text{PN}(\sim p^2_{in}, \sim s^2_{in}, d_{in} \wedge (p^1_{in}=s^1_{in}) \wedge (p^2_{in}=s^2_{in}))$$

$$\text{PCol} =$$
$$(\text{True}\{\bar{t}/\text{true}\} \ \chi$$
$$\text{PN}\{t/d_{in}, \bar{d}/d_{out}\} \ \chi$$
$$\text{PosAccum}\{d/d_{in}\})\backslash t\backslash d$$
$$= \Sigma_{p^2_{out}, s^2_{out}, d_{out}} \ \text{PCol}(p^2_{out}, s^2_{out}, d_{out})$$

$$\text{PCol}(p^2_{out0}, s^2_{out0}, d_{out0}, t) =$$
$$(\text{True}\{\bar{t}/\text{true}\} \ \chi$$
$$\text{PN}(p^2_{out0}, s^2_{out0}, d_{out0})\{t/d_{in}, \bar{d}/d_{out}\} \ \chi$$
$$\text{PosAccum}(t)\{d/d_{in}\})\backslash t\backslash d$$

$= p^1_{in}?p^1_{in}, s^1_{in}?s^1_{in}, p^2_{out}!p^2_{out0}, s^2_{out}!s^2_{out0},$

$\lambda_{in}?\lambda_{in}, x_{in}?x_{in}, r_{in}?r_{in}:$

$p^2_{in}?p^2_{in}, s^2_{in}?s^2_{in}, p^1_{out}!\sim p^1_{in}, s^1_{out}!\sim s^1_{in},$

$\lambda_{out}!\sim\lambda_{in}, x_{out}!\sim x_{in}, r_{out}!\lambda_{in}\Rightarrow\sim t;\sim r_{in}:$

$PCol(\sim p^2_{in}, \sim s^2_{in}, (p^1_{in}=s^1_{in})\wedge(p^2_{in}=s^2_{in}),$

$\lambda_{in}\Rightarrow tt;t\wedge(x_{in}\vee d_{out0}))$

We now build a negative column:

let $\{Nren\} = \{p^1_{in}/p_{in}, p^1_{out}/p_{out}, s^1_{in}/s_{in}, s^1_{out}/s_{out}, \bar{d}/d_{out}\}$

and $\{Pren\} = \{p^2_{in}/p_{in}, p^2_{out}/p_{out}, s^2_{in}/s_{in}, s^2_{out}/s_{out}, d/d_{in}\}$

$NP = (NegBitComp\{Nren\}\ \chi\ PosBitComp\{Pren\})\backslash d$

$= \Sigma_{p^1_{out}, s^1_{out}, d_{out}}\ NP(p^1_{out}, s^1_{out}, d_{out})$

$NP(p^1_{out0}, s^1_{out0}, d_{out0}) =$

$(NegBitComp(p^1_{out0}, s^1_{out0}, d_{out0})\{Nren\}\ \chi\ PosBitComp\{Pren\})\backslash d$

$= p^2_{in}?p^2_{in}, s^2_{in}?s^2_{in}, p^1_{out}!p^1_{out0}, s^1_{out}!s^1_{out0}:$

$p^1_{in}?p^1_{in}, s^1_{in}?s^1_{in}, d_{in}?d_{in},$

$p^2_{out}!\sim p^2_{in}, s^2_{out}!\sim s^2_{in}, d_{out}!\sim(d_{out0}\wedge(p^2_{in}=s^2_{in})):$

$NP(\sim p^1_{in}, \sim s^1_{in}, \sim d_{in}\wedge(p^1_{in}=s^1_{in}))$

$NCol =$

$(False\{\bar{f}/false\}\ \chi$

$NP\{f/d_{in}, \bar{d}/d_{out}\}\ \chi$

$NegAccum\{d/d_{in}\})\backslash f\backslash d$

$= \Sigma_{p^1_{out}, s^1_{out}, d_{out}, \lambda_{out}, x_{out}, r_{out}, t_{out}}$

$NCol(p^1_{out}, s^1_{out}, d_{out}, \lambda_{out}, x_{out}, r_{out}, t_{out})$

$$
\begin{aligned}
&\mathrm{NCol}(p^1_{out0}, s^1_{out0}, d_{out0}, \lambda_{out0}, x_{out0}, r_{out0}, t) = \\
&\quad (\mathrm{False}\{\bar{f}/\mathrm{false}\}\ \chi \\
&\quad\ \mathrm{NP}(p^1_{out0}, s^1_{out0}, d_{out0})\{f/d_{in}, \bar{d}/d_{out}\}\ \chi \\
&\quad\ \mathrm{NegAccum}(\lambda_{out0}, x_{out0}, r_{out0}, t)\{d/d_{in}\})\backslash f\backslash d
\end{aligned}
$$

$$
\begin{aligned}
&= p^2_{in}?p^2_{in}, s^2_{in}?s^2_{in}, p^1_{out}!p^1_{out0}, s^1_{out}!s^1_{out0}, \\
&\qquad \lambda_{out}!\lambda_{out0}, x_{out}!x_{out0}, r_{out}!r_{out0}: \\
&\quad p^1_{in}?p^1_{in}, s^1_{in}?s^1_{in}, p^2_{out}!\sim p^2_{in}, s^2_{out}!\sim s^2_{in}, \\
&\qquad \lambda_{in}?\lambda_{in}, x_{in}?x_{in}, r_{in}?r_{in}: \\
&\quad \mathrm{NCol}(\sim p^1_{in}, \sim s^1_{in}, (p^1_{in}=s^1_{in}), \\
&\qquad \sim\lambda_{in}, \sim x_{in}, \sim\lambda_{in}\Rightarrow t;\sim r_{in}, \\
&\qquad \sim\lambda_{in}\Rightarrow tt;\sim t\wedge(\sim x_{in}\vee(d_{out0}\wedge(p^2_{in}=s^2_{in}))))
\end{aligned}
$$

Finally, we compose the two columns into a pattern matcher:

$$
\begin{aligned}
&\mathrm{PM}(p^2_{out0}, s^2_{out0}, d^P_{out0}, t^P, \\
&\qquad p^1_{out0}, s^1_{out0}, d^N_{out0}, \lambda_{out0}, x_{out0}, r_{out0}, t^N) = \\
&p^1_{in}?p^1_{in}, s^2_{in}?s^2_{in}, \lambda_{in}?\lambda_{in}, x_{in}?x_{in}, \\
&\qquad s^2_{out}!s^2_{out0}, p^1_{out}!p^1_{out0}, \lambda_{out}!\lambda_{out0}, x_{out}!x_{out0}: \\
&p^2_{in}?p^2_{in}, s^1_{in}?s^2_{in}, r^1_{in}?r^2_{in}, \\
&\qquad p^2_{out}!\sim p^2_{out0}, s^1_{out}!\sim s^1_{out0}, r_{out}!\lambda_{in}\Rightarrow\sim t^P;\sim r_{out0}: \\
&\mathrm{PM}(\sim p^2_{in}, \sim s^2_{in}, (p^1_{in}=s^1_{out0})\wedge(p^2_{in}=\sim s^2_{in}), \\
&\qquad \lambda_{in}\Rightarrow tt;t^P\wedge(x_{in}\vee d^P_{out0}), p^1_{in}, \sim s^1_{in}, \\
&\qquad (\sim p^1_{in}=s^1_{in}), \lambda_{in}, x_{in}, \lambda_{in}\Rightarrow t^N;\sim r_{in}, \\
&\qquad \lambda_{in}\Rightarrow tt;\sim t^N\wedge(x_{in}\vee(d^N_{out0}\wedge(p^2_{out0}=s^2_{in}))))
\end{aligned}
$$

2.4 Connector-Switch-Attenuator (CSA) Networks

Classical switching theory turns out to be inadequate for describing MOS circuits because, as we shall see later, the underlying boolean logic model fails to account for MOS behaviour at the transistor level. Hence we turn to a more sophisticated model, known as the CSA (Connector-Switch-Attenuator) model [Hayes 81]. CSA gives a static informal semantics to MOS networks, and can be the basis for more formal and dynamic characterisations of MOS behaviour.

The problem with the semantics of integrated circuits is that an exact semantics, modelling what physically happens, seems to be bound to be intractable for large circuits (not only for proofs but even for descriptions); an extreme example of such a semantics is the physics of semiconductor devices. On the other hand a tractable semantics seems to be bound to be inexact, because of the highly complex phenomena occurring in semiconductors which are often exploited by electronic circuits.

Even if it is possible to make simplifying assumptions on the behaviour of devices (and maybe require that these assumptions be met by the fabrication processes) some very basic characteristics of semiconductor devices are intrinsically complex, and critically influence the behaviour of the simplest components.

The two troublesome features, which confer great expressive power to these devices, are the bidirectionality of wires and various forms of capacitive effects. Because of bidirectionality it is difficult to model components by input-output functions of some sort (one possibility is to split every wire into two monodirectional wires and use, for example, the semantic model of Chapter 4, but this seems to lead to intractable formal systems). A more serious difficulty is the modelling of capacitive effects, which in MOS confer the ability to store information. Many circuits critically use the relative sizes of capacitances and their decay times, so that it is difficult to abstract from these electrical parameters and it is not possible to use boolean algebra to model them.

An attempt to give static semantics to MOS circuits is reported in [Hayes 81], where an elegant set of primitives is identified. A very interesting dynamic semantics for MOS circuits is due to Bryant [Bryant 81], deriving from a switch level simulation algorithm. The value domain which we are going to investigate in this section diverges slightly from [Hayes 81] and is a special case of the one proposed in [Bryant 81]. The original points in this section concern the fact that we have a language for expressing CSA networks, and that we regard CSA networks and expressions as an intermediate step in the translation from behaviours to stick diagrams. Moreover, a formal static semantics is defined for CSA circuits, and at the end of the section we give an example of a translation from CTA expressions into CSA.

2.4.1 The Value Domain

Electronic circuits are based on the movement of electrons in conducting materials. Electrons move because of the electrostatic force between them, i.e. because of the presence of an electric field. The electrostatic force is conservative, and it is therefore possible to mesure the work done in moving an electron from one point to another, regardless of the path between the two points. The amount of work needed to move a charge between two points, divided by the value of the charge, is called the potential difference between the points. Only the difference is significant, and the potential value can be set to zero at an arbitrary point.

In digital circuits, at an appropriate level of abstraction, only two values of the potential are relevant: the "zero" level, and another level called "one". The potential difference between zero and one corresponds to the potential difference of the power supply relative to ground. Normally in a digital circuit no potential value can exist below zero or above one, and the intermediate values only exist for comparatively short times, during transitions from zero to one and vice versa.

Boolean operations can be implemented in hardware on this simple two-level domain, and the behaviour of circuits can be understood using boolean algebras plus some notion of time. It all works very well and these principles (plus some "tricks" here and there) are used in all the digital systems built out of discrete hardware, e.g. using TTL or CMOS gate-level chips [Melen 80].

Unfortunately, to fully exploit the possibilities of MOS devices, this simple model is not sufficient. Physicists and electronic engineers, who are familiar with charge distributions and differential equations, work their way through VLSI technologies on the ground of fairly detailed and precise models. Other kinds of people (potential VLSI users) can choose between that and some rough analogies, like for example the water-pipe model [Mead 80]. Here we use CSA models, which seem to capture most of the characteristics of MOS devices at the proper level of abstraction. These models try to reproduce a situation similar to the use of boolean algebras in modelling gate-level hardware. Because we need to operate below the gate level, the model is more refined and is based on a value domain containing seven logical values.

The major step consists in realising that two voltage levels and two current levels (at least) should be quantified in MOS circuits.

The voltage levels are T (for true, or 1) and F (for false, or 0). The current levels are strong (connected to a strong source of charges, like the power supply) and weak (weakened by some obstacle, like a resistor, or coming from a weak source of charges, like a capacitor). Combining them we get four values which can be physically present on a line: strong one (1), strong zero (0), weak one ($\tilde{1}$) and weak zero ($\tilde{0}$). Moreover T is used to denote 1 or $\tilde{1}$ and F to denote 0 or $\tilde{0}$, when we do not care to distinguish between them.

The distinction between weak and strong values is important because a weak value can coexist with a strong value of the opposite sign on the same line; this situation is well defined and the strong value overrides the weak one. The situation is not well defined when two strong values of opposite sign coexist; to model this situation we invent a new state, U, called undefined. Similarly the coexistence of the two opposite weak values gives rise to a weakly undefined situation $\tilde{U}$ which can be overridden by a strong value. The undefined states do not actually exist physically, and they only reflect our inability to describe what exactly happens (or our discomfort about the fact that something undesirable happens). The correct interpretation of U is "we do not know whether it is 1 or 0" rather than "it is both 1 and 0" or "it is 0.5"; similarly for $\tilde{U}$.

Finally, another state is needed to model the case of a point p which is not connected to any source of charges. Such a point cannot be said to have value T or F, because T and F are two definite values of potential, while the potential of p is simply arbitrary. Moreover, if we connect p to a source of charges, it will immediately assume the potential of that source (assuming that p is small enough) whatever that is. A point like p is said to be floating, and the symbol for this state is Z.

These seven logical values can be arranged into a lattice V, to show how they override each other when they coexist; the higher values override the lower ones.

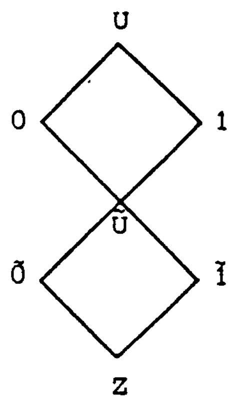


Figure 2.7 The partial order of CSA values

A basic operation on this lattice is the connection $v' \diamond v''$ of two logical values $v', v'' \varepsilon V$, defined as their least upper bound in V.

The following laws hold for $\diamond$:

Associativity:	$v \diamond (v' \diamond v'') = (v \diamond v') \diamond v''$
Commutativity:	$v \diamond v' = v' \diamond v$
Absorption:	$v \diamond v = v$
Zero:	$v \diamond Z = v$
One:	$v \diamond U = U$

Apart from this basic connection operation, different VLSI technologies can be characterised by different primitive functions over V, generally reflecting the different kinds of switches present in a particular technology.

In [Hayes 81] the two undefined states U and $\tilde{U}$ are identified. This reduces the number of primitive values to six, but leads to problems because the set of values is not a lattice and the analogue of ◇ is not associative. The latter fact seems particularly counter-intuitive.

2.4.2 Connectors, Switches and Attenuators

A CSA algebra is a network algebra over a set of CSA literals. There are four classes of CSA primitives: **Sources**, **Connectors**, **Switches** and **Attenuators**; we shall only consider a set of primitives tailored to nMOS circuits.

Sources: a source is a one-terminal device which is either a source of 1 (also called power, or VDD) or a source of 0 (also called ground, or GND).

Connectors: a connector is a multi-way device which performs the connection operation of all its terminals and produces the result on all the terminals. A connector can have various shapes, but we assume that all these shapes can be collapsed to a single point; we do not consider delays or resistances inside connectors.

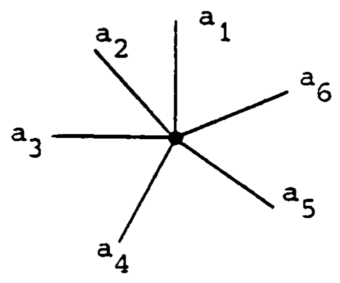


Figure 2.8 A connector

Attenuators: an attenuator is a device which transforms strong values into weak values. A 1 on one side of an attenuator becomes a $\tilde{1}$ on the other side, and similarly for 0. The attenuator is symmetric, and it is perfectly possible to connect one of its terminals to 1 and the other one to 0; in this case the $\tilde{0}$ on one side will be overridden by 1 and the $\tilde{1}$ on the other side will be overridden by 0. This configuration is crucial for the implementation of logical gates.

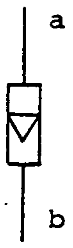


Figure 2.9 An attenuator

The dual of the attenuator is the amplifier, a device which transforms weak values into strong ones; amplifiers can be built out of attenuators, amplifying switches (which we have in nMOS) and power supplies, and are not taken as primitives.

Switches: There can be a great variety of switches and new switches can be introduced as required by some particular technology; these devices have three terminals called gate, source and drain (source and drain can be swapped without affecting the behaviour).

The only switch available in nMOS is the switching-on-T switch. More precisely, when the gate is T (i.e. 1 or $\tilde{1}$) the junction source-drain behaves like a connector; when the gate is F source and drain are not connected; when the gate is Z or $\tilde{U}$ then if source=drain they remain unchanged, else source and drain are U; when the gate is U then source and drain are U. The behaviour of the switch with gate = Z is defined so that undefined states are generated only when strictly necessary. This helps avoiding situations in which undefined values propagate explosively all over the circuit.

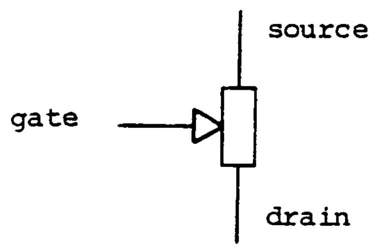


Figure 2.10 A switch

Switches have the important ability of working as dynamic storage devices. An nMOS switch is basically a capacitor which influences with its charge the flow of current in the gate-drain connection. Hence a charged switch with an isolated gate will remember its state until it discharges; if the switch is not **refreshed** it will gradually lose its charge (whence the name of dynamic storage device). The decay time is usually much larger than the clock period, and static storage devices can be obtained by connecting pairs of switches in such a way that they periodically refresh each other.

2.4.3 Basic CSA Circuits

CSA circuits can be expressed in net algebra notation by taking sources, connectors, switches and attenuators as literals. However in this section we shall just describe basic CSA circuits by pictures; examples of expressions of the same order of complexity will be shown in section 2.5 for stick diagrams with planarity constraints.

The most important nMOS structure is the **inverter**. An nMOS inverter acts as a not-gate but, what is more interesting, it can be used as a dynamic storage device. An inverter can be built by a switch (called the **pulldown** in this configuration) and an attenuator (called the **pullup**) connected to power and ground.

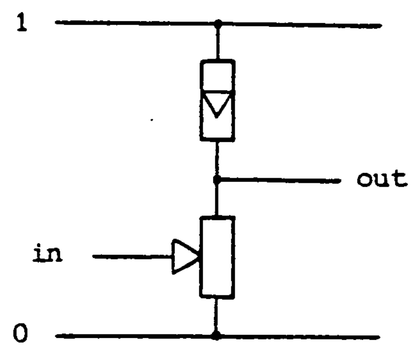


Figure 2.11 A CSA inverter

The attenuator constantly supplies a weak one to the output. When the value T is on the input, the switch connects a strong zero to the output which overrides the weak one, and the result is F. When F is on the input, the switch is open and only the weak one is connected to the output; hence the result is T.

Inverters are often connected into shift register structures:

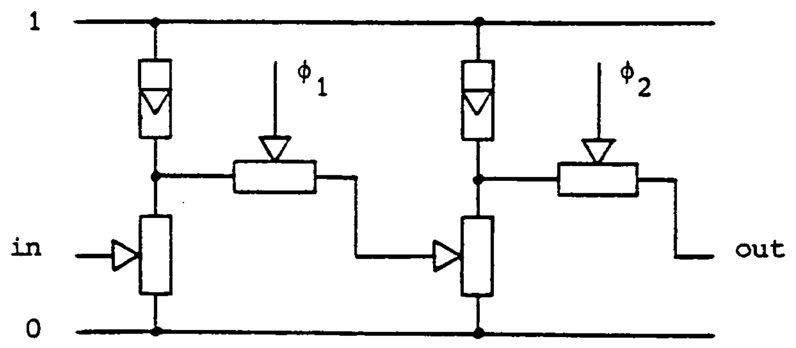


Figure 2.12 Two shift register cells

When a two-phase non-overlapping clock is used, a signal can ripple through a chain of shift register cells, getting inverted at each stage. The switches controlled by the clock signals are called in this configuration **pass transistors**. Note how each pass transistor isolates (when open) the gate of the following switch, so that the charge stored in the following stage is trapped into a dynamic storage configuration.

More complex logic circuits can be built in essentially two ways;

by making more complicated pulldown structures, or by making more complicated pass transistor structures. Very common examples of the first class are the nand and nor configurations:

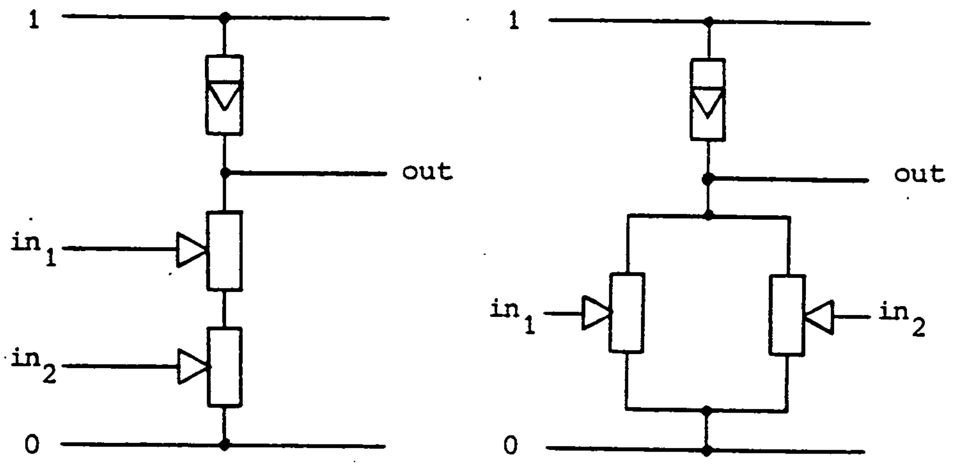


Figure 2.13 CSA Nand and Nor

while selectors and multiplexors can be built very cheaply as pass transistor structures:

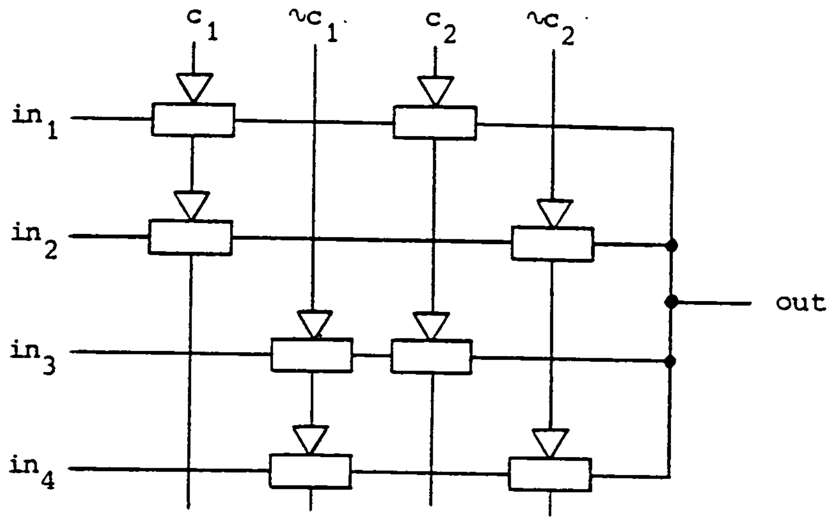


Figure 2.14 A selector

As a last example we show an amplifier (also called non-inverting superbuffer) which converts weak values to strong ones.

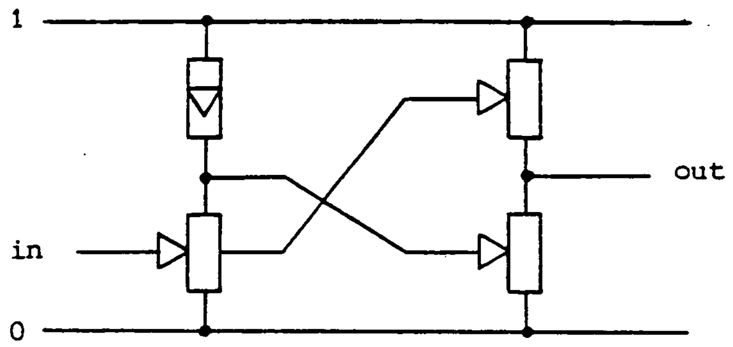


Figure 2.15 An amplifier

Note that this amplifier can only work because nMOS switches have the ability of switching on weak values, so that they are themselves proto-amplifiers.

2.4.4 Static Semantics of CSA

Because of the above mentioned difficulties in giving a formal dynamic semantics for CSA, we define here a simpler static semantic, which only works for circuits which can reach stability. We think that this kind of semantics helps one to understand the general behaviour of CSA circuits, and may be a good starting point for a dynamic semantics.

A **CSA expression** is a net expression with several kinds of literals. There are sources 1: {tt} and 0: {ff}, connectors C_n: $\{c_1,...,c_n\}$, a switch S: {g,s,d}, an attenuator A: {s,d}, and the usual operators e\a, e{r}, e[r]e'.

A static semantics can be given to CSA expressions by considering the set of all the **stable** configurations of a CSA circuit. Intuitively, a stable configuration is an assignment of values to each point of a circuit which does not change when considering the propagation of values through the circuit.

The values present on the terminals of a CSA component or circuit depend in general on the context in which we put the circuit. For example the source 1:{tt} has a stable configuration which assigns the value 1 to the port tt (we write $\langle tt \mapsto 1\rangle$ for this configuration), but in a context in which 1 is connected to 0 the port tt will (stably) present the value U. There is no other stable situation, so that the static semantics of 1 is $\{\langle tt \mapsto U\rangle, \langle tt \mapsto 1\rangle\}$ which is the set of all the stable value assignments to the ports of 1 in every possible context.

Formally, given the CSA value domain V and a finite set of port labels A, a **configuration** (value assignment) on A is an association $c \in V^A$ of values to port labels, written $\langle a_1 \mapsto v_1, \ldots, a_n \mapsto v_n\rangle$ when $A = \{a_1,\ldots,a_n\}$. A **configuration set** on A is a set $C_A \subseteq V^A$; sets C_A will be used to give semantics to CSA circuits of sort A.

There is a natural partial ordering of configurations which is the one induced by the ordering on V, namely for $c,c' \in V^A$:

$$c \leq c' \Leftrightarrow \forall a \in A.\ c(a) \leq c'(a)$$

This partial order is not used in the formal development, but it is convenient when drawing configuration sets, to give them some structure.

Here are the configuration sets for sources:

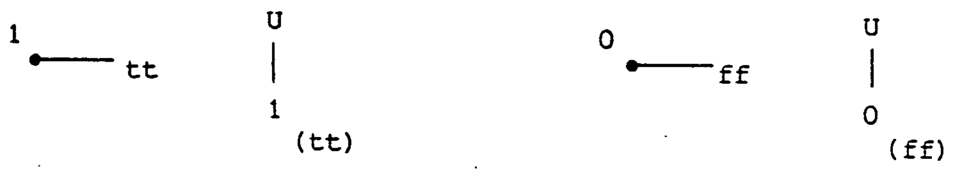


Figure 2.16 Sources

The stable configurations of a connector C_n are those in which all the ports have the same value:

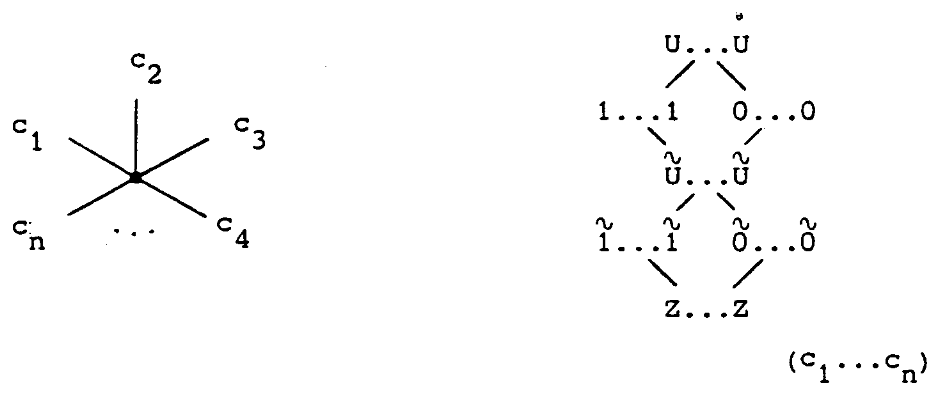


Figure 2.17 Connectors

For attenuators, we can think of trying all the possible pairs of values for source and drain, and see which can be maintained. This gives 17 stable configurations:

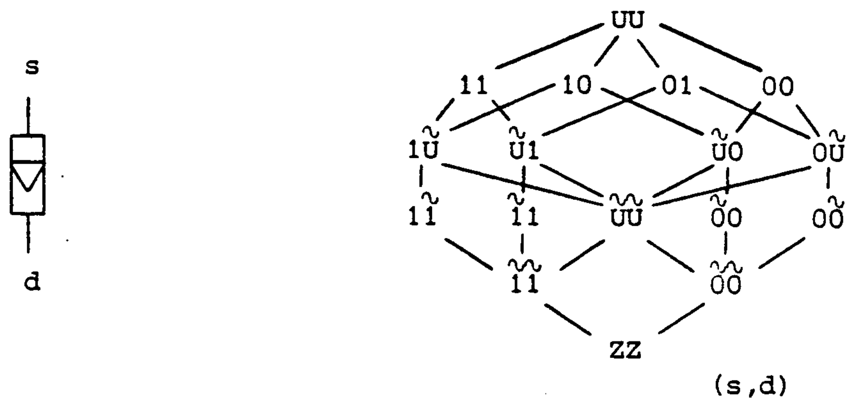


Figure 2.18 Attenuators

Switches have a large number of stable configurations, mostly because source and drain are independent when the gate is F. The configuration set also depends very much on the technology and we just give one possible candidate:

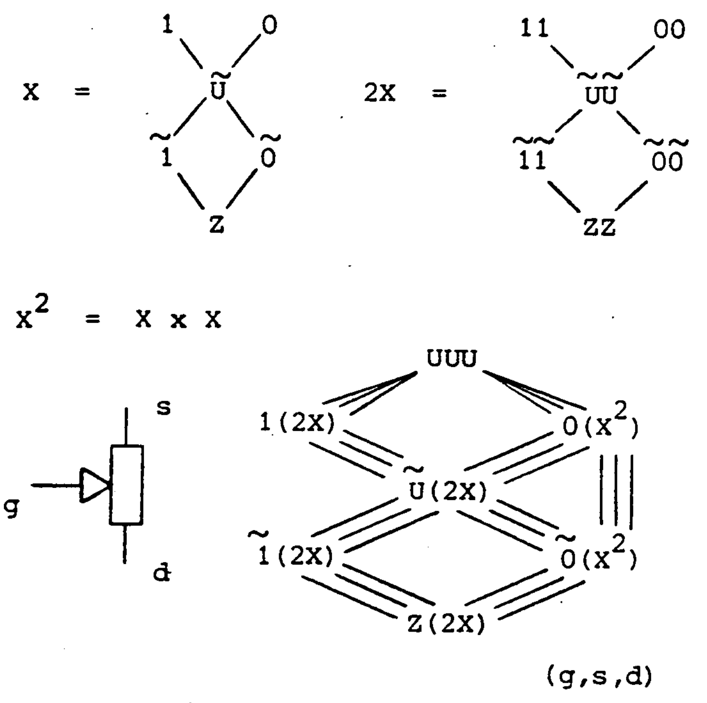


Figure 2.19 Switches

Three operations on configuration sets can be defined, corresponding to the net algebra operators. Restriction hides a port in each of the configurations of a set:

$$C_A\backslash a \ : \ A\backslash a \quad \triangleq \quad \{c{\downarrow}(A\backslash a) \mid c \in C_A\}$$

Renaming simply changes the port names of the configurations:

$$C_A\{r{:}A{\cong}A\} \ : \ r(A) \quad \triangleq \quad \{c \circ r^{-1} \mid c \in C_A\}$$

Composition merges two configuration sets C_A and $C'_{A'}$. Two configurations $c \in C_A$ and $c' \in C'_{A'}$ are compatible if they give the same values to the ports which are being connected. The result of the composition is then the set of all the compatible pairs of configurations, which are merged pairwise while hiding, as usual, the connected ports. In other words, the composition of two configurations is stable if the component configurations are stable and also the connection points are stable, i.e. if the values of the connection points agree in the two configurations.

$$C_A \mid C'_{A'} : A \oplus A' \triangleq$$
$$\{c \oplus c' \mid c \varepsilon C_A;\ c' \varepsilon C'_{A'};\ \forall a \varepsilon A \cap A'.\ c(a) = c'(a)\}$$

where $c \oplus c' \triangleq \{\langle a \mapsto v \rangle \mid a \ \varepsilon \ A \oplus A'\}$.

Let us see some examples; the first one is a "short circuit" configuration which has a unique (undefined) stable state:

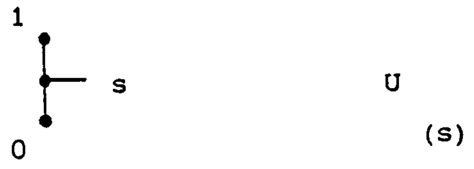


Figure 2.20 Short circuit

An attenuator connected to 1 has instead five configurations, which are the configurations of the attenuator lattice having s=U or s=1:

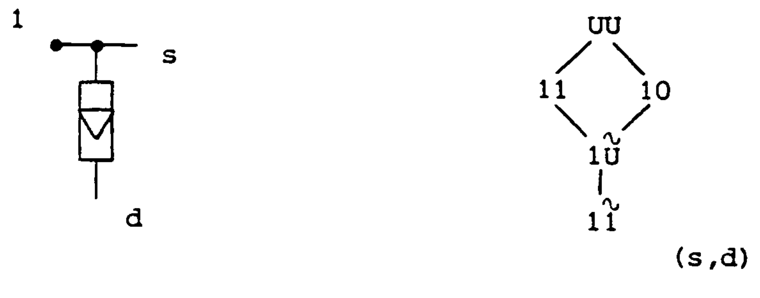


Figure 2.21 Powered attenuator

Here is a switch with the gate connected to the source:

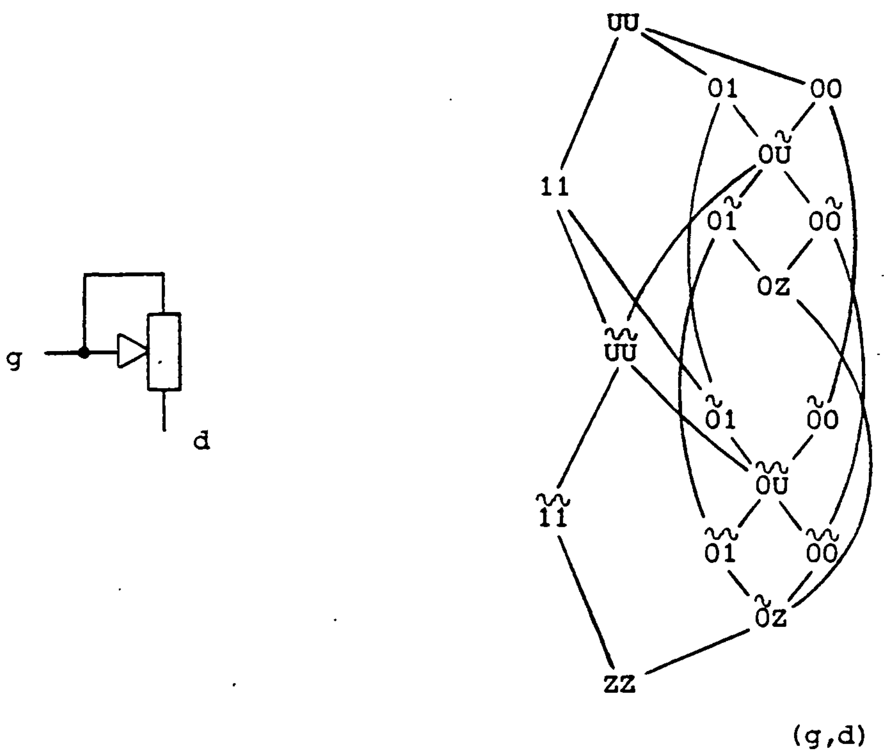


Figure 2.22 Gate-to-source switch

this large number of possibilities reduces to just two if we connect the gate to 1:

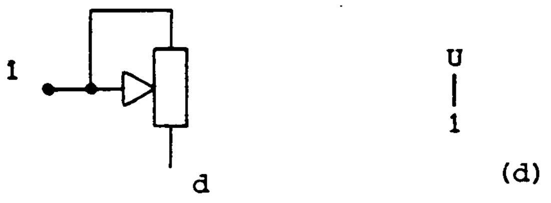
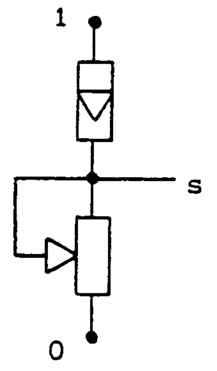


Figure 2.23 Powered gate-to-source switch

Finally, we can use a gate-to-source switch to build an oscillator:



U

(s)

Figure 2.24 Oscillator

initially there is a $\tilde{1}$ on the gate of the switch, which closes connecting the gate to 0; then the switch opens again, and so on forever. The static semantics of the oscillator contains a single undefined configuration; all the other ones are unstable (note that in the diagram for the gate-to-source switch there is no configuration with $g=\tilde{1}$ and $d=0$).

Given a circuit and its configuration set C_A we might ask whether the circuit is "well-behaved". This can be done by assigning well-defined values (i.e. T or F) to some terminals designed as inputs, and check that all the other terminals (the outputs) stabilise in a unique and well-defined way. Formally this means that if we choose the well-defined values $v_1 \ldots v_n$ for the ports $a_1 \ldots a_n \in A$, then the set $\{c \in C_A \mid c(a_i) = v_i\}$ should contain a unique totally well-defined configuration.

Finally, note that the configuration set for switches is not a lattice. This is sensible because if it were a lattice, we would have lattices as semantics for all the literals. Then the semantics of every CSA circuit would be a lattice, because the operations preserve lattices, and we would be able to define a uniquely determined relaxation operation mapping any arbitrary configuration into the least upper bound of all the stable configurations bigger than it. This would mean that every circuit could stabilise in a unique "most defined" way, which is not what happens in reality when, for example, we power up a flip-flop.

2.4.5 Main Example

The CSA layout of a positive bit comparator is shown in the next figure.

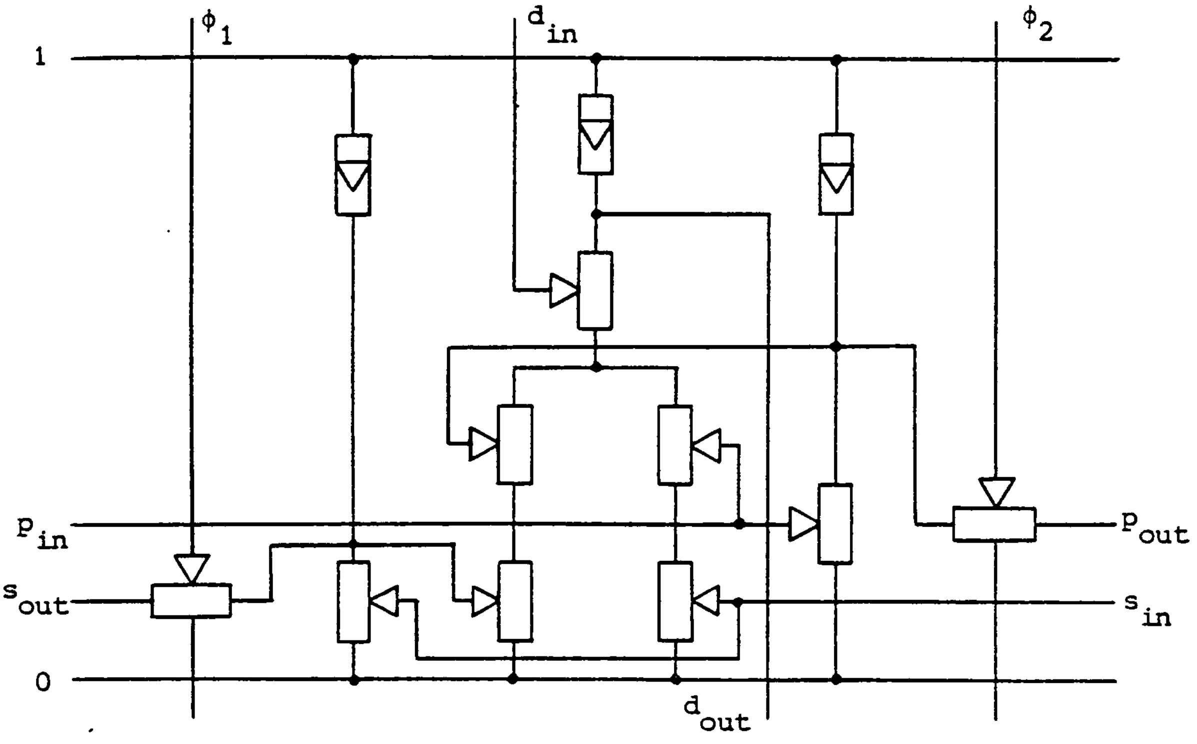


Figure 2.25 Positive bit comparator

The bit comparator can be split vertically into two clocked inverters which act as shift registers, surrounding a proper comparator implemented as a 5-switch pulldown structure.

2.4.6 From CTA to CSA

The basic idea underlying the translations among net algebras is that structure is preserved, i.e. net algebra literals and operators are, more or less directly, mapped into similar literals and operators of another net algebra. These translations are not, technically, algebra homomorphisms because literals may be mapped to complex nets with different sorts and signatures, and a single composition can be mapped to a set of compositions. What is needed here is a more general kind of algebra morphism called a derivor [Goguen 78, Sannella 81].

The preservation of structure implies that our translations are essentially simple because they only act locally, and hopefully the redundancies possibly introduced by the translation process can be removed by local optimisation.

Moreover, if structure is preserved, then the programmer has fine control on the structure of the end product. This characteristic makes our notations suitable for expressing special purpose hardware, where the emphasis is always on how a computation is carried out, rather then on the input-output behaviour. This is to be contrasted with the attitude one might take to translating arbitrary Algol-like programs in arbitrary (correct) ways into general purpose hardware components (e.g. microprocessors and read only memories). This approach can only produce standardised architectures, unless complex optimisation strategies are applied in order to rediscover the particular architecture one had in mind.

We show here how to systematically translate CTA expressions into CSA. The major step consists in translating CTA literals into CSA networks; the translation of the operators is then induced.

We assume that in a clocked transition $t \rightsquigarrow b$, t is built from input variables, boolean expressions and conditionals. Each value is translated into a VDD line, a GND line and a value line. For example an input variable of a phase-i cluster is translated as an appropriate forking, clocked by phase i:

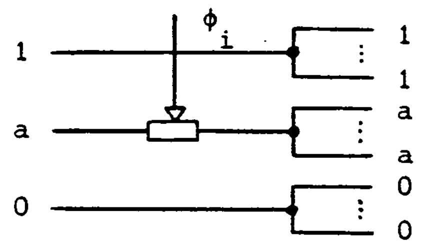


Figure 2.26 Input variable

Boolean operations like not, and, or etc. can be implemented by standard gates:

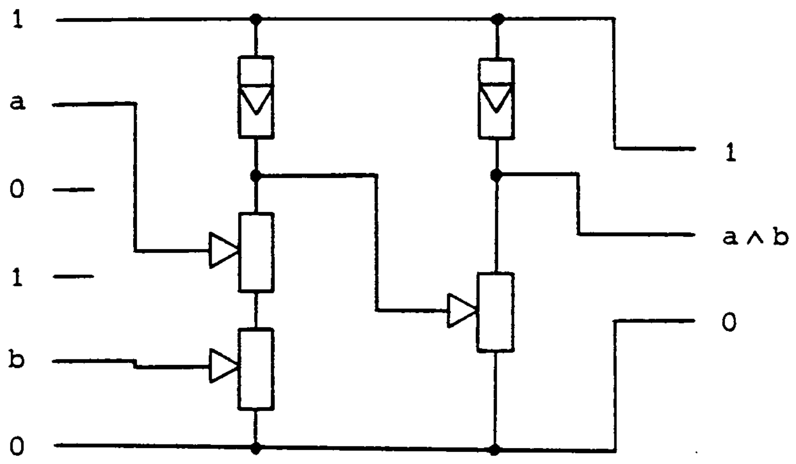


Figure 2.27 And

and conditionals "a ⇒ b ; c" become:

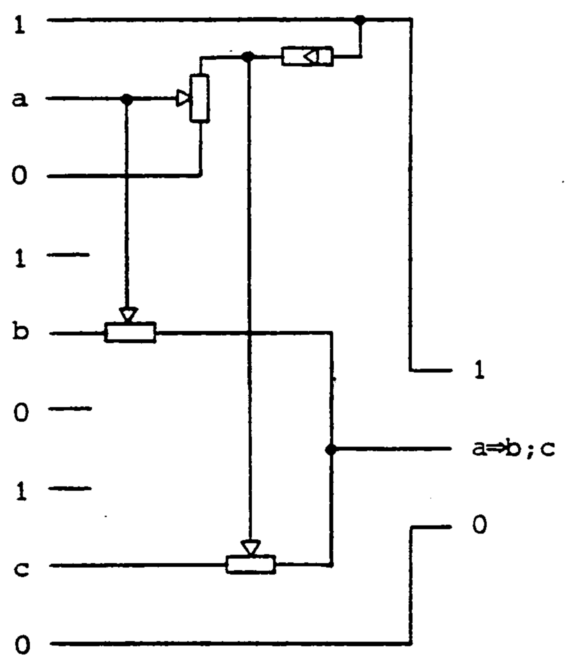


Figure 2.28 Conditional

Transitions t⇝b of phase i are translated by first translating t and then composing an output box to the output. Phase-1 output boxes are simply:

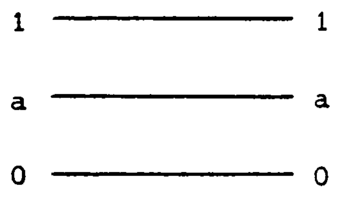


Figure 2.29 Phase-1 output box

while Phase-2 output boxes must consider the power-up values specified in the transitions; there are three cases:

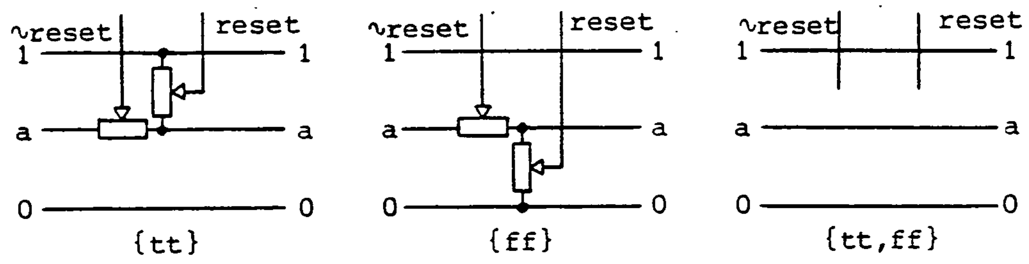


Figure 2.30 Phase-2 output boxes

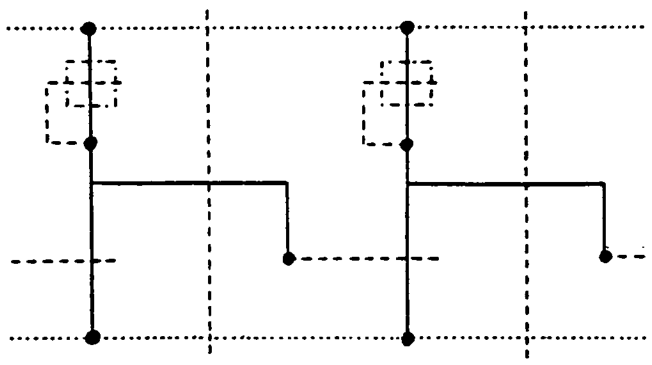
The CTA operators are translated into corresponding CSA operator, noting that the CSA expressions have 1's and 0's in their sort, and these ports have to be properly connected.

The translation proposed above is only a simple example, and it is very inefficient by VLSI standards. In fact, the signals are restored at each step of the translation and the large number of pullups introduced have large area and power requirements. A better result can be obtained by translating each value into a pair of lines (carrying the value and its complement), plus VDD and GND lines, and introducing a restoring stage into the output box. The translation for boolean operations and conditionals has to be modified accordingly, and local optimisation can group most of the logic into pulldown and pass transistor structures inside the output box. A translation of this kind, for CMOS circuits, is sketched in [Rem 81].

2.5 Stick Networks

2.5.1 Sticks

Stick diagrams were devised as an attempt to abstract away from detailed geometric layouts while still retaining their essential topological information content. Assuming that the design rules are known and that electrical properties are ignored, a stick diagram is about the minimal information allowing a human or a program to reconstruct the original layout, or one very close to it. Stick diagrams are meant to specify the choice of materials (i.e. colours) and to hint at the general position and orientation of lines and components, but to leave the exact geometry (and hence, the electrical properties) of the circuit unspecified. The geometric implementation of a stick diagram is usually one of the smallest obtainable according to the geometry rules, unless the context requires otherwise; in the latter case some stretching or routing is required.



Code: ------red; ——green; ··········blue; -·-·-·yellow.

Figure 2.31 A shift register stick diagram

There are two evident ways to analyse a stick diagram. The first is to identify coloured lines, black dots and yellow patches as basic constituents; a stick diagram is then an unstructured set of such components. While this can be convenient for some purposes (interactive graphic stick editors often work this way) we prefer to look for a hierarchical decomposition, in order to turn stick diagrams into a network algebra. The second approach is then to identify the stick intersections (i.e. transistors, contacts and crossovers) as the primitive objects, and to express stick diagrams as hierarchies of connected intersections. Stick intersections also happen to correspond to functional units in VLSI circuits, so that the second approach is very helpful in relating semantic and syntactic properties of circuits.

A stick diagram is a planar network. As explained in Chapter 1, the ports of a planar network are organised into a cycle which represents the anticlockwise order in which the ports appear in a layout. This cyclic structure is preserved during composition, so that starting from planar primitives we can only build planar graphs. Here is an example of a sort:

{rs,re,rn,rw: red}

where rs,re,rn,rw (i.e. red south, red east, etc.) are all red ports, and the cyclic order is rs<re<rn<rw<rs. Port names have no particular significance, types are the three colours {green,red, blue}

Two planar sorts are equal if they have the same set of ports, associate the same types to the same port names, and if the cyclic ordering of ports is the same. Swapping ports around the perimeter is forbidden by the cyclic ordering, so that no non-planar crossovers or unwanted transistors are generated. This constraint is actually stronger than needed because it also forbids red-blue and green-blue swappings and requires the introduction of red-blue and green-blue crossovers among the literals.

2.5.2 Stick Expressions

A **stick expression** is a planar expression denoting a stick network. Stick expressions are built from a set of literals denoting the basic building blocks of stick diagrams. The set of literals is listed here, together with their interpretation.

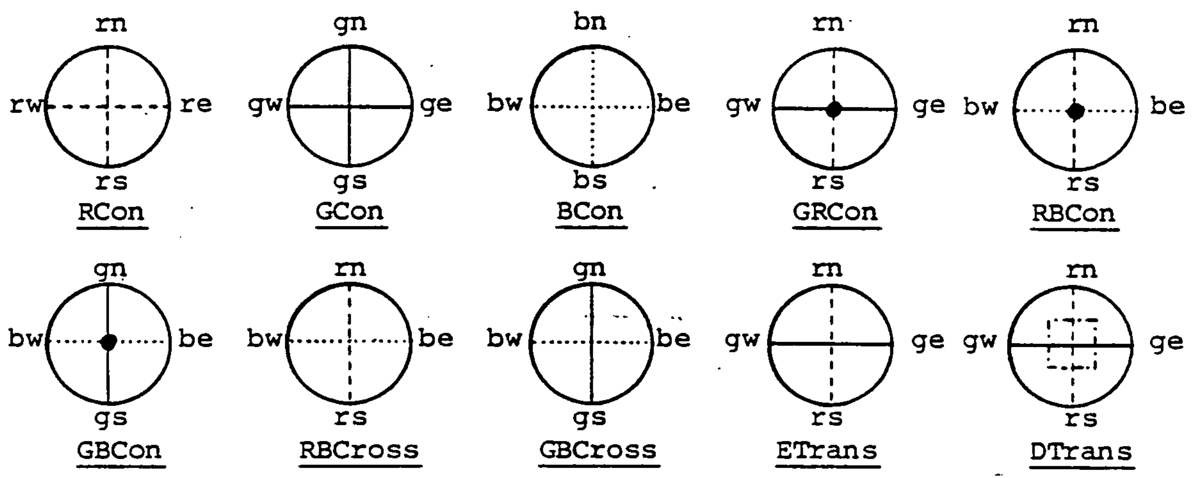


Figure 2.32 Stick literals

There is a very simple correspondence between stick literals and CSA literals. ETrans is a switch with gate at "rn" and "rs", source at "ge" and drain at "gw". RCon,GCon,BCon,GRCon,RBCon and GBCon are 4-terminal connectors. RBCross,GBCross and DTrans are crossovers. An attenuator is a composite object, which in stick terminology is called a pullup:

```
PullUp =
    GRCon\rs\gw [rn--rn]
    Dtrans\rs{ge\gn} [ge--gw,gw--gn]
    GCon\ge
```

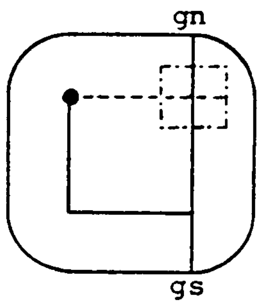


Figure 2.33 A pullup

2.5.3 Examples

An inverter can be built by first defining VDD (power supply) GND (ground) and PullDown components:

```
VDD  =  GBCon\gn {bw\VDD_in,be\VDD_out}
GND  =  GBCon\gs {bw\VDD_in,be\VDD_out}
PullDown  =  ETrans\rs{gn\ge,gs\gw}
```

an then composing them vertically with a PullUp:

```
Inverter  =
   VDD [gs--gn]
   PullUp [gs--gn]
   GCross\gw{out\ge} [gs--gn]
   PullDown{in\rn} [gs--gn]
   GND
```

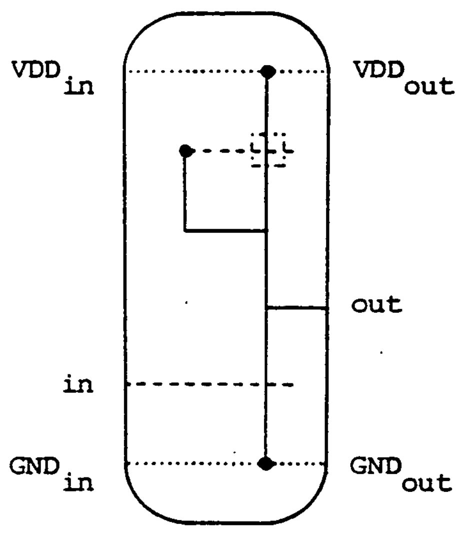


Figure 2.34 An inverter

More complex examples involve parametric definitions, local definitions (let-in), conditionals and recursion or iteration (we have already seen some examples in Chapter 1).

In VLSI most of the parametric structures are regular arrays of cells, and in these cases iteration is the most obvious programming construct to use. We introduce iteration in the following specialised form which applies only to the iterated connection of sticks:

```
for <variable> in <list>
iter <body>
with <connection>
```

<list> is an expression denoting a list of integers, e.g. n..m is the list of integers from n up to (or down to) m; <body> is an expression denoting a stick diagram i.e. a stick expression augmented by control structures like "let-in", "if-then-else" and "for-iter"; and <connection> is an explicit composition operator

[r]. The body of the iteration (possibly containing the iteration variable) is composed by the connection part to the accumulated result of the previous iterations, while the iteration variable ranges through the list. Bunched sorts are used extensively: iteration very often bunches together many of the ports which are not connected. Note that the form of iteration we use can be easily translated into recursion, and of course no side effects are involved.

We now program the tally circuit example of [Mead 80]. A tally circuit has n inputs and n+1 outputs, and the output k is high if and only if k of the inputs are high. We are not interested here in the behaviour of the tally circuit, but in its rather unusual triangular topology. The reader is advised to draw the pictures corresponding to the expressions we present.

First let us define the basic tally cell:

TallyCell =

NegPart [res_{out}--res_{in}] PosPart

NegPart =

NegCross [neg_{in}--neg_{out}] NegGate

PosPart =

PosCross [pos_{in}--pos_{out}]

PosGate [de--gs, res_{out}--gw]

GCon{gn\de, ge\res_{out}}

NegCross =

GBCross{gn\dw, bw\$\mathrm{neg}_{in}$, gs\dn, be\$\mathrm{neg}_{out}$}

PosCross =

GBCross{gn\$\mathrm{res}_{in}$, bw\$\mathrm{pos}_{in}$, gs\$\mathrm{res}_{out}$, be\de}

```
NegGate  =
  GBCross{gn\res_in, be\neg_out} [gs--gw]
  ETrans\rn{ge\res_out} [bw--be,rs--rs]
  RBCon\rn{bw\neg_in}
PosGate  =
  GBCross{gs\de, be\pos_out} [gn--ge]
  ETrans\rn{gw\ds} [rs--rn,bw--be]
  RBCon\rs{bw\pos_in}
```

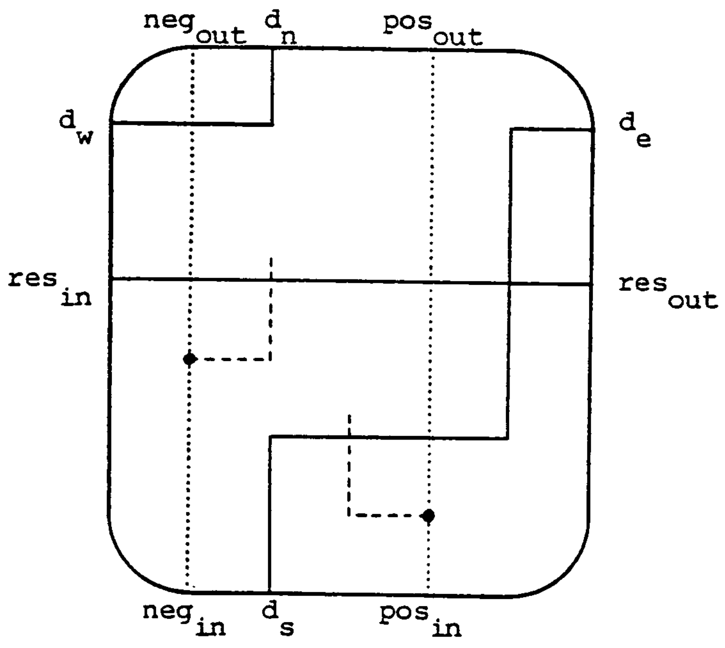


Figure 2.35 Basic tally cell

Then the central part of the circuit can be composed by a double iteration:

```
TallyBody n  =
   for j in [1..n+1]
   iter (for i in (if j=n+1 then 1..n else 1..j)
           iter TallyCell
           with [dw--de, res_in--res_out])
         [dw--gn, res_in--ge]
         GCon\gs{gw\res_in}
   with [neg_in--neg_out, ds--dn, pos_in--pos_out]

Tally n  =
   TallyBody n \de \dn \pos_out \neg_out
   [res_in[n+1]--gs]
   PullUp
```

note the debunching operation used to connect the pullup to the tally body. The inputs are collected in the n-bunch pos_{in} and neg_{in} are their negations; res_{out} are the outputs; the pullup should be connected to VDD and all the remaining ports to GND.

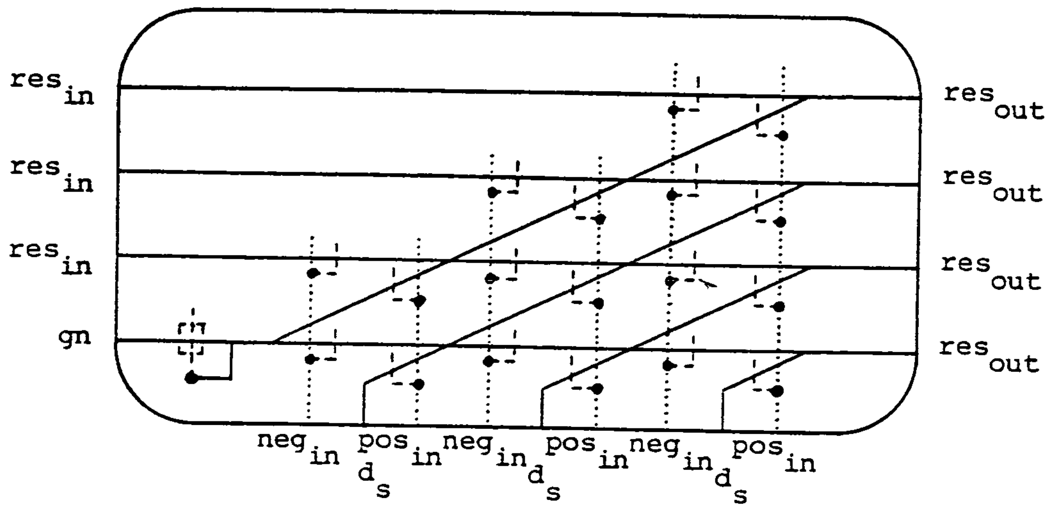


Figure 2.36 Tally circuit

The next example is a PLA generator (PLA structures can implement arbitrary finite state machines [Mead 80]). The generator accepts as inputs two arrays of boolean values, (which can be automatically generated from sets of boolean equations) coding the disposition of switches in the so-called and and or planes.

We first introduce the basic building blocks of the PLA as pictures:

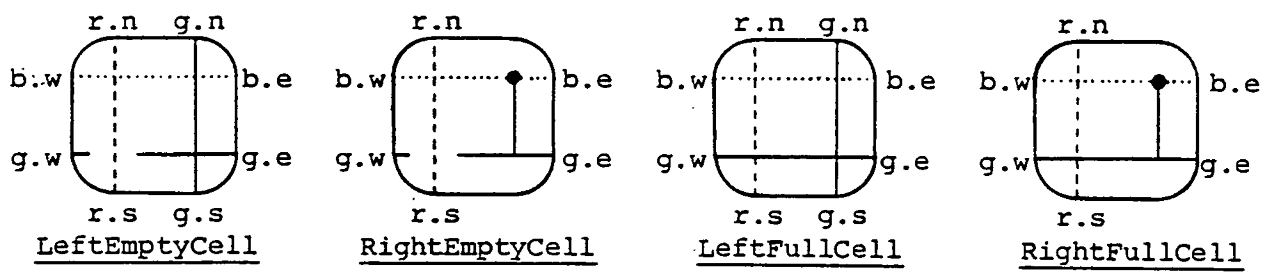


Figure 2.37 Building blocks for programmed cells

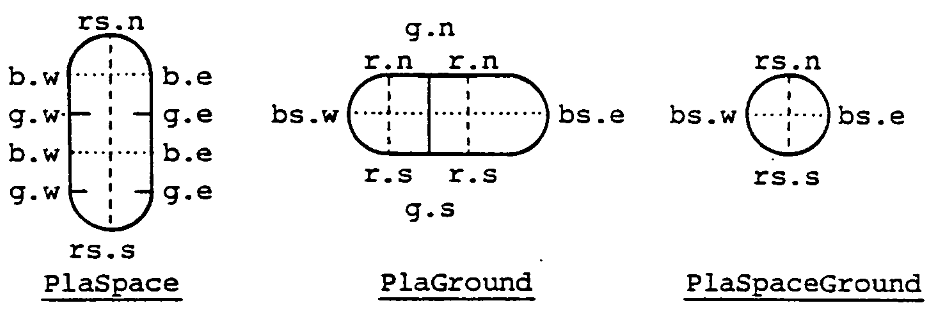


Figure 2.38 Building blocks for ground lines

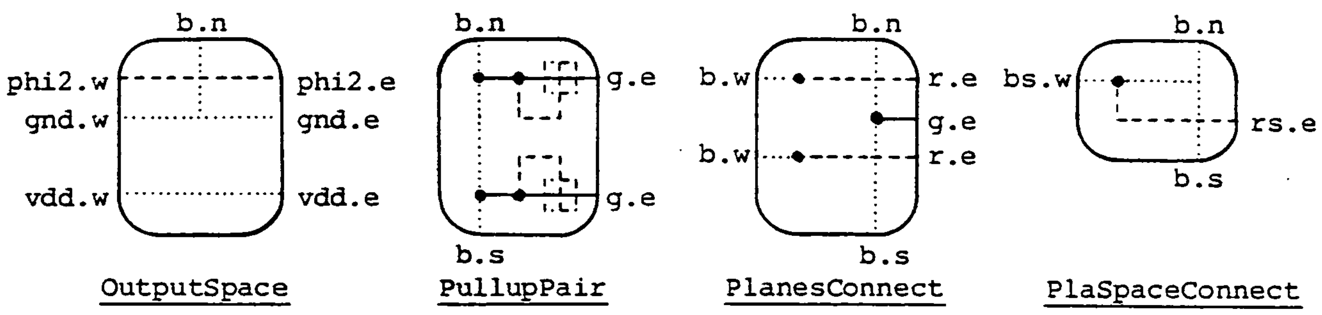


Figure 2.39 Peripheral building blocks

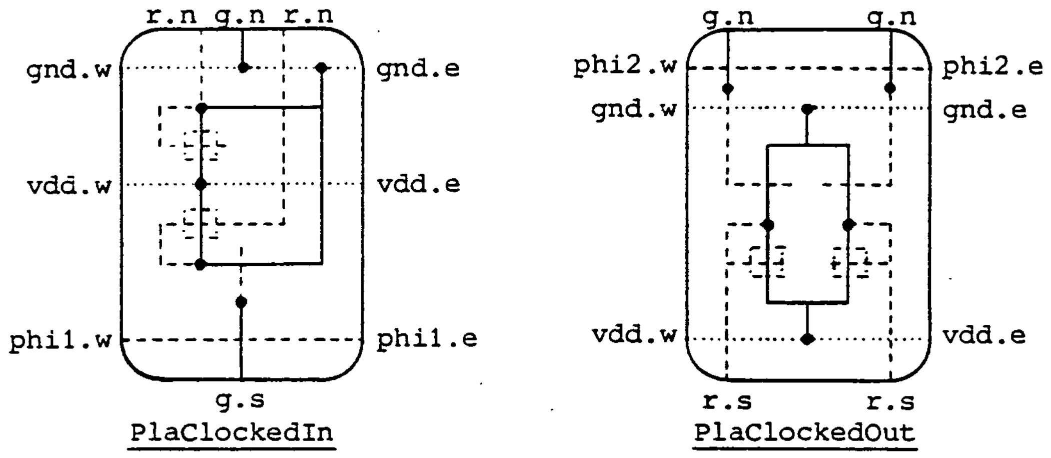


Figure 2.40 Input and output

The following program generates a single plane; the inputs are a pattern (i.e. an array of booleans arranged for convenience as a list of lists of quadruples of booleans) and the frequencies with which ground lines have to be interleaved with cells, in the horizontal and vertical directions. Note that we use here simultaneous iteration of two iteration variables through two lists.

```
let plane (pattern,Xspace,Yspace) =
  for strip in pattern and Y in 0..(length pattern)-1
  iter (PullupPair [b.e--b.w, g.e--g.w] row)
```

```
      where row =
      (for highleft,highright,lowleft,lowright in strip
       and X in 0..(length strip)-1
       iter (if Y mod Yspace = 0
             then if X mod Xspace = 0 And Not X = 0
                  then (PlaSpace [b.e--b.w, g.e--g.w] cell)
                       [r.n--r.s, g.n--g.s, rs.n--rs.s]
                       (PlaSpaceGround [bs.e--bs.w] PlaGround)
                  else cell [r.n--r.s, g.n--g.s] PlaGround)
             else if X mod Xspace = 0 And Not X = 0
                  then PlaSpace [b.e--b.w, g.e--g.w] cell
                  else cell
             where cell = cell(highleft,highright,lowleft,lowright)
       with [b.e--b.w, g.e--g.w, bs.e--bs.w])

    with [b.s--b.n, r.s--r.n, g.s--g.n, rs.s--rs.n]
    \r.n \g.n \rs.n \bs.w

    where cell(highleft,highright,lowleft,lowright) =
       (if highleft then LeftFullCell else LeftEmptyCell
        [b.e--b.w, g.e--g.w]
        if highright then RightFullCell else RightEmptyCell)
       [r.s--r.n, g.s--g.n]
       (if lowleft then LeftFullCell else LeftEmptyCell
        [b.e--b.w, g.e--g.w]
        if lowright then RightFullCell else RightEmptyCell)
```

Note that the complication of the inner iteration loop is due only to the interleaving of ground lines.

The generator composes an and-plane and an or-plane via a connection strip. The and-plane is obtained by connecting the inputs to a plane, and the or-plane by connecting the outputs to another plane.

```
let pla (andpattern,orpattern,space) =
  (andplane [gnd.e--gnd.w, vdd.e--vdd.w, b.e--b.w, bs.e--bs.w]
   conn     [gnd.e--gnd.w, vdd.e--vdd.w, phi2.e--phi2.w,
             r.e--r.s, g.e--g.s, b.n--b.s]
   orplane) \phi1.e \phi2.w

where andplane =
   plane (andpattern,length(hd andpattern),space) \ge
   [r.s--r.n, g.s--g.n, b.s--vdd.w]
   inputs

and orplane =
   plane (orpattern,space,space)
   [g.e--g.n, bs.e--b.n]
   outputs

where inputs =
   length(hd andpattern) times PlaClockedIn
   with [gnd.e--gnd.w, vdd.e--vdd.w, phi1.e--phi2.w]
   \gnd.w {g.s\in.s}
```

```
and outputs =
   for Y in 1..(length orpattern)
   iter (if (Y mod space)=0
         then PlaClockedOut
                 [gnd.e--gnd.w, vdd.e--vdd.w, phi2.e--phi2.w]
              OutputSpace
         else PlaClockedOut
   with [gnd.e--gnd.w, vdd.e--vdd.w, phi2.e--phi2.w]
   {r.s\out.s}


and conn =
   for Y in 0..(length pattern)-1
   iter (if (Y mod space)=0
         then PlaSpaceConnect [b.s--b.n] PlanesConnect
         else PlanesConnect
   with [b.s--b.n] \b.n
   [b.s--b.n]
   OutputSpace
```

A version of this program was written in the design system described in Chapter 3 (the only differences, due to the geometric nature of that language, being the use of geometric literals, and the use of some "geometric renaming" (see Chapter 3)). The result is shown in the next figure.

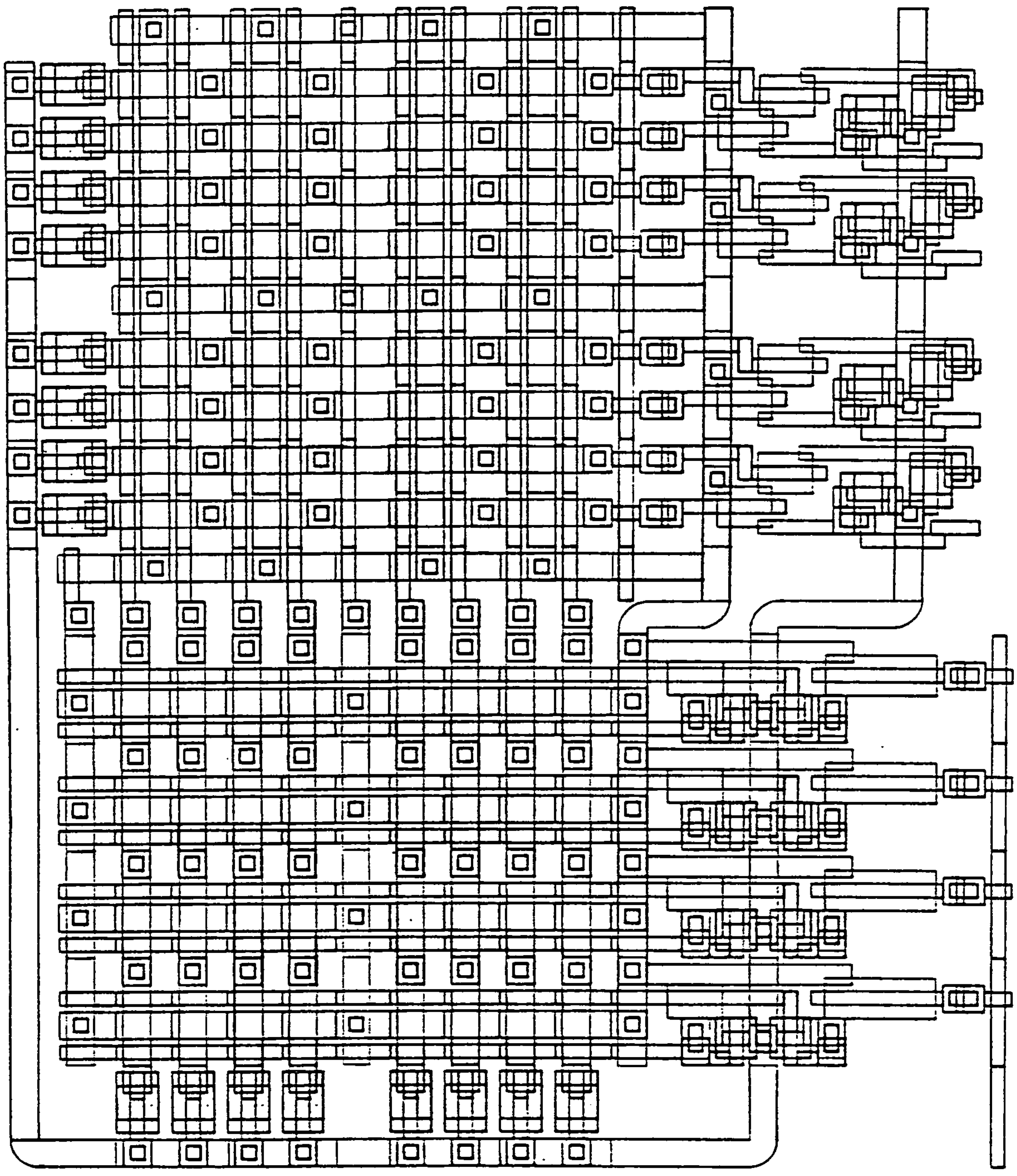


Figure 2.41 PLA layout

2.5.4 From CSA to Sticks

Three steps are needed to translate a CSA network into a stick network. Here we simply sketch them.

The first step consists in finding an almost-planar embedding for the graph of a CSA expression, imposing a planar sort on the CSA network and possibly preserving the structure of the expression. Note that a stick graph does not need to be completely planar because of the crossover literals RBCross,GBCross and DTrans.

Planar embeddings are always possible by inserting extra crossover components at critical points. The result should be reasonably good if the initial CSA network was thought of in planar terms (as should often be the case for VLSI networks), otherwise very complex algorithms and heuristics will probably be needed to get good results.

The second step is the "colouring" of the graph. Components like switches, attenuators, power, ground and clocks have precisely coloured terminals, and a simple colour propagation scheme (where terminals of connectors may receive arbitrary colours) should be sufficient to colour the whole graph.

The third step simply translates attenuators into pullup structures, switches into transistors and connectors into wires and, when needed, contacts.

2.6 Grid Networks

We are now going to investigate a net algebra which is situated, so to speak, in between purely topological stick networks and purely geometrical layout networks. This algebra can be of practical significance because it seems to minimise the complexity of the translations from sticks to layouts; it can be regarded as a very abstract geometrical algebra or as a very concrete topological one.

2.6.1 Grids

A grid is an array of orthogonal segments such that all the vertical segments intersect all the horizontal ones, and vice versa. For convenience we shall lay the segments parallel to the axes of the cartesian plane with spacing two units, end-points projecting of one unit outwards, and with the origin in the lower left corner.

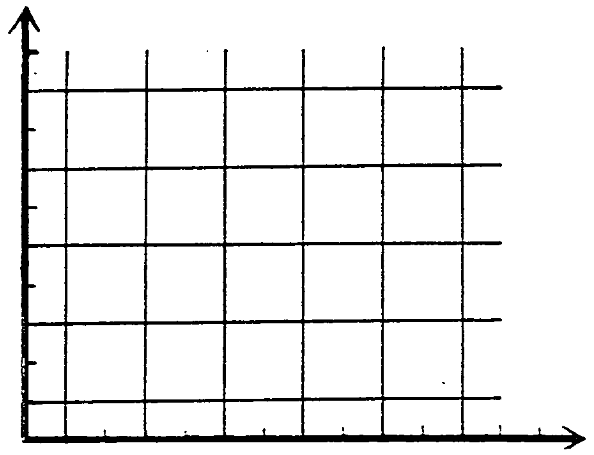


Figure 2.42 A canonical grid

The end-points of segments in a grid are called its **ports**; the **boundary** of a grid is the set of its ports and the **perimeter** is given by the cardinality of the boundary. The south,east,north and west boundaries are defined in the obvious way and are also called respectively the **south**,**east**,**north** and **west** of the grid; collectively these are the **sides** of the grid. The **knots** of a grid are the intersections of its segments, and the area is the number of knots.

A grid can be regarded as a rectangular matrix of knots. An interpretation of a grid is a mapping from its knots into a set T of tiles (which are little 2x2 squares). Here is the set of basic tiles needed for nMOS stick diagrams.

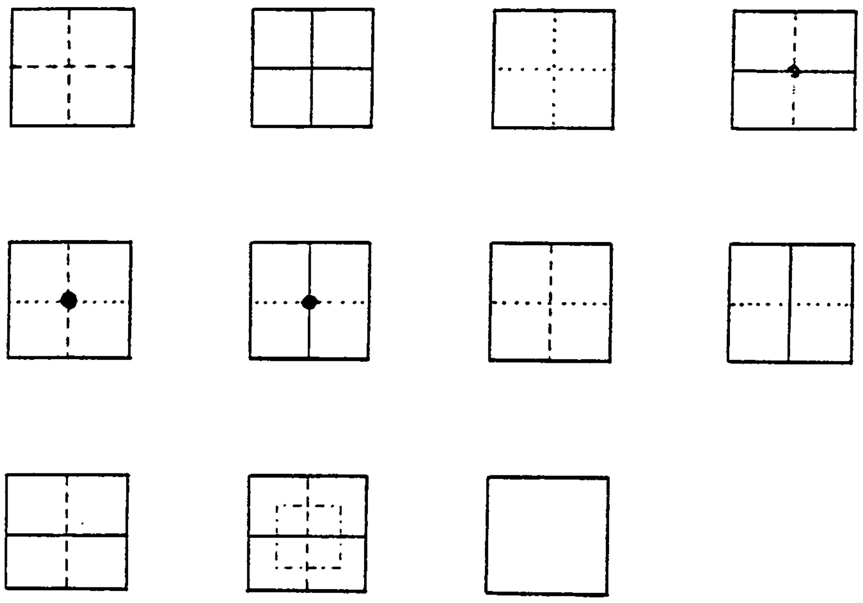


Figure 2.43 Basic tiles for nMOS stick diagrams

Non-basic tiles can be produced from the above tiles by rotation and by dropping one or two of the segments joining the centre of a tile to its boundary; the blank tile is needed to fill the empty spaces of a stick diagram. An interpretation of a grid according to this set of tiles is given in the next figure, showing a shift register cell.

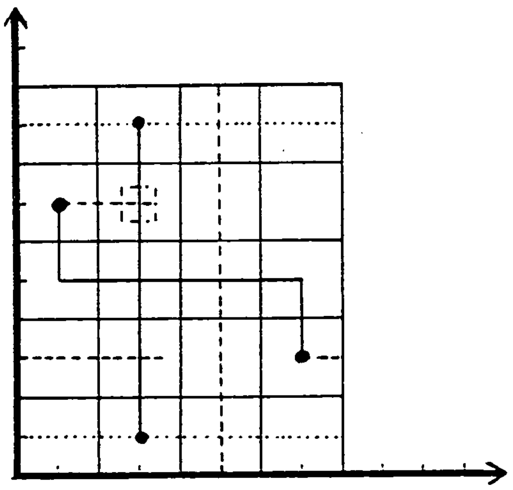


Figure 2.44 A stick diagram interpretation of a grid

The sort of a grid associates a **name** and a **type** to each port. The ports are cyclically ordered anticlockwise, and assigned to one of the four sides {south, east, north, west}. There is a special name called the **NullName** and a special type called the **NullType**; the NullName is always associated to the NullType and vice versa. A pair NullName-Nulltype is also called a **null port**, written "()", which represents the lack of a proper port on the perimeter of the grid. For example, the sort of the shift register cell is written:

```
{south: [(), (), ClockOut: red, ()],
 east:  [GndOut: blue, Out: red, (), (), VddOut: blue],
 north: [(), ClockIn: red, (), ()],
 west:  [VddIn: blue, (), (), In: red, GndIn: blue]}
```

A **grid network** (sometimes ambiguously called a grid) is an interpreted grid together with a compatible grid sort. Grid networks can be built by repeated compositions, starting from a set B of **basic grids** (each b ε B being a rectangular assembly of tiles t ε T) of sort given by λ(b). A **composition** g'[r]g" of two grids g',g" is obtained by embedding without overlapping g' and g" into a bigger grid g. This embedding must satisfy [r] (in the evident sense) and must also define the tiles of g which do not belong to g' or g". A particular kind of grid composition will be analysed in a forthcoming section.

The two other net operations are defined on grids as follows (they just operate on the sorts leaving the underlying grids unchanged):

Restriction: g\a transforms the name a in the sort $\sigma(g)$ into NullName, and the associated type into NullType (a should not be NullName).

Renaming: g{r} applies the name-bijection r to the names in $\sigma(g)$, leaving the types unchanged (r should not contain any NullName).

2.6.2 Discrete Stretch Transformations

In this section we develop some tools needed to define grid compositions. An n-dimensional discrete stretch transformation, or stretching for short, is an n-tuple of boolean vectors $\bar{S} = S_1..S_n$ (we are actually only interested in the cases n=1 and n=2). For every boolean vector S_i we define $\#S_i$ as the length of the vector and ρS_i as the number of "1"'s (i.e. "true"'s) in the vector. If $M_{m_1..m_n}$ is an n-dimensional matrix of size $m_1 X..X m_n$, then a n-dimensional stretching can be regarded as a mapping:

$$S_1..S_n : M_{\rho S_1..\rho S_n} \hookrightarrow M_{\#S_1..\#S_n}$$

The result matrix is obtained from the argument matrix by inserting an (n-1)-dimensional plane orthogonally to the i-th dimension in correspondence of every "0" in S_i. For example:

$$s_1 = 1\ 0\ 1\ 0,\ s_2 = \begin{matrix}1\\0\\1\end{matrix} : \begin{matrix}a & b\\ c & d\end{matrix} \hookrightarrow \begin{matrix}a & * & b & *\\ * & * & * & *\\ c & * & d & *\end{matrix}$$

where * is any fixed fill-in value.

In the case of 1-dimensional stretching we can apply a stretching to another stretching (using 0 as fill-in value). This allows us to define the composition of stretchings as follows:

$$S_1..S_n \circ S'_1..S'_n \triangleq S_1(S'_1)..S_n(S'_n)$$

That this is really composition can be seen from the following properties:

$$(\bar{S} \circ \bar{S}')(M) = \bar{S}(\bar{S}'(M))$$

$$(\bar{S} \circ \bar{S}') \circ \bar{S}'' = \bar{S} \circ (\bar{S}' \circ \bar{S}'')$$

In case of 1-dimensional stretching we get the curious-looking equation:

$$(S \circ S')(S'') = (S(S'))(S'') = S(S'(S''))$$

2.6.3 Normal Grid Composition

We are interested in a particular kind of grid-network composition called **normal composition**. This composition might appear to be exceedingly restrictive; actually there is no loss of generality and we shall see in the following sections that any stick expression can be mechanically translated into a series of normal grid compositions in a non-unique but fairly controllable way.

Normal composition is determined (up to choice of stretch lines) by specifying the **connection side** s of one of the networks; the connection side of the other network is then taken to be the side opposite to s. For consistency with stick expressions we shall use the full notation g'[r]g" also in this case, where [r] is of the form [a_i--b_i] and a_i are all the non-null ports on one side of g' and b_i are all the non-null ports on the opposite side of g". We extend this notation to expressions like g'[south--north]g" in order to describe composition on sides with no non-null ports.

Normal composition is legal if and only if the number of non-null ports on the two connection sides is the same and the names and types of these ports match pairwise (according to [r]) moving in parallel along the connection sides. The result grid is obtained by minimally stretching the two component grids uniformly in the connection-side direction until the level of all the non-null ports match pairwise and the two connection sides have the same length.

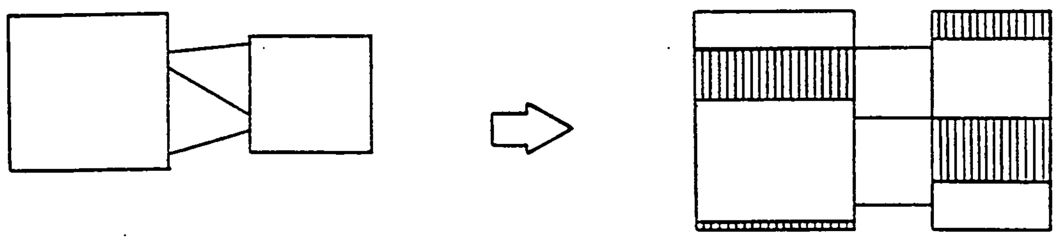


Figure 2.45 Stretching

The exact choice of stretch lines is not important, as far as stretching is minimal. The stretched grids are then embedded into a grid of area the sum of their areas, with the connection sides facing each other. The way in which the stretched areas are filled-in with tiles, depends in general on their neighbouring tiles, and should be specified together with the set of tiles T; in case of nMOS stick diagrams we fill these spaces by the appropriate straight-line tiles, so that connected stick nodes remain connected under stretching. The result sort of normal composition is the sort of the result grid, obtained from the component sorts by dropping the ports on the respective connection sides and by possibly inserting some null ports where stretching has occurred.

In order to compute the normal composition of grids, we might represent grids as matrices and then define grid composition by brute-force stretching of matrices. Instead, we describe an efficient algorithm which simulates this stretching process by considering grid sorts together with stretch transformations.

Let us assume that the grid composition g'[r]g" is normal and legal in the sense previously defined. Starting from the sorts of g' and g" we produce the sort of the result, together with the bidimensional stretch transformations v',h' and v",h" (vertical and horizontal respectively) to be applied to g' and g" in order to embed them exactly in the result.

The first step consists in identifying the connection sides in g' and g", which are given by any pairs of connections as specified in [r]. For convenience we fake a standard orientation for composition, placing g' on the left and g" on the right; we define the pseudo-east side of g' and the pseudo-west side of g" to be their respective connection sides, and accordingly we pseudo-name all the other sides of g' and g".

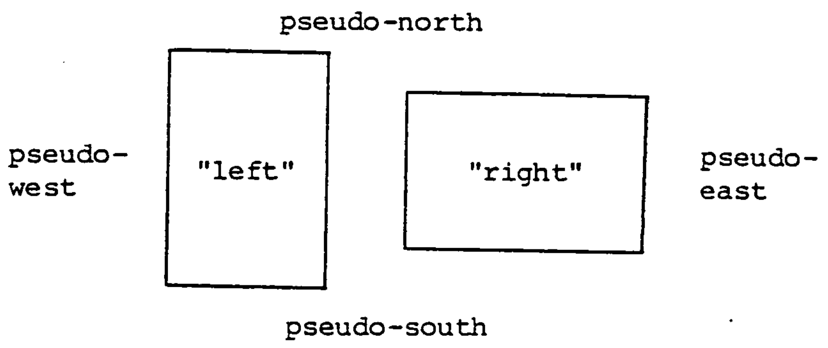
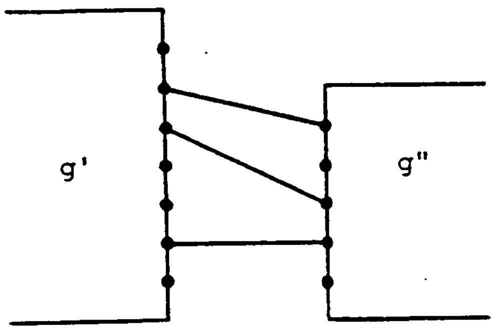


Figure 2.46 Pseudo orientation of composition

Next we compute a minimal pair of 1-dimensional stretch transformations, pseudo-v' and pseudo-v", which make the ports of g' and g" match along the connection sides. This can be done by walking in parallel on the pseudo-east and pseudo-west sides of $\sigma(g')$ and $\sigma(g")$, "skipping" all the null ports in pairs and "pushing" any non-null port along one side (while skipping any null port on the other side) until there is a non-null port on the other side, and then skipping the pair. Here "skipping" means inserting a "1" in the resulting transformation, and "pushing" means inserting a "0".



1	0
1	1
0	1
1	1
1	0
1	0
1	1
1	1
pseudo-v'	pseudo-v"

Figure 2.47 A minimal stretching

Next we form the sort of the result in the following way:

- The resulting pseudo-south is the concatenation of the two pseudo-south;
- The resulting pseudo-north is the concatenation of the two pseudo-north;
- The resulting pseudo-west is the result of stretching the pseudo-west of g' according to v', filling in with null ports;
- The resulting pseudo-east is the result of stretching the pseudo-east of g" according to v", filling in similarly.

Finally we produce the stretch transformations:

- pseudo-v' and pseudo-v" have already been produced;
- pseudo-h' is a vector of "1"'s as long as the pseudo-south of g';
- pseudo-h" is a vector of "1"'s as long as the pseudo-south of g".

All the results have to be renormalised with respect to the pseudo orientation. The sort of the result gives the total size of the composed objects and can be used in further compositions. The stretch transformations v',h' and v",h" and the connection sides are enough information for building a matrix of the result if we are given matrices for g' and g". They can also be composed in interesting ways with the stretchings computed from subexpressions of g' and g", as we shall see shortly.

2.6.4 Grid Expressions

A **grid expressions** is an expression with operators "\a", "{r}" and "[r]" (normal composition), over a set of **grid literals** denoting basic grids. The grid denoted by a (legal) grid expression is obtained by actually performing the operations described in the expression. Grid expressions will be denoted by the letter "g".

If we have a grid expression in form of a tree, we can apply the grid composition algorithm described in the previous section from the bottom up, obtaining at the end a corresponding tree of stretch transformations plus the grid sort of the whole expression. The stretch tree and grid sort of a grid expression g are produced as follows:

- If g is a literal, the stretch tree is a leaf containing that literal and the grid sort is the grid sort of g.
- If g is g'\a, we recur on g' obtaining its tree t' and sort s'. The result tree is t' and the result sort is s'\a (restriction as defined for grid networks).
- If g is g'{r}, we recur on g' obtaining its tree t' and sort s'. The result tree is t' and the result sort is s'{r} (renaming as defined for grid networks).
- If g is g'[r]g", we recur on g' and g" obtaining t',s' and t",s". We apply the grid composition algorithm to s',s" obtaining a sort s and two stretchings v',h', v",h". The result sort is s. The result tree contains t',v',h', t",v",h" and the connection side of g'.

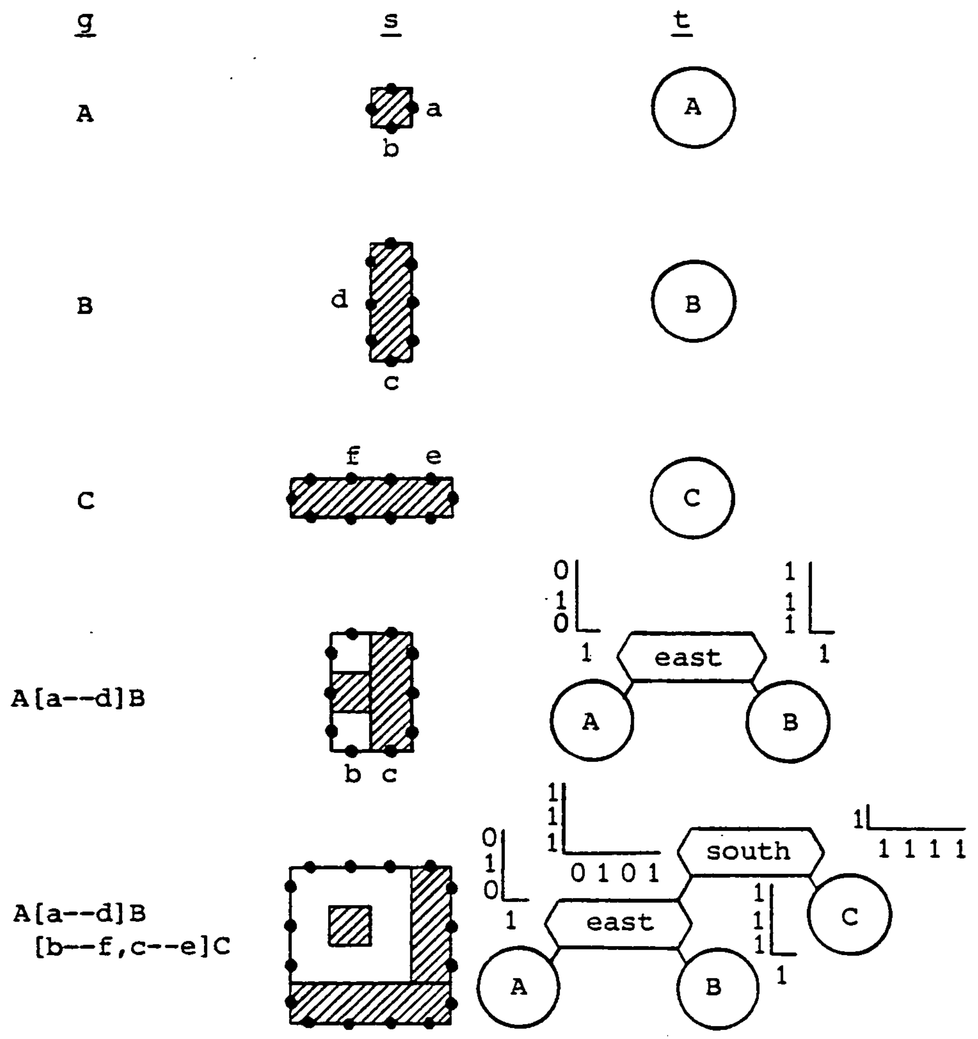


Figure 2.48 Stretch trees

2.6.5 Grid Recomposition

Given a grid expression we have seen how to produce a stretch tree and a grid sort for it. To produce a picture of a grid, we walk down the tree accumulating the stretch transformations as we proceed. When we get to a literal we know its position and the amount of stretching to be applied to it. Hence we draw the literal in the computed position with the appropriate stretching patterns to match its expected size.

The accumulation of stretch transformations is not done in the most obvious way, which would be by stretch composition: this method produces, for every literal, a stretch transformation as big as the whole final layout which places exactly the literal in the layout, but gives no information about the amount of stretching to be applied to it. Luckly enough, the kind of accumulation we need uses less space, and produces for every literal a stretch transformation as big as the stretching to be applied to the literal; the position of the stretched literal in the global layout is derived by maintaining an origin point as we go along.

The algorithm takes a stretch tree and its grid sort, and "draws" the result; drawing is just an example of grid recomposition. We start with a stretch tree t, a vertical-horizontal stretching v,h all made of "1"'s as big as the size of the grid sort, and an origin Or=0,0 (the lower left corner of the layout).

If the tree t is a leaf, it contains a grid literal. We then stretch the layout of this literal using v,h and draw it starting from the origin Or.

If the tree t is a composition node, suppose it was generated by the composition g'[r]g". Then t contains two subtrees t',t" (corresponding to g' and g") two stretchings v',h' and v",h", and the connection side of g'. Let us make up a pseudo orientation with the connection side of g' on the east, modifying v,h etc. appropriately into pseudo-v,pseudo-h, pseudo-v',pseudo-h', pseudo-v",pseudo-h" and pseudo-Or.

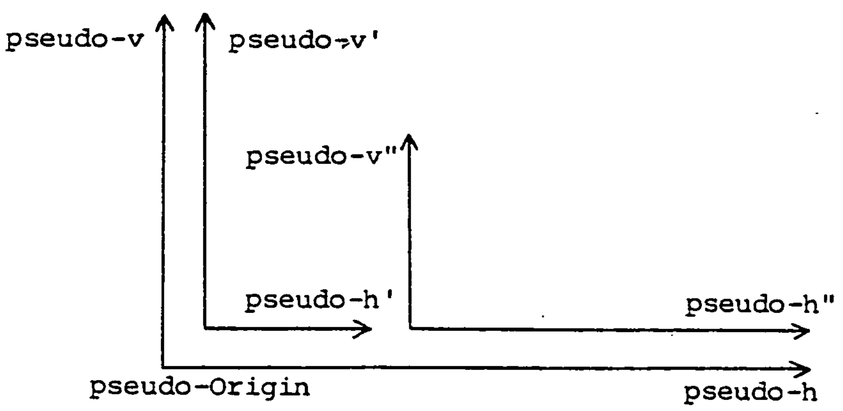


Figure 2.49 Pseudo orientation for stretching

We need to compute the stretchings and origins to be passed down recursively to the subtrees; let's call them newv',newh', newv",newh", Or' and Or". They can be obtained by "unpseuding" the following definitions:

pseudo-newv' = pseudo-v $\circ$ pseudo-v'

pseudo-newv" = pseudo-v $\circ$ pseudo-v"

pseudo-newh',pseudo-newh" = split(pseudo-h,pseudo-h',pseudo-h")

pseudo-Or' = pseudo-Or

pseudo-Or" = (pseudo-Or.x + length(pseudo-newh')),pseudo-Or.y

where split(h,h',h") splits h into two parts newh',newh" such that newh' concatenated to newh" is equal to h; ρnewh'=ρh' and ρnewh"=ρh" (it does not matter where the split exactly occurs). Nothing is drawn for composition nodes, and we recur with t',v',h',Or' on one side and t",v",h",Or" on the other side.

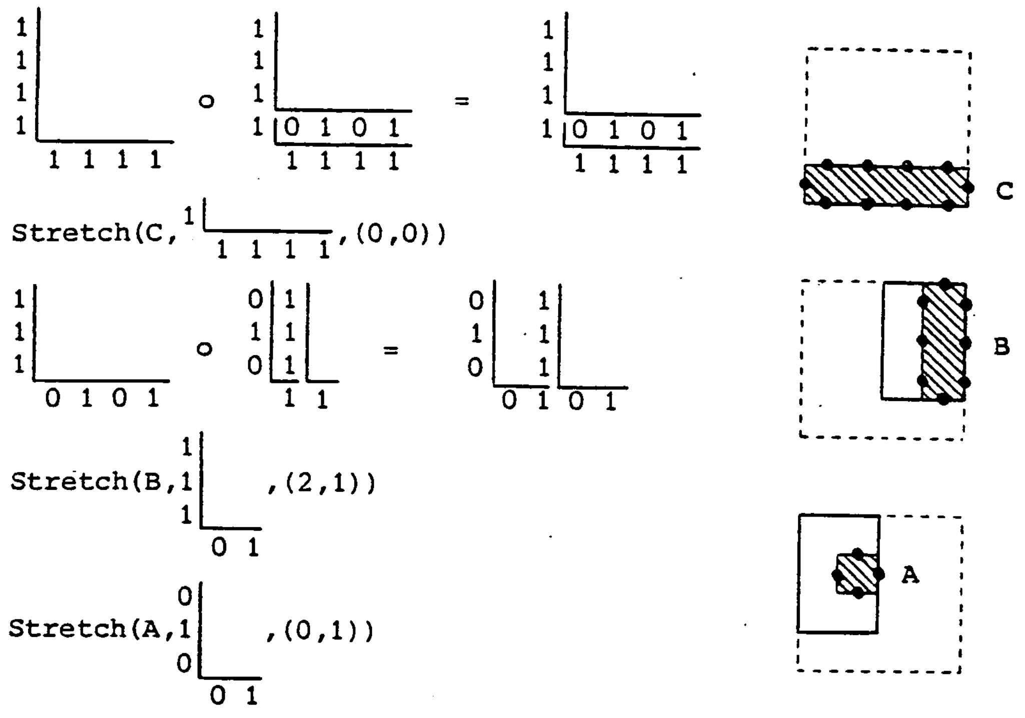


Figure 2.50 Grid recomposition

2.6.6 From Sticks to Grids

It is conceivable to use stick expressions as a general purpose programming notation for stick diagrams and as a target language for silicon compilers. However, in order to use them in this sense it is necessary to develop an algorithmic translation from any stick expression into layouts, and this will be done passing through grids.

The first step consists in arranging the planar graph described by a Stick expression on a rectangular grid. The arrangement of a graph in some particular geometrical or topological space is called a **realisation** of the graph. The choice of a particular grid realisation for a particular stick diagram is purely arbitrary, except that attempts will be made to keep the grid as small as possible.

To limit the number of possible grid realisations for a given expression, we shall also provide a context specifying constraints on the relative position of ports on a grid. We consider rectangular contexts only, where the ports of a grid realisation lie on the perimeter of a rectangle. Moreover every port will be explicitly assigned to one of the four sides of the rectangle, named **south, east, north** and **west** (this assignment should agree with the cyclic ordering of ports).

Given a stick expression e, a context for e is a mapping of <u>some</u> of the ports of σ(e) into a side (south,east,north or west) in such a way that all the east ports cyclically follow the south ports, and so on for the other three sides. A context of this kind on σ(e) is said to be **compatible** with e. A context is also said to be compatible with a grid network g when it lists some of the non-null ports of σ(g) assigning them to the correct side. A full context specifies the sides of <u>all</u> the ports of σ(e).

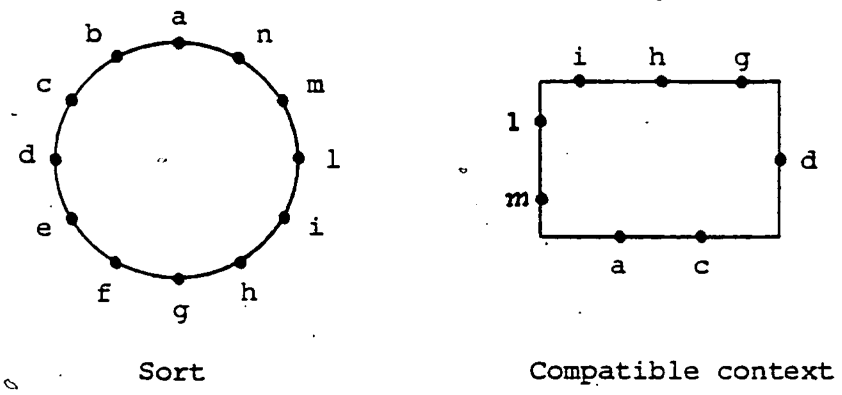


Figure 2.51 A context

The Sticks-to-Grids algorithm takes as input a sorted stick expression (i.e. a stick expression where all the subexpressions are indexed by their sort) and a compatible full context, and produces a grid network which realises the stick expression. The result is supplied in the form of a grid expression where all the compositions are normal. Previous sections have shown how to generate grid layouts from grid expressions. (This form of the output is just for explanatory purposes; we could combine this algorithm with the one in Section 2.6.4, translating directly into stretch trees and grid sorts.)

Basis of the recursion: we assume that for every stick literal and for every compatible context there is a standard interpreted grid network which is also compatible with the context (this standard grid is chosen among all the grids satisfying the requirements). This set of basic grid networks is rather big (there are 140 full contexts for every stick literal) but can be cut down by taking into account similarity and symmetries, and by compromising on the grid area. We shall simply assume here that a grid literal matching a given context is always selected and returned as result for this base case. The next figure shows a set of 35 minimal patterns for a transistor (most of them made of several stick tiles); the missing patterns can be obtained by rotation and by dropping some of the non-null ports (and consequently modifying the tiles).

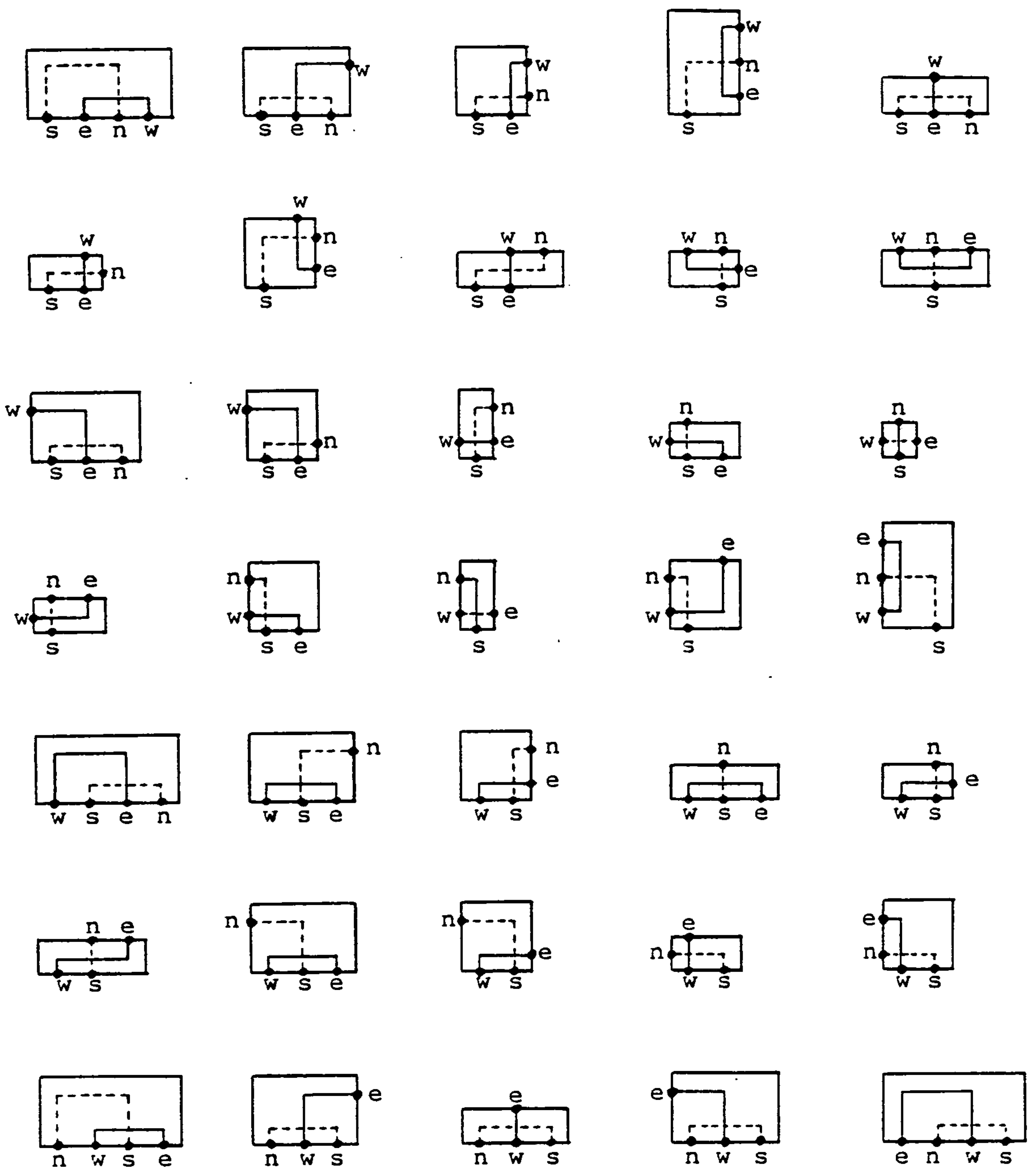


Figure 2.52 Basic grids

The recursive step for a restriction e\a consists in recurring with e and the current context, obtaining a grid expression g.

Because a is not in the current context, this is how partial contexts are generated, even starting from full contexts. At the base of the recursion, the ports not contained in the context are unused (i.e. not connected to anything) and may be optimised away in the grid layout (actually they must, to avoid "accidental" connections). The resulting grid expression is then g\a.

The recursive step for a renaming e{r} consists in applying r^{-1} to the context and recurring with e and the renamed context, obtaining a grid expression g. The result is g{r}.

The recursive step for composition is the interesting one. Given a context and a composite stick expression e'[r]e", the problem is to derive two subcontexts to be applied to the respective subexpressions. This should be done in such a way that the resulting composition is normal, so that we can apply the grid composition technique developed in the previous sections when we come to need a grid back as a result. Let us assume that [r] is $[a_1\text{--}b_1,\ldots,a_n\text{--}b_n]$. We define a pseudo orientation for the context in the following way: a_1 faces pseudo-east, b_1 faces pseudo-west and the pseudo-south side of the context is parallel to their connection (this is always possible):

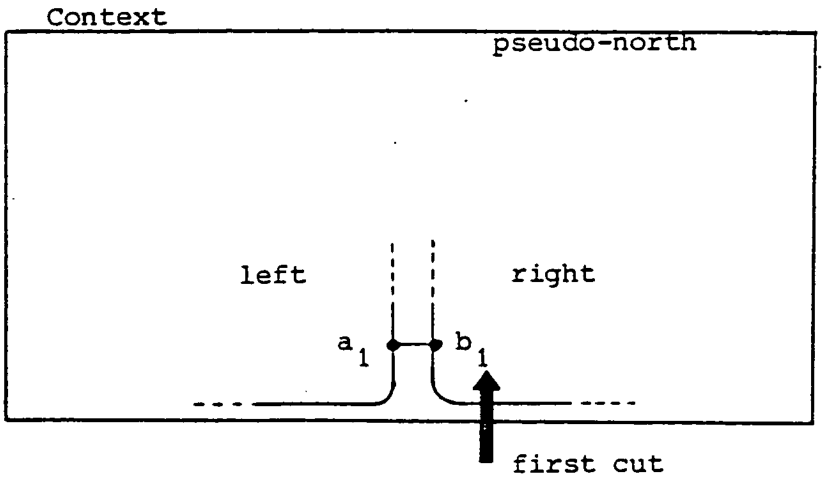


Figure 2.53 Pseudo orientation of a context

We define the first cut as a point on the pseudo-south of the context, which is after b_1 and before any other port (before-after in the anticlockwise sense). The second cut is going to be placed after a_n, and this can fall on any side of the context. Given any placement of first and second cut, we must be able to split the context into two parts and then insert $a_1..a_n$ in one part and $b_1..b_n$ in the other.

The second cut can fall in five substantially different places, and each place corresponds to a different way of splitting the context:

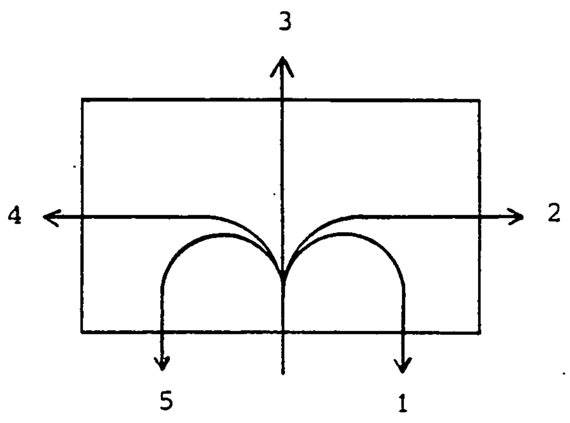


Figure 2.54 The five basic context splits

For each different split, there is a way of arranging the sorts of e' and e" so that they will match the context. Here are five examples of sort fitting in the standard pseudo orientation:

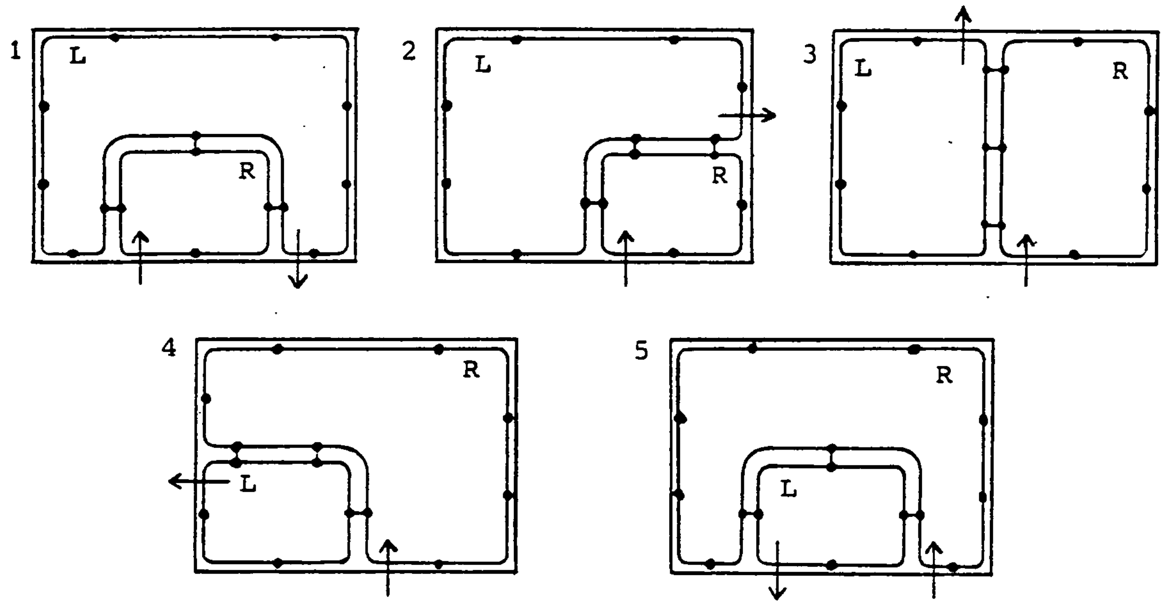


Figure 2.55 Fitting the sorts

Moreover all the sort fitting patterns can be decomposed (in several ways) into normal grid compositions:

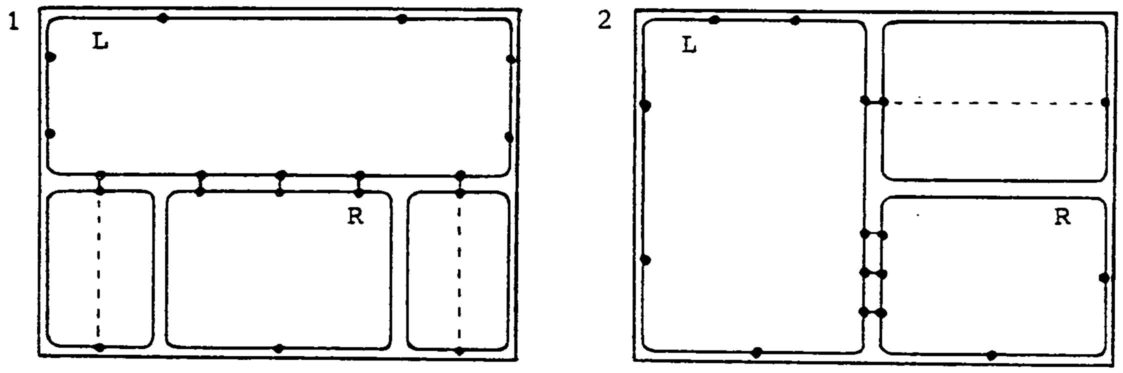


Figure 2.56 Normal decomposition

In the case number 3, all we have to do is to break the context in correspondence of the first and second cut, add the ports $a_1..a_n$

to the pseudo-east of the left context and $b_n..b_1$ to the pseudo-west of the right context, and apply recursively the algorithm on the subexpressions obtaining g' and g". The result is then g'[r]g" which is a normal grid composition.

The process is similar in the other four cases, even if more complicated. More than one grid composition may have to be generated, and dummy grids may have to be inserted. Grid decomposition is not a deterministic process, and heuristics are needed to get better and smaller layouts.

As an example, let us try to fit the stick expression

ETrans[ge--gw](GCon\gn\gs)

in an unfriendly context where all the ports lay on a single side:

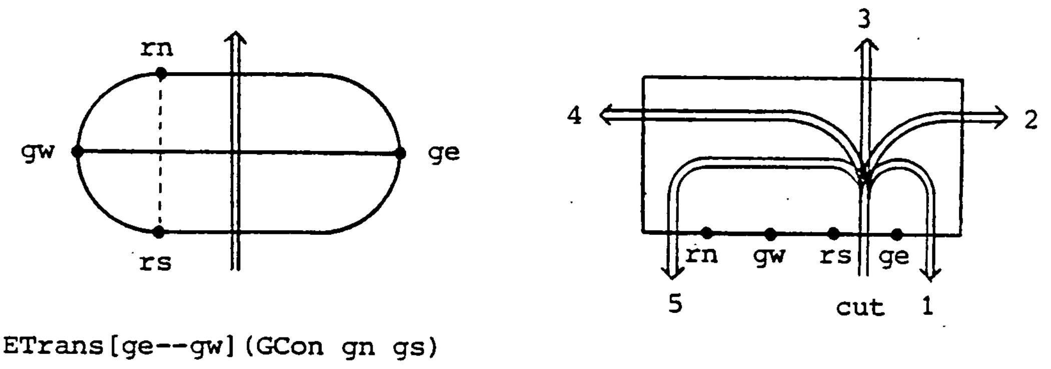
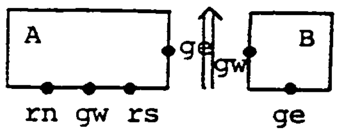
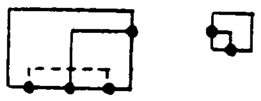
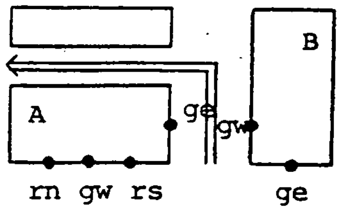


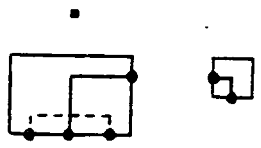
Figure 2.57 Context splits

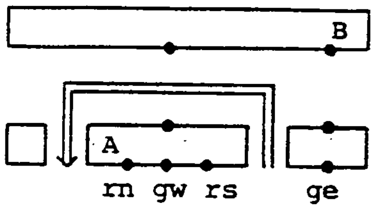
From the composition [ge--gw] we see that the cut must fall between rs and ge on one side, and between ge and rn on the other side. This corresponds to five possible cuts of the context; let us examine them in turn, starting with 3 which is the easiest one. On the left of the next figure are the normal decompositions, and on the right the corresponding (possibly empty) grids:

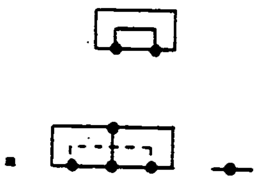
3.


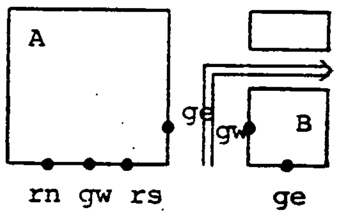


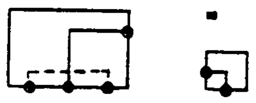
4.


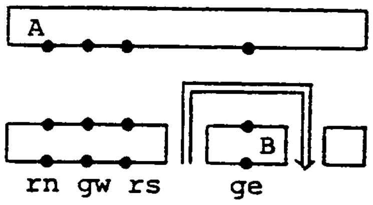


5.




2.




1.


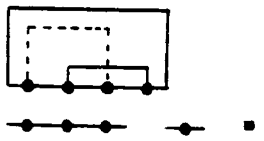
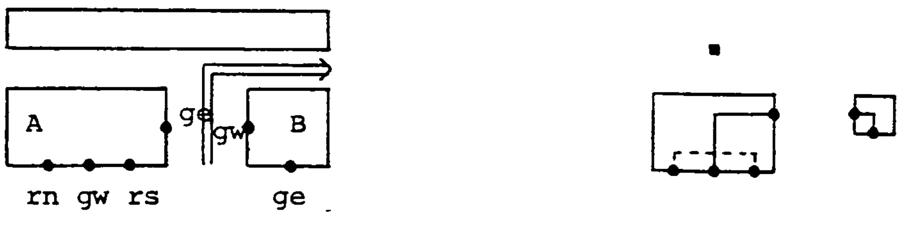


Figure 2.58 Decompositions

There are also two alternative decompositions of 4 and 2:

2'.



4'.

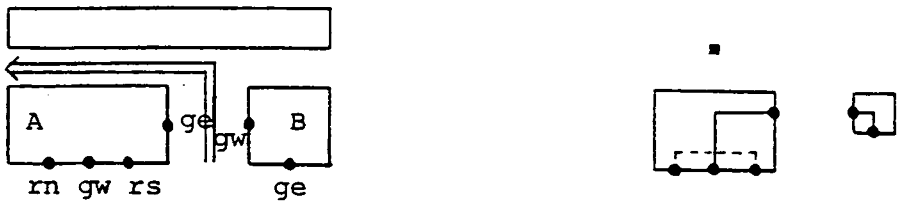


Figure 2.59 Alternative decompositions

We can see that after compositions of the grids on the right, we get two different solutions which correctly fit the context.

Note that not all of the five decompositions can be used in any case. A very simple minded heuristics for getting good results is to use decomposition 3 whenever possible, otherwise 1 or 5 (because they do not have alternative decompositions like 2 and 4), and then 2 or 2' or else 4 or 4'. The choice between 2 and 2' or between 4 and 4' can critically influence the size of the result, because these decompositions introduce empty areas.

Let us recall the phases of the Sticks-to-Grids algorithm:

1. From sorted stick expressions and contexts to grid expressions.
2. From grid expressions to stretch trees and grid sorts.
3. From stretch trees and grid sorts to grid layouts.

where phases 1 and 2 can be combined into a single phase.

In phase 1, every stick composition e'[r]e" is translated into the composition of two grid expression g',g" (corresponding to e' and e") possibly augmented by a limited number of padding literals, depending on the form and number of ports of the context. In phase 2 information is accumulated in a stretch tree in order to perform the stretching of grids. In phase 3 the various partial stretching are accumulated and the resulting stretched grid is produced.

The correctness of these translations can be expressed as the commutativity of the following diagram:

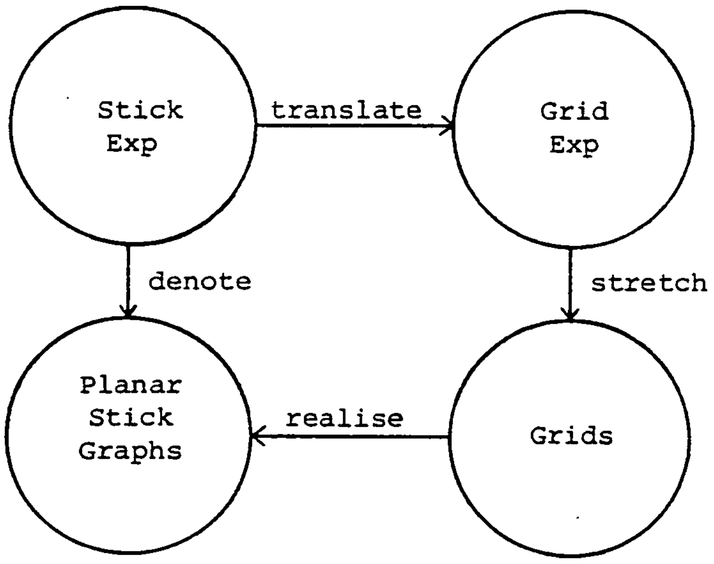


Figure 2.60 Correctness of translations

Grids realise planar stick diagrams (note that there are several grids for the same stick diagram). The translations are correct if the stick diagram denoted by a stick expression is realised by the grid produced by the Stick-to-Grid algorithm on that stick expression.

The grid composition algorithm could be improved to include a limited amount of routing, in order to avoid explosive stretching situations. Moreover, iteration is probably going to be a primitive control construct in stick expressions, and we can use this fact to improve the form of the layout and to avoid "diagonal fugues" (i.e. situations in which the cells of an array get incrementally stretched).

The solution we have adopted for the base case (namely considering all the possible grids for a given stick literal and context) is not feasible for bigger literals which could arise in practice, like standard cells and functional units. In those cases some routing must be used to connect the cells to their expected context; an interesting question is whether the decomposition process can be driven so that the matching of standard cells and contexts is made easy.

The most important operation in phase 2 is grid composition, i.e. the computation of stretch vectors. In the worst case, the area of the result grid is twice the sum of the areas of the component grids, and the stretch vectors are as long as the sum of the sides.

Every composition node of a stretch tree contains two boolean vectors of the same length as the result of the composition (the other two vectors described in the algorithm are identically "1"'s and do not need to be represented explicitly). Let us assume that all the grid literals have size 4x4, (they are always smaller); if g' and g" have size <z,z>, then in the worst case the stretch tree of g'[r]g" has size 4z (the two stretch vectors) plus the size of the trees for g' and g". For a balanced tree giving rise to a square diagonal layout, this makes a total of 8(n*log n) bits of stretch vectors and the resulting layout has area less than $(4n)^2$. Even in this pessimistic situation, the stretch vectors for a grid expression with 1,000,000 literals would occupy some 19 megabytes, still in the range of current virtual memories. The construction of the grid sort takes another O(n*log n) space, but all this storage (except for the final result) can be reclaimed during the process.

In practice the stretching algorithm is expected to behave in a slightly better way, especially in case of structured design styles. In the best "square" case (a balanced square composition of 1x1 literals with no stretching) the size of the tree is n*log n and the layout area is n. Hence, in the above example, the best square case needs 2.4 megabytes.

This complexity analysis can however be rather pessimistic in real situations, because of the hierarchical nature of our approach. Stick expressions are very likely to contain a considerable amount of sharing (e.g. register arrays and ALU's) and the shared parts are very likely to get identically stretched. If we preserve this sharing (for what is possible) during the translation to grid expressions and the generation of stretch trees, a considerable amount of space can be saved; this can be done at the expense of checking for the occurrence of already processed subexpressions and contexts. Then, for example, the storage occupation of stretch vectors for regular arrays of cells becomes constant.

Other common parametric structures which are not likely to contain sharing (e.g. PLA's and ROM's) have predetermined size and do not need to be stretch-analysed. They can be conveniently introduced as primitives at the stick expression level.

Phase 3 requires another O(n*log n) space to compute the stretch vectors of the grid literals, but the stretch tree can be demolished in the process so that little extra memory should be needed.

The time complexity can vary from linear in the number n of grid literals (with n^2 space occupation) to exponential (with optimal space occupation). A satisfactory compromise should be achieved by using heuristics, or (failing those) by direct user interaction.

2.6.7 From Grids to Layouts

And here is the final step in our long road towards layouts. Some forms of translation from grid structures into layouts are already well known: in general the sticks are first inflated into lines and transistors of the appropriate size; then the result is compacted in order to achieve low area occupation.

Mosteller describes a compaction algorithm which is at the basis of an interactive editing system for sticks [Mosteller 81]. The great advantage of orthogonal grids is that the compaction can be carried out independently on the x and y axis, achieving good results.

Expansible grids also constitute the main data structure at the basis of the remarkable VLSI workstation by Weste and Ackland [Weste 81]. Their system retains geometrical information (like transistor sizes) and compaction is used to optimise screen-drawn layouts and after cell composition.

Both the approaches mentioned above allow the user to interactively modify the default sizes of wires and transistors, providing the same freedom as in hand-drawn layouts. On the other hand these translations are not completely automatic, or at least do not always lead to perfect layouts if used in an automatic way. This is not a criticism of the above systems, which address different issues, but an indication that further work is needed, especially in the generation of electrical parameters from stick structures.

3. Sticks & Stones

In this chapter we describe a design language for VLSI, based on the ideas presented in the first two chapters. The language works at the geometrical layout level, constituting what is generally called a "chip assembler", and produces output in a standard format suitable for nMOS processing.

The language has been implemented as an experimental interactive system which uses a colour graphics display for the preparation of VLSI layouts. The examples shown here have been produced by this system and drawn on a 4-colour flat-bed plotter (and then shaded).

This chapter can be read independently from chapters 1 and 2, giving a self-contained description of the design system. As a consequence some information concerning net algebras is repeated, also to emphasise the occasional differences in style and semantics due to practical implementation issues.

3.1 Introduction

The most important attribute of a flexible design language for VLSI is perhaps its ability to parameterise any possible aspect of a picture, such as its size, the number and type of components and the distance between them. This suggests that the language should be primarily text oriented but with graphic facilities; then parameterisation can be easily achieved by using the parameter passing mechanism of procedures. On the other hand, a display oriented language has severe problems with parameterisation: it is very easy to assemble figures on a screen with a pointing device, but it is difficult to express how these figures are actually meant to change as a function of some parameters.

Now purely textual languages for graphics suffer from severe drawbacks as the identification of text and image can be very difficult. Any such language should therefore be highly interactive with immediate visual feedback, and the syntax should recall as far as possible the structure of the picture, i.e. its topological properties. This is in sharp contrast for example with graphic packages, in their use as extensions to existing host languages.

The kind of language we are interested in should be able to express VLSI circuits naturally in terms of their hierarchical structure and their topological properties [Buchanan 80, Rowson 80, Williams 78] and the structure of the circuits should appear through the text of the descriptions.

In Sticks&Stones, pictures are handled just like an abstract data type within a general purpose programming language, so that every picture is denoted by a program which builds it. The operations over pictures are inspired by net algebras, whose expressiveness and algebraic properties have been studied in the first two chapters. These operations are topological in nature and give rise to programs which are suggestive of the pictures they represent. Pictures are embedded in an applicative higher-order language, which is based on a subset of Edinburgh ML [Gordon 79a]. The control structures of the language can be very easily used to define arbitrary parameterisations and conditional assemblies of pictures.

The language is applicative in two of the senses commonly attributed to this word; it is expression oriented and free from side-effects. Expressions seem to be more suited than statements to an interactive language. They improve and enforce the structured description of complex pictures and help in keeping information local. Every picture is taken to be an unmodifiable and unbreakable object, which can only be used to make larger pictures, and which can only be manipulated through its set of named ports. Picture composition is then done by port names (and not by geometrical position or displacement) with automatic translations and rotations.

There are many advantages in manipulating pictures by their ports only. For example, the order in which the pictures have been put together becomes irrelevant (as there is no way to access the inside of the picture) and programs are guaranteed not to rely on irrelevant structural details. Moreover, the orientation and scaling of pictures are unimportant, and the system can automatically rotate pictures and adjust them to fit the screen.

Side effects might be needed to edit a picture, but we regard this problem as completely distinct from that of picture construction. Editing a picture is also very different from editing a text or a tree, as in the former case there may be very troublesome context dependent effects, like those resulting from increasing the size of a subcomponent. In this context, editing by rebuilding can be much more convenient than editing by modifying, especially if an adequate structure of program modules is provided.

If side effects are forbidden, a "correctness by construction" approach can be applied. We might be able to show that a picture enjoys some property P (e.g. absence of geometric rule violations) if its basic components have the property P and if the picture operations preserve the property P. Thus, the amount of checking to be done when composing two pictures can be drastically reduced. In the implementation of this system we decided to concentrate on different issues, and we did not incorporate hierarchical checks (such as hierarchical design rule checking [Withney 81]), which however seem to fit particularly well in this framework.

3.2 Pictures

We now describe how pictures can be generated. A picture is either an elementary picture (called a **form**) or the composition of smaller pictures. Pictures form an abstract data type and are first-class objects in the language.

3.2.1 Forms

A form is made of a set of figures (boxes, polygons, etc.) with a sort. The sort of a picture is a list of ports, and ports are used to connect pictures together.

```
- let bluesquare =
      form(b.S : W port [0↑0,0,1];
           b.E : W port [1↑0,90,1];
           b.N : W port [1↑1,180,1];
           b.W : W port [0↑1,270,1])
      with B box [0↑0,1↑1];

  bluesquare = <> : (b.S:W; b.E:W; b.N:W; b.W:W) : [1,1]
```

A phrase like "let bluesquare = ... ;" is used to define the variable "bluesquare" at the top level (the string "- " preceding it, is the Sticks&Stones prompt). The answer from the system is "bluesquare = ——", where "——" is the result of the evaluation of "...". In this case the result is a "<>" (i.e. a picture whose structural details have been omitted) of sort "(...)" and of size 1,1 which is the size of the minimum enclosing rectangle.

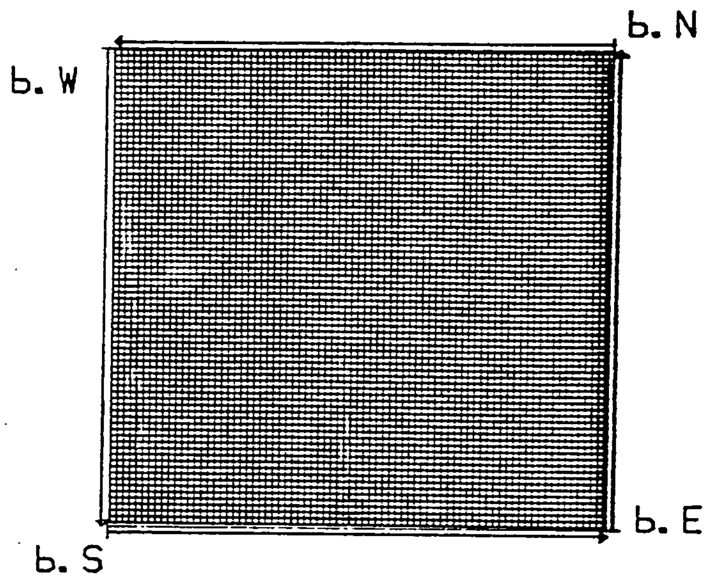


Figure 3.1 A blue square

The figure bluesquare is a form (an elementary picture) made of a single B (blue) box with lower left corner at the point 0↑0, and upper right corner at the point 1↑1. It has four ports "b.S", "b.E", "b.N" and "b.W".

A port name can be any list of identifiers and numbers (starting with an identifier) separated by dots, like "a" or "aaa.bbb.1.c'.3"; these identifiers and numbers are called **atomic parts** of a **compound port name**. Port names have no semantic significance, but they will often suggest the function of their associated port (e.g. "b.E" will stand for "blue East").

The port "b.S" is a W (white) port; geometrically this is the vector with tail at 0↑0 oriented 0 degrees anticlockwise from the x axis and of length 1 (hence its tip is at 1↑0). The north of a vector is by convention in the tail-to-tip direction.

A more complete example is provided by an nMOS inverter:

```
- let inverter =
   form (b.E:B port [5↑5,90,4];
         b.W:B port [1↑9,270,4];
         g.S:G port [2↑0,0,2];
         r.E:R port [6↑1,90,2];
         g.E:G port [6↑4,90,2];
         r'.E:R port [6↑7,90,2];
         g.N:G port [4↑15,180,2];
         r.W:R port [0↑3,270,2])
   with B box [1↑4,5↑10]
   and  G box [0↑0,6↑8; 2↑8,4↑15]
   and  R box [0↑7,6↑15; 0↑1,6↑3]
   and  Y box [0.5↑5.5,5.5↑16.5]
   and  C box [2↑5,4↑9];

inverter = <> : (b.E:B; b.W:B; g.S:G; r.E:R; g.E:G;
                r'.E:R; g.N:G; r.W:R) : [6,16.5]
```

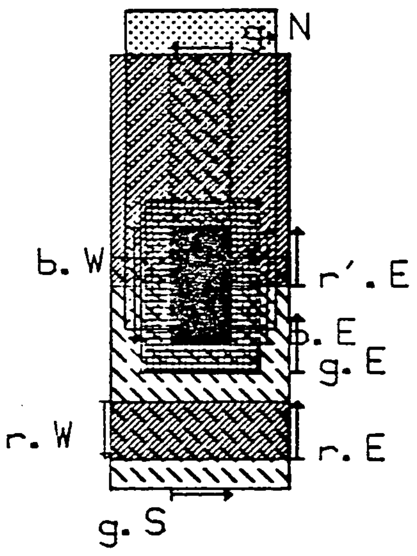


Figure 3.2 An nMOS inverter

Ports of type B (blue) G (green) and R (red) are drawn in the corresponding colour. Ports of any other type are also admitted, and are drawn in the foreground colour (depending on the graphical device).

Boxes can have colours B (blue) G (green) R (red) Y (yellow) C (black) or W (white), and they may overlap; other colours are syntactically admitted but are all drawn in the foreground colour. Note that a list of rectangles can be specified after the keyword "box".

Ports should always be oriented anticlockwise around a picture. This is not mandatory, but picture composition is made connecting ports on their east sides (tail to tip and tip to tail), and the anticlockwise convention ensures that pictures are joined on their outer sides. A picture may have no ports and/or no figures. The empty picture is simply:

```
- form;

<> : () : [0,0]
```

3.2.2 Restriction

Restriction is used to forget about some of the ports of a picture; the syntax is: expression, followed by "\", followed by a list of port names:

```
- inverter \ b.W  !.E  g.?;

<> : (r.W:R) : [6,16.5]
```

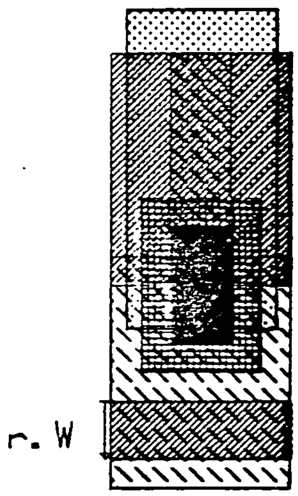


Figure 3.3 Restriction

Question marks and exclamation marks are used to pattern match port names. Any variable beginning with an exclamation mark (like "!"; "!!", "!abc" or "!3") matches with a single atomic part of a compound port name, while any variable beginning with a question mark matches with an arbitrary number (zero included) of atomic parts.

In the example above we withdraw the port b.W, all the E(ast) ports and all the g(reen) ports from the inverter. The inverter itself is not affected by this operation and a truly new picture is generated.

3.2.3 Renaming

The renaming operation performs a simultaneous substitution over the ports of a picture; the syntax is: expression, followed by "{", followed by a list of single renamings separated by ";", followed by "}". A single renaming "a\b" means "a becomes b".

```
- inverter {r'.E\inv.r'.E; !.W\inv.!.W};

<> : (b.E:B; inv.b.W:B; g.S:G; r.E:R; g.E:G; inv.r'.E:R;
      g.N:G; inv.r.W:R) : [6,16.5]
```

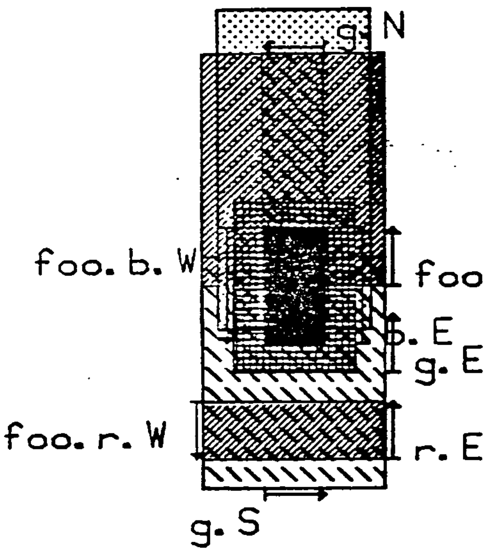


Figure 3.4 Renaming

Match variables instantiated in the left part of a substitution can be used in the right part to get group renamings like "!.W\inv.!.W" which is an abbreviation for "b.W\inv.b.W; r.W\inv.r.W". Note that "!.!" matches "a.a" but does not match "a.b", which is matched by "!.!!", "!.?", "!.!!.?" or "?", but not by "!" or "!.!!.!!!". You can go as far as "!.!.!!.? \ !!.!.?.?.!!", which renames "a.a.b.3.5" into "b.a.3.5.3.5.b". A question mark in the left hand side can only appear as the last atomic part, otherwise the matching might be ambiguous. A matching variable in the right hand side which does not appear in the left hand side is illegal.

3.2.4 Composition

Having two pictures, we can compose them by port names; the syntax is: expression, "[:", list of single links separated by ";", ":]", expression. A single link has the form: portname, "—", portname.

```
- redsquare [: r.E — g.W :] greensquare;

<> : (r.S:W; g.S:W; g.E:W; g.N:W; r.N:W; r.W:W) : [2,1]
```

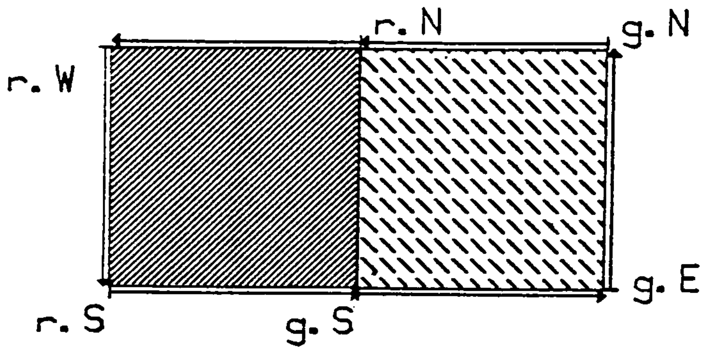


Figure 3.5 Composition

where redsquare and greensquare are defined similarly to bluesquare. This composition produces two adjacent squares, where the ports r.E of redsquare and g.W of greensquare have been connected and forgotten.

Several links can be specified inside the composition brackets, separating them by semicolons. All the ports involved in a connection are forgotten in the result, whose sort is otherwise the union of the sorts of the composing pictures. Pattern matching is not allowed in composition; programming experience has shown that its use leads to unclear programs.

Composition is a symmetric operation (in the sense: $P[:p_i{-}{-}q_i:]Q = Q[:q_i{-}{-}p_i:]P$), and as an infix operator associates to the left. Every pair of ports which are being linked in a composition must have the same type and the same size. Composition with the empty picture by any pair of ports leaves a picture unchanged.

Connection of two ports is made tail to tip and tip to tail with no distance between them. In case of connection of several pairs of ports, the main link is connected first, and all the other pairs of ports must face each other, maybe with a gap in the middle. The main link is defined as the first link on the left, inside the composition brackets.

3.3 Bunching

Every port is actually a bunch, or collection of collinear vectors. Up to now we only considered single-vector ports, but a port can also be a list of vectors:

```
R port [0↑0,0,1; 2↑0,0,1; 5↑0,0,1]
```

Every vector in a bunch must have the same type, orientation and size and they must be collinear, but they can be differently spaced. Bunches may also be interleaved. When two ports are composed, every vector in one port must match with a corresponding vector in the other port.

Bunches usually arise from composition: when two pictures are composed, the ports with equal names which are not being linked get bunched together:

```
- bluesquare[:b.E—b.W:]bluesquare;

<> : (b.S:B; b.E:B; b.N:B; b.W:B) : [2,1]
```

here b.S and b.N are two bunches of two ports, which are drawn as a single arrow. Again bunching only succeeds for collinear ports of the same size; otherwise an error is reported.

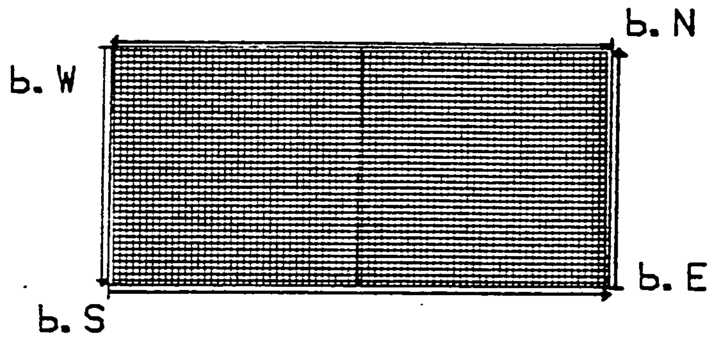


Figure 3.6 Bunching

Bunches allow one to compose regular arrays of pictures without having to explicitly index the ports of every picture in the array by renaming them. They thereby keep the total number of ports in a picture low, making composition simpler and more efficient.

3.4 Iteration

Iteration is used to make regular arrays of cells, as in:

```
- 3 times bluesquare with [:b.E—b.W:];

<> : (b.S:W; b.E:W; b.N:W; b.W:W) : [3,1]
```

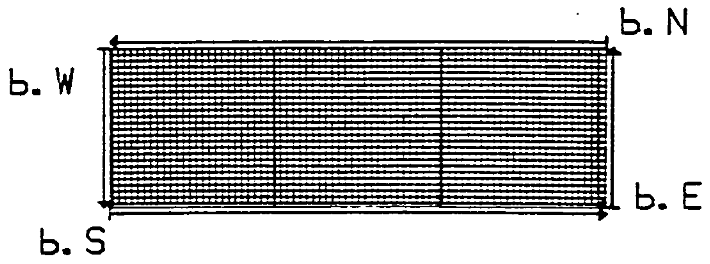


Figure 3.7 "times" iteration

which is equivalent to:

```
- bluesquare [:b.E—b.W:]
  bluesquare [:b.E—b.W:]
  bluesquare;


<> : (b.S:W; b.E:W; b.N:W; b.W:W): [3,1]
```

Iteration is equivalent to the obvious recursive program one might write in the language, but is more efficient and syntactically clearer. Iteration often produces bunches, as in the example above.

Iteration variables are admitted in the "for" form of iteration:

```
- let blue  = bluesquare{b.?\?}
  and red   = redsquare{r.?\?}
  and green = greensquare{g.?\?};

blue = <> : (S:W; E:W; N:W; W:W) : [1,1]
red = <> : (S:W; E:W; N:W; W:W) : [1,1]
green = <> : (S:W; E:W; N:W; W:W) : [1,1]

- for square in [blue; red; green]
  iter square
  with [:E—W:];

<> : (S:W; E:W; N:W; W:W) : [3,1]
```

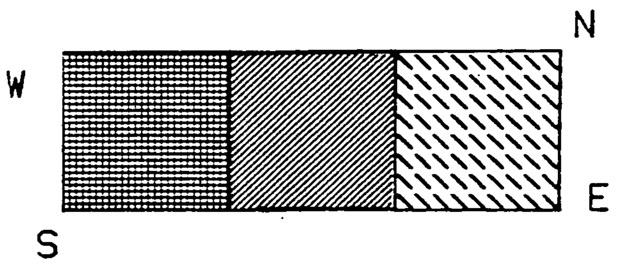


Figure 3.8 "for" iteration

which produces the above picture. The iteration variable "square" takes in turn the values blue, red, green in the list.

Double iteration can be used to produce arrays of pictures:

```
- let squares array =
      for row in array
      iter for item in row
           iter item
           with [:E—W:]
      with [:S—N:];

squares = ""
```

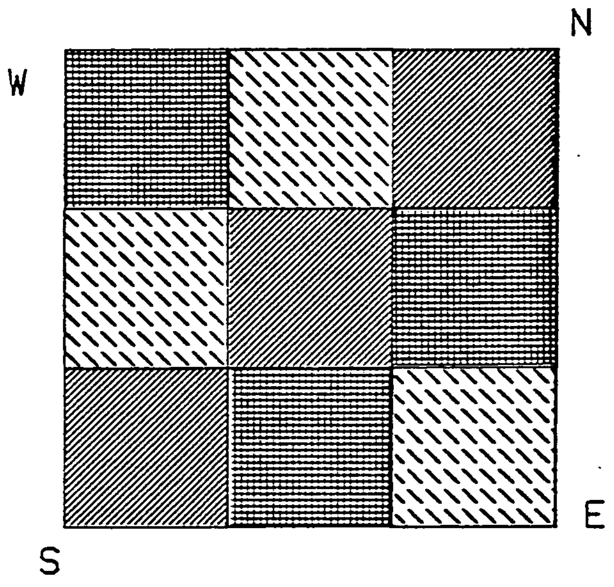


Figure 3.9 Double iteration

(where '""' means that 'squares' is a function). This is the definition of a parametric picture, that is a function taking a list of lists (i.e. an array) of pictures and producing a picture. It can be used as follows:

```
- squares [[blue;  green; red  ];
           [green; red;   blue ];
           [red;   blue;  green]];

<> : (S:W; E:W; N:W; W:W) : [3,3]
```

Sometimes it is useful to iterate through several lists at once; this feature is used in the following definition of "squares'" which substitutes a green column every three input columns:

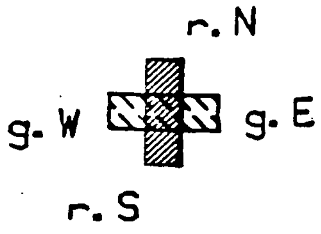
```
- let squares' array =
    for row in array
    iter for item in row and i in 1::length row
         iter (i mod 3)=0 ⇒ green | item
         with [:E—W:]
     with [:S—N:];

squares' = ""
```

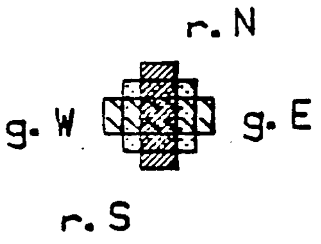
where the operation "n::m" produces the list of all numbers from n to m, and "a ⇒ b | c" means "if a then b else c".

A selector is a realistic example of a parametric picture with which can be built by double iteration. We first need to define three basic building blocks: "pos" (an enhancement transistor), "neg" (a depletion transistor) and "out" (a piece of the common output):

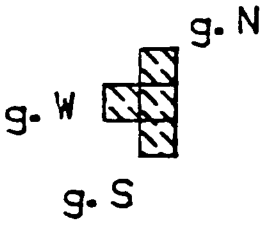
```
- let pos =
    (form (r.S:R port [2↑0,0,2];
           g.E:G port [6↑2,90,2];
           r.N:R port [4↑6,180,2];
           g.W:G port [0↑4,270,2])
     with R box [2↑0,4↑6]
     and  G box [0↑2,6↑4])
```



```
  and neg =
    (form (r.S:R port [2↑0,0,2];
           g.E:G port [6↑2,90,2];
           r.N:R port [4↑6,180,2];
           g.W:G port [0↑4,270,2])
     with R box [2↑0,4↑6]
     and  G box [0↑2,6↑4]
     and  Y box [0.5↑0.5,5.5↑5.5])
```



```
  and out =
    (form (g.S:G port [2↑0,0,2];
           g.N:G port [4↑6,180,2];
           g.W:G port [0↑4,270,2])
     with G box [2↑0,4↑6; 0↑2,2↑4]);
```



We now need to put these pieces together: the following program takes a number n and produces a selector with n control inputs (the n-bunch "r.N"), n complemented control inputs (the interleaved n-bunch "r'.N"), 2^n input lines (the 2^n-bunch "g.W"), one output line (the 1-bunch "g.N") and the appropriate pattern of enhancement and depletion transistors (produced by the auxiliary function "bit").

```
- let sel n =
    for i in 1::exp(2,n)
    iter (for j in n::1
          iter bit(i-1,j-1)=0 ⇒
                 pos [:g.E—g.W:] (neg{r.?\r'.?}) |
                 neg [:g.E—g.W:] (pos{r.?\r'.?})
          with [:g.E—g.W:])
         [:g.E—g.W:] out
    with [:r.S—r.N; r'.S—r'.N; g.S—g.N:]

  whererec bit(i,j) =
    j=0 ⇒ i mod 2 | bit(i//2,j-1);
```

here "exp" is exponentiation and "//" is integer division.

The circuit shown in the next figure is the result of the evaluation of sel 2 (selector with two control inputs). The selector is obtained by two nested iterations, first building the rows and then joining them up into an array. At the core of the double loop we have to choose between a pair of pos-neg' and a pair of neg-pos' (where pos' and neg' are pos and neg with their r ports renamed to r'); this is done using a function "bit". The inner loop connects all these pairs into a row, with the variable j ranging from n to 1. At the end of the inner loop, an out element is added to the right of the row. In the outer loop the variable i ranges from 1 to 2^n while all the rows are connected from south to north by bunch connections.

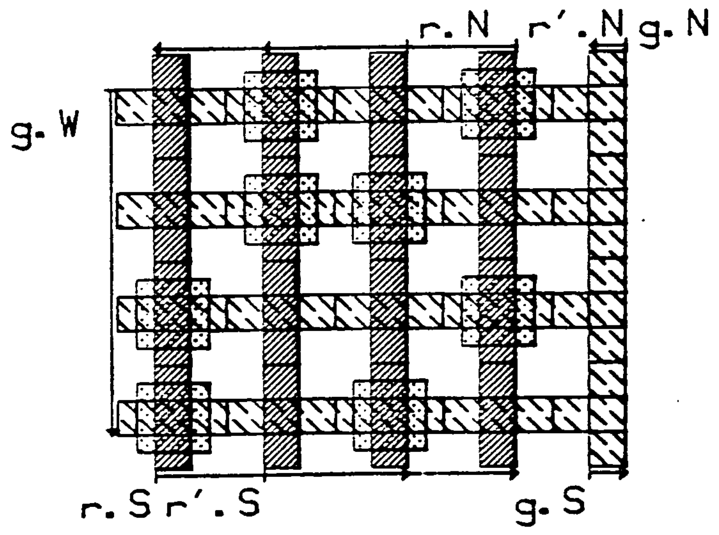


Figure 3.10 A selector

It should be emphasised that the selector program contains no explicit geometric information, and this is to be expected for many common VLSI subsystems. The double loop (array) pattern is also very common in structured design, and many other interesting examples can be produced by the use of parameterisation and recursion.

3.5 Paths and Geometric Renaming

A **path** can be generated by taking a port and moving it around: the wake of the port forms the resulting path. The outcome of this operation is a list of polygons (one or more for every step the port has made) and a new port (i.e. the old port in the new position). Hence a path is the following data type:

path = (polygon list) x port

Given a path the following operations extend it from the port, thereby generating a new path:

```
stay: path -> path
move: num -> path -> path
step: num -> path -> path
rotl: num -> path -> path
rotr: num -> path -> path

move': num -> path -> path
step': num -> path -> path
rotl': num -> path -> path
rotr': num -> path -> path
```

The operation stay leaves a path unchanged;

The operation move takes a positive number n, a path p and moves the port of the path n units. The direction of movement is towards the east of the port (i.e. generally outwards with respect to the picture if anticlockwise ports are used). The new path generated is made of the new port and the old polygon list with a new rectangular polygon having the old and new ports as edges.

The operation step is like move, but "step n" means "move n times the size of the port" for simple ports, and "move n times the size of the vectors in the port" for bunches.

The operation rotl (rotate left) takes a number n (in degrees), a path p and rotates the port of the path n degrees anticlockwise describing a circular arc with centre in the tip of the port. If the port is a bunch, the distances between the vectors are respected and the result is a set of concentric paths. The new path generated is made of the new port (or bunch) with the old polygon list plus the new polygon(s) generated by the rotation.

The operation rotr (rotate right) is the same as rotl, but the rotation is clockwise and its centre is in the tail of the port.

The operations move', step', rotl' and rotr' are similar to their unprimed versions, but they move a port without producing any path between the old and new position. The operations move' and step' also accept negative arguments.

Functions from paths to paths are called **path functions**; the following are path functions:

```
stay
move 2
step 5
rotl 90
rotr 270
```

Function composition is used to compose path functions; in particular it is convenient to use the inverse function composition operator "&":

```
(f & g) x = g(f x)
```

Here is an example of a composite path function:

```
move 2 & rotl 90 & step 4 & rotr 90 & move 2
```

note that "&" behaves like an append on paths, as function composition is associative.

How do we use path functions? Ports are not available to the user as data objects separated from pictures, so that path objects can never be built, and there is nothing to apply path functions to. The only place where is possible to use path functions is in the geometric renaming feature of the renaming operation:

```
- bluesquare {?\? move 2};

<> : (b.S:W; b.E:W; b.N:W; b.W:W) : [5,5]
```

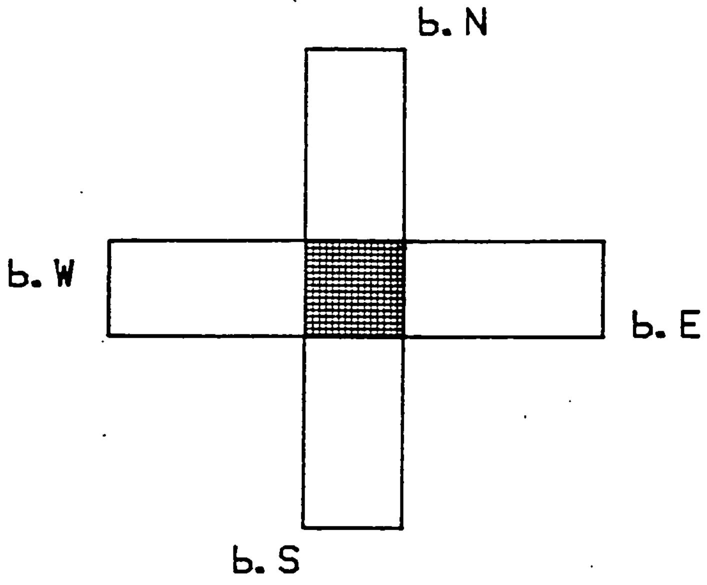


Figure 3.11 A blue cross

The meaning of this is to rename every port in bluesquare by its own name, moving it 2 units outwards. The result is a blue cross of size [5,5]. The path function "move 2" is applied in turn to the paths obtained pairing the ports of bluesquare with the empty list of polygons.

Here is a very flexible blue square which can be stretched symmetrically in four directions by applying a path to it:

```
- let bluewheel path = bluesquare {?\? path};

bluewheel = ""
```

```
- bluewheel (move 2 & rotl 45 & move 15 & rotr 135 &
             move 30 & rotr 45 & move 20 & rotr 270);

<> : (b.S:W; b.E:W; b.N:W; b.W:W) : [68.9,68.9]
```

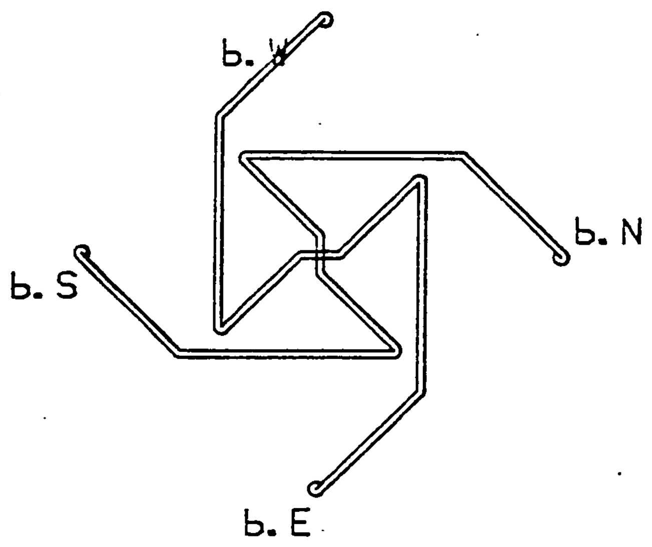


Figure 3.12 Geometric renaming

A limited form of routing (called river-routing) can be obtained by using geometric renaming on bunches:

```
- sel 2 {g.W \ g.W rotr 60 & rotl 60 & move 6} \ ?;

<> : () : [51.32,32]
```

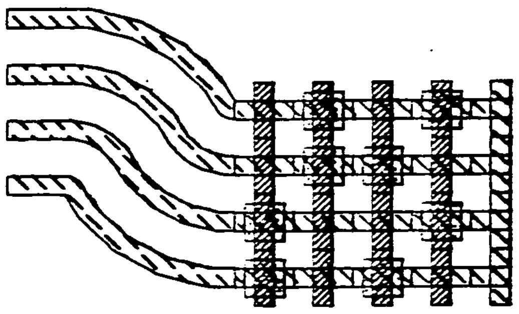


Figure 3.13 Geometric bunch renaming

3.6 Figures

There is a variety of elementary figures. Actually many of them have no application in VLSI and are intended mainly for graphics. All of the following options can appear syntactically after the keyword "with" inside forms (in the place of boxes in the examples of the previous section).

dot [p1; ... ;pk] draws dots at the specified points p1 ... pk.

line [l1; ... ;lk] draws a set of lines l1 ... lk; every line is a list of points li=[p1; ... ;pki] which are joined by straight segments.

path [l1; ... ;lk] draws a set of paths l1 ... lk; every path is a list of pairs of numbers and points li=[n1,p1; ... ;nki,pki]. Adjacent points p(j),p(j+1) in a path are joined by a circular arc of aperture n(j+1) degrees (if n(j+1) is 0 or any multiple of 360, a straight segment is used). If n(j+1) is positive, the arc is convex on the east of the vector p(j)⟶p(j+1); if negative it is convex on the west. The first aperture n1 is not used.

spline [l1; ... ;lk] draws a set of non periodic cubic B-splines l1 ... lk; every spline is built from a list of control points li=[p1; ... ;pki]. The spline does not pass through the control points (except the first and the last), but is tangent to every segment joining two adjacent control points.

loop [l1; ... ;lk] draws a set of periodic cubic B-splines l1 ... lk; every spline is built from a list of control points li=[p1; ... ;pki]. The spline is tangent to every segment joining two adjacent control points (the last point is adjacent to the first) and describes a closed curve.

box [p1,q1; ... ;pk,qk] draws a set of boxes with lower left corner at the point pi and upper right corner at the point qi.

poly [l1; ... ;lk] draws a set of polygons l1 ... lk; every polygon has a line li=[p1; ... ;pki] as perimeter. The last point pki is joined back to the first.

area [l1; ... ;lk] draws a set of areas l1 ... lk; every area has a path li=[n1,p1; ... ;nki,pki] as perimeter, where the first aperture n1 is used to join the last point back to the first.

blob [l1; ... ;lk] draws a set of blobs l1 ... lk; every blob has a loop li=[p1; ... ;pki] as perimeter.

text [p1,s1; ... ;pk,sk] draws a set of character strings s1 ... sk starting respectively at the points p1 ... pk. Every string may contain control information (following the escape character "%") according to this code: "%r" change colour to red; "%g" change colour to green; "%b" change colour to blue; "%y" change colour to yellow; "%B" change colour to background (black for Charles, white for HP plotter etc.); "%F" change colour to foreground (white for Charles, black for HP plotter etc.); "%0" ... "%9" change text size (0=min, 9=max); "%S" halt plotting and wait for a carriage return to continue (e.g. to change page on the HP plotter); "%x" for any other character "x" to actually display "x" (e.g. "%%"). Note that the escape character "%" is only interpreted by the plotting routines while the normal escape character "/" should be used for any other purpose (e.g. to insert a "`").

3.7 Commands

The following commands are accepted at the top level.

mode: this command investigates the state of the environment, showing what options are active and what are not. Options are:

print: when active, the result of every top-level evaluation is printed at the terminal.

charles: when active, the result of every top-level evaluation is drawn on a Charles colour graphic terminal.

tektronix: when active, the result of every top-level evaluation is shown on a Tektronix terminal.

hpplot: when active, the result of every top-level evaluation is plotted on a HP-7221A plotter.

drawnames: when a plotting device is active, draws the names of the ports at their location.

drawports: when a plotting device is active, draws the ports at their location as little arrows.

signature: when a plotting device is active, puts a signature "Sticks&Stones" in the lower right corner.

page: when a plotting device is active, plots in "page" mode. Every picture shown will fit incrementally the available space from top to bottom (it will try to make pictures horizontally as large as possible). On the HP plotter, pictures will fit an A4 sheet of paper.

logfile: produces a log file "STICKS.LOG" containing a transcript of the terminal input. Type "addmode logfile" to open a new logfile (destroying the old one) and start writing on it, and "submode logfile" to save it and stop writing on it.

addmode m1, ... ,mn: adds the modes mi to the current mode.

submode m1, ... ,mn: subtracts the modes mi from the current mode.

print v: prints the object v; all the plotting actions are suppressed for the duration of this command.

draw v: draws the object v on the currently active device(s). Print is suppressed for the duration of this command. If v is a picture, it is plotted. If v is a list of n items, the screen is horizontally divided into n viewports, and every item in the list is drawn in a viewport; if an item in v is again a list, its viewport is divided vertically, and so on horizontally and vertically to any depth. If v is not a picture, nothing is shown (this should be intended recursively).

contents: shows the names of the variables defined at the top level.

undo: the result of the last expression evaluated is always kept in the top level variable "it". The command "undo" can be used to reset "it" to its previous value (only once).

use: loads a module (described in section "Modules and externals").

import: imports an external picture (described in section "Modules and externals").

export: creates an external picture and generates a CIF file (described in section "Modules and externals").

3.8 Modules and Externals

Some modules (called library modules) are predefined in the system, as for example "constants" (basic cells) and "pla" (pla generator). Modules can contain data (like "constants") or programs (like "pla"), and can be used by the command:

```
- use constants,pla;
```

which loads the definitions contained in constants and pla.

New modules can be generated by editing files with extension ".STK", containing Sticks & Stones expressions and definitions. Every module can "use" other modules.

Externals arise when, at the end of a session, we want to save the pictures produced so far. If a very big and very time-consuming ALU (Arithmetic-Logic Unit) has been produced, it can be saved as follows:

```
- export ALU;

ALU exported
```

This command generates: (i) a CIF file of the ALU, called "ALU.CIF", and (ii) a file containing boundary information about the ALU, called "ALU.STX". The ALU can be recalled by:

```
- import ALU;

ALU = <> : ...
```

The advantage of externals is that it is possible to use the ALU in another session without having to build it again. To import something takes almost no time, as only boundary information (i.e. ports) is used (an imported picture is drawn as a white frame with ports). Moreover the ALU can be used as a component of a CPU, and when the CPU is exported, the system merges the already existing ALU.CIF file with the rest of the picture. CIF files generated by "export" can be used for plotting or for mask fabrication.

The import command is also used to interface already existing CIF files to Sticks & Stones. Given a CIF file REG.CIF, we only have to write a file REG.STX and then "import REG;". The STX file should contain a form describing the ports of the REG, and should declare it to have a figure (e.g. a box) of the right size:

```
let REG =
  form (VddIn:B port ...; VddOut:B port ...;
        GndIn:B port ...; GndOut:B port ...;
        BusIn:B port ...; BusOut:B port ...;
        ReadIn:R port ...; ReadOut:R port ...;
        WriteIn:R port ...; WriteOut:R port ...;
        ClockIn:R port ...; ClockOut:R port ...)
  with W line [[0↑0;36↑0;36↑36;0↑36;0↑0]];
```

"export" uses a "line" to generate a white frame, like in this example.

CIF files generated by Sticks & Stones are <u>compact</u>, as common subpictures are factorised into CIF symbols, and calls to these symbols are generated where necessary. Moreover they are <u>commented</u>: every CIF symbol is associated to the name(s) used in Sticks & Stones to denote it.

3.9 Efficiency

The composition algorithm is linear in the number of (bunch) connections and independent of the number of ports of the sorts involved.

If possible, iteration should be used instead of recursion and the "times" form of iteration should be preferred. In the latter case the iteration body needs to be evaluated just once (because the language is applicative) instead of n times. But what is more important, the system can use a logarithmic algorithm instead of a linear one, producing at any step 1,2,4,8,16 etc. instantiations of the iteration body and then composing them up to get the desired number. The gain in efficiency is considerable: to produce a 16x16 array of four-port cells the "times" iteration takes 8 connections against the 255 of the "for" iteration.

Because of the absence of side-effects, it is possible to share in memory everything that is sharable; hence "let" should be used to factorise common subexpressions. An array of 16x16 cells can be produced by allocating just one cell plus 8 connection records. If instead we put an expanded cell definition inside a double iteration with iteration variables we can cause the allocation of 256 identical cells plus 255 connection records.

3.10 Conclusions

The implementation of Sticks&Stones allowed us to gain some experience in the area of VLSI design tools, and to test and demonstrate the practical utility of the notation we are proposing. For example, the ideas of bunches and planar sorts can be considered a direct consequence of the implementation effort and of the fact that we had to cope with real-life circuits.

The subsequent investigation of more theoretical issues (described in chapters 1 and 2), together with the experience already gained, brought up new problems and ideas, so that **Sticks&Stones** would probably be rather different, if we had to implement it today. We would expecially like to make it safer to use (by more rigorous syntactic checks) and more interactive (by the use of pointing devices).

However we are now of the opinion that an experiment at the layout level of description should not be repeated, and in the field of silicon assemblers we should strive directly for stick-oriented systems, as suggested in sections 2.5 and 2.6. **Sticks&Stones**, in the present form, still retains much interest for computer graphics, for its ability to manipulate graphical and geometrical entities, and as alternative to turtle graphics.

3.11 Syntax

3.11.1 Syntax Definition

The notation used here is explained in Appendix I.

```
topterm ::= (command | toplet | topletrec | term) ';'

command ::= mode | addmode | submode | print |
            draw | undo | use | begin | end |
            contents | import | export
```

```
mode ::= 'mode'

addmode ::= 'addmode' {ide / ','}1

submode ::= 'submode' {ide / ','}1

print ::= 'print' term

draw ::= 'draw' term

undo ::= 'undo'

use ::= 'use' {ide / ','}1

begin ::= 'begin' port

end ::= 'end' port

contents ::= 'contents'

import ::=  'import' ide

export ::=  'export' ide


toplet ::= 'let' declaration

topletrec ::= 'letrec' declaration


term ::= variable | bool | string | number | point | pair |
         list | form | composition | restriction | rename |
         conditional | abstraction | application | iteration |
         let | letrec | where | whererec | parterm |
         and | or | not | minus | cons | append | sum | diff |
         times | divide | equal | great | less | greateq |
         lesseq | range | mod | directcomp | reverscomp
```

```
variable ::= ide

bool ::= 'true' | 'false'

string ::= '`' characters '`'

number ::= unsignedreal

point ::= term '↑' term

pair ::= term ',' term

list ::= '[' {term / ';'} ']'

form ::= 'form' [sort] ['with' {figure / 'and'}1]

sort ::= '(' {port ':' ide ['port' term] / ';'}1 ')'

figure ::= ide shape term

shape ::= 'dot' | 'line' | 'path' | 'spline' | 'loop' |

          'box' | 'poly' | 'area' | 'blob' | 'text'

composition ::= term connection term

connection ::= '[:' {port '—' port / ';'}1 ':]'

restriction ::= term '\' {match}1

rename ::= term '{' {substitution / ';'} '}'

substitution ::= match '\' match [term] |

                 match term

iteration ::= term 'times' term 'with' connection |

              'for' {struct 'in' term / 'and'}1

                    'iter' term 'with' connection

conditional ::= term '⇒' term '|' term

abstraction ::= '"' {struct}1 '"' term

application ::= term term

let ::= 'let' declaration 'in' term

letrec ::= 'letrec' declaration 'in' term

where ::= term 'where' declaration

whererec ::= term 'whererec' declaration

declaration ::= {funstruct '=' term / 'and'}1
```

```
funstruct ::= struct | ide {struct}1
struct ::= '(' ')' | ide | struct '↑' struct |
           struct ',' struct | '[' {struct / ';'} ']' |
           struct '_' struct | '(' struct ')'
parterm ::= '(' term ')'

and ::= term 'And' term
or ::= term 'Or' term
not ::= term 'Not' term
minus ::= '-' term
cons ::= term '_' term
append ::= term '∂' term
sum ::= term '+' term
diff ::= term '-' term
times ::= term '*' term
divide ::= term '/' term
equal ::= term '=' term
greater ::= term '>' term
less ::= term '<' term
greateq ::= term '>=' term
lesseq ::= term '<=' term
range ::= term '::' term
mod ::= term 'mod' term
directcomp ::= term 'o' term
reverscomp ::= term '&' term
```

```
letter ::= 'a' | ... | 'z' | 'A' | ... | 'Z' | ''''
digit ::= '0' | ... | '9'
ide ::= letter | ide letter | ide digit
matchide ::= '!' | '?' | ide '!' | ide '?' |
             matchide '!' | matchide '?' |
             matchide letter | matchide digit

integer ::= digit | integer digit
unsignedreal ::= integer ['.' integer]
port ::= ide | port '.' ide | port '.' integer
match ::= matchide | port '.' matchide |
          match '.' matchide | match '.' ide |
          match '.' integer
```

3.11.2 Precedence of Operators

"m * n" means that the infix operator "*" has left precedence m and right precedence n. An expression "x * y *' z" associates like "(x * y) *' z" if n>=m' and like "x * (y *' z)" if n<m'. Hence m<=n means that "*" is left associative and m>n that it is right associative.

100	Or	100
200	And	200
301	,	300
401	_	400
500	∂	500
600	=	600
700	>	700

700	<	700	
700	>=	700	
700	<=	700	
800	mod	800	
900	↑	900	
1000	::	1000	
1100	+	1100	
1100	-	1100	
1200	*	1200	
1200	/	1200	
1200	//	1200	
1300	o	1300	
1300	&	1300	
1400		1400	(application)

3.11.3 Predefined Functions

And (infix) boolean and.

Or (infix) boolean or.

Not (infix) boolean not.

= (infix) equality over booleans, numbers, points, pairs and lists only.

> (infix) greater than.

< (infix) less than.

>= (infix) greater then or equal to.

<= (infix) less than or equal to.

- (prefix) number complement.

+ (infix) number sum.

- (infix) number difference.

* (infix) number product.

/ (infix) number division.

// (infix) integer division.

mod (infix) number modulo: "a mod b" is the

remainder of "a//b".

lft point left: lft (a↑b) = a.

rht point right: rht (a↑b) = b.

fst pair first: fst (a,b) = a.

snd pair second: snd (a,b) = b.

hd list head: hd [a1; ... ;an] = a1 (n>0).

tl list tail: tl [a1; ... ;an] = [a2; ... ;an] (n>0).

null list null: null [] = true;

null [a1, ... ,an] = false (n>0).

_ (infix) list cons: a_[a1; ... ;an]

= [a;a1; ... ;an] (n>=0).

∂ (infix) list append: [a1; ... ;an] ∂ [b1; ... ;bm]

= [a1; ... ;an;b1; ... ;bm] (n,m>=0).

:: (infix) range: n::m = [n;n+1; ... ;m-1;m] (n<=m);

n::m = [n;n-1; ... ;m+1;m] (n>=m).

length list length: length [a1; ... ;an] = n (n>=0).

o (infix) function composition: (f o g) a = f (g a).

& (infix) reverse function composition: (f & g) a = g (f a).

4. Analog Processes

In this chapter and in the next one we try to set up formal frameworks in which the semantics of low-level hardware can be defined and studied. In the process we revert from net algebras to Milner's flow algebras [Milner 79] for consistency with existing literature and because flow algebras are more convenient from a formal point of view.

4.1 Introduction

In this chapter we develop a formal framework for describing continuous interaction, like for example the gravitational interaction of planets around a star. These interactions are not "communications" in the sense of discrete packets of information being exchanged, but rather various forms of "being in contact" on an instant by instant basis.

Although most of the phenomena in concurrent systems can be studied in a discrete framework, some of them seem to imply some notion of continuity or, at least, of arbitrarily small discreteness. A very well known example is the arbitration problem, which disappears as soon as a discrete time scale is introduced; other examples include measurement problems, and the study of asynchronous interaction of internally synchronous systems. Most of these problems are unwelcome, both from the theoretical and practical point of view, and their study can help in understanding when they can be safely ignored or controlled.

Asynchronous electronic circuits will be used as a source of interesting examples, and we shall be able to model and analyse asynchronous feedbacks, metastable states, arbitration and indeterminacy. We shall also discover some basic (and plausible) limitations on the kinds of systems we can express, which seem to indicate some correspondence between our model and what we may consider to be "physically feasible" processes.

Finally, it is interesting to notice that all these phenomena arise from the mere consideration of concurrency in real time, and do not necessarily depend on other characteristics of the physical universe, like quantum mechanic or relativistic effects.

4.2 Analog Processes

A signal is a value varying through (continuous) time, which is carried by a line (we use α, β etc. for lines). An analog process is a collection of transformations of such signals (called transitions), for example:

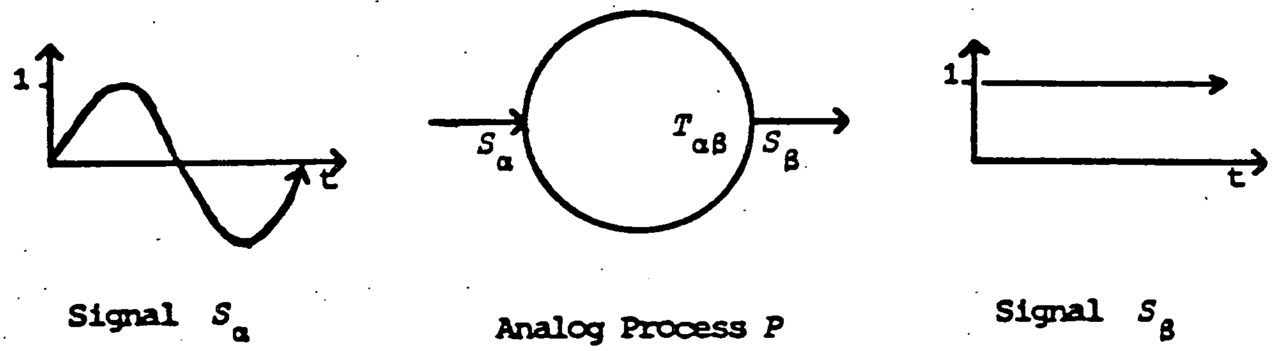


Figure 4.1 A process

The signals above can be expressed as functions of time:

$$S_\alpha(t) = \sin t \qquad S_\beta(t) = 1$$

and the process P transforming S_α into S_β can be described by a transition $T_{\alpha\beta}$ which in this case might be:

$$T_{\alpha\beta}(s)(t) = s(t) - \sin t + 1$$

For then, applying $T_{\alpha\beta}$ to S_α we get S_β as we have:

$$\begin{aligned} T_{\alpha\beta} &= \lambda t.\ S_\alpha(t) - \sin t + 1 \\ &= \lambda t.\ \sin t - \sin t + 1 \\ &= \lambda t.\ 1 \\ &= S_\beta \end{aligned}$$

In general a process will consist of several transitions, and systems will comprise several connected processes.

4.3 An Algebra of Analog Processes

A process is described by a collection of transitions $M \rightsquigarrow \beta$, where the term M denotes the signal produced by the transition, and β is an identifier denoting the output port of the transition. The signal M is an expression of the input ports of the process. Here is an example of the syntax we shall use to talk about transitions:

$$(\alpha \rightsquigarrow \beta) \between ((\alpha \uplus \gamma) \rightsquigarrow \delta)$$

For clarity we shall sometimes prefix processes with input ports, although this is not strictly necessary as the input ports of a process will always coincide with the free variables of the signal part of the transitions:

$$\alpha\ \gamma:\ \alpha \rightsquigarrow \beta \between \alpha \uplus \gamma \rightsquigarrow \delta \qquad (*)$$

This is a process with input ports α,γ and output ports β,δ (parentheses have been omitted).

The intended behaviour of processes will be explained by algebraic laws. We shall only be concerned with some of the laws and we shall not try to present a complete set of equations. The following three laws express the fact that processes are unordered collections of transitions:

$$[\between\between] \quad (T \between T') \between T'' = T \between (T' \between T'')$$
$$[\between] \quad T \between T' = T' \between T$$
$$[NIL] \quad T \between NIL = T$$

where NIL is the empty transition and T,T' and T" range over transitions.

The intended meaning of the expression (*) above is a process which at any instant of time produces on the output port β the current value of the input port α, and on the output port δ the current value of the join ($\uplus$) of α with γ. The join operator represents the simultaneous presence of two signals on the same "line", and its exact meaning is left unspecified, except that the join operation must exist for every pair of signals (of the same type) and it must satisfy:

[$\uplus\uplus$] $\quad (M \uplus M') \uplus M'' = M \uplus (M' \uplus M'')$

[$\uplus$] $\quad M \uplus M' = M' \uplus M$

For example, for boolean-valued signals s', s'' we might define $s' \uplus s''$ to be at any instant of time a boolean or, i.e.:

$(s' \uplus s'')(t) = s'(t) \vee s''(t)$

The existence of a constant $\perp$ (nosignal) is also assumed; it relates to join as follows:

[$\perp$] $\quad M \uplus \perp = M$

In the previous boolean example we can define nosignal as the signal constantly false, i.e.: $\perp(t)$ = false. The join operation is also used in the following law, which accounts for the presence of repeated output ports:

[$\uplus X$] $\quad M \rightsquigarrow \beta \; X \; N \rightsquigarrow \beta = M \uplus N \rightsquigarrow \beta$

Now we define some basic operators on processes, together with their algebraic laws.

4.3.1 Composition

The composition of two processes P and Q is written P|Q. The output ports of P are linked to those input ports of Q with the same name, and the output ports of Q are linked to the input ports of P with the same name; the idea being that signals flow through these connections from one process to the other. We have the following laws for composition:

[||] $(P \mid Q) \mid R = P \mid (Q \mid R)$

[|] $P \mid Q = Q \mid P$

[|X] $(\prod_{i \in I} T_i) \mid (\prod_{j \in J} T_j) = \prod_{k \in I \cup J} T_k$

where I and J are disjoint sets of indexes

(Here $\prod_{i \in I} T_i$ abbreviates $T_1 \times \ldots \times T_n$ with $I=\{1,\ldots,n\}$)

An example of law [|X] is:

$(\alpha: \alpha \rightsquigarrow \beta) \mid (\beta: \beta \rightsquigarrow \gamma) = \alpha\,\beta: \alpha \rightsquigarrow \beta \times \beta \rightsquigarrow \gamma$

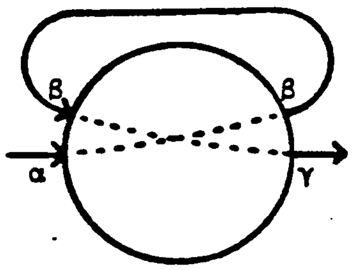


Figure 4.2 Composition

Note that composition may introduce loops (β being both an input and an output port) and indeed such loops may be present in the first place. We shall come later to the exact semantics of such situations; for the moment just think of a looping signal as overwriting itself by a join operation.

4.3.2 Restriction

The restriction $P\backslash\alpha$ of P cancels α from the input and output ports of P, making communication via α impossible. We have:

[\] $P\backslash\alpha = P$ if $\alpha \notin$ ports(P)

[\\] $P\backslash\alpha\backslash\beta = P\backslash\beta\backslash\alpha$

[\|] $(P \mid Q)\backslash\alpha = P\backslash\alpha \mid Q\backslash\alpha$

if not (($\alpha \in$ in-ports(P) and $\alpha \in$ out-ports(Q)) or

($\alpha \in$ out-ports(Q) and $\alpha \in$ in-ports(P)))

Now we need laws to distribute \ over χ, and at first sight these might be:

$$(\prod_{i\in I} T_i)\backslash\alpha = \prod_{i\in I}(T_i\backslash\alpha)$$

$$(M \rightsquigarrow \alpha)\backslash\alpha = NIL$$

$$(\ldots\ \alpha\ \ldots \rightsquigarrow \beta)\backslash\alpha = \ldots \div \ldots \rightsquigarrow \beta$$

Unfortunately this does not work well in the case:

$$(\alpha\colon M \rightsquigarrow \alpha \ \chi\ \alpha \rightsquigarrow \beta)\backslash\alpha = \div \rightsquigarrow \beta$$

In fact we wish to interpret $\backslash\alpha$ as a hiding operator, which should not change the inner behaviour of the process. The result we want to get is, at least:

$$(\alpha\colon M \rightsquigarrow \alpha \ \chi\ \alpha \rightsquigarrow \beta)\backslash\alpha = M \rightsquigarrow \beta$$

But even this is not enough in the case where M is an expression M[α] of α itself, e.g. when we have a loop over the restriction variable whose result is exported through another output port (in this case β). To solve these problems we need to introduce recursively defined signals ($\mu\alpha$. M):

[μ] $\mu\alpha.\ M = \mu\beta.(M[\beta/\alpha])$

[$\mu\mu$] $\mu\alpha.\ M = M[(\mu\alpha.\ M)/\alpha]$

Then the law for restriction is:

[\X] $(\prod_{i \in I} M_i \rightsquigarrow \alpha_i)\backslash\alpha = \prod_{j \in J} T'_j$
where $J = \{i \in I:\ \alpha_i \neq \alpha\}$
and $T'_j = (M_i \rightsquigarrow \alpha_i)[(\mu\alpha.\ \uplus_{\alpha_i=\alpha} M_i)/\alpha]$

Here $\uplus_{i \in I} M_i$ is the join of all the M_i, and it is $\dot{-}$ if I is empty.

Examples:

$(\alpha:\ \alpha \rightsquigarrow \beta)\backslash\alpha = (\mu\alpha.\ \dot{-}) \rightsquigarrow \beta = \dot{-} \rightsquigarrow \beta$

$(\alpha\ \beta:\ \alpha \rightsquigarrow \beta\ X\ \beta \rightsquigarrow \gamma)\backslash\beta = \alpha:\ \alpha \rightsquigarrow \gamma$

$(\alpha\ \beta:\ \alpha \rightsquigarrow \beta\ X\ \beta \rightsquigarrow \alpha)\backslash\beta = \alpha:\ \alpha \rightsquigarrow \alpha$

$(\alpha:\ \alpha \rightsquigarrow \alpha)\backslash\alpha$ = NIL

$(\alpha:\ \alpha \rightsquigarrow \alpha\ X\ \alpha \rightsquigarrow \beta)\backslash\alpha = (\mu\alpha.\ \alpha) \rightsquigarrow \beta$

The important point in law [\X] is that looping situations are somehow hidden of preserved, but never "unfolded" by $\backslash\alpha$.

4.3.3 Renaming

The renaming $P\{\alpha_1/\beta_1,\dots,\alpha_n/\beta_n\}$ is the process obtained from P by simultaneously substituting $\alpha_1\dots\alpha_n$ for the (input and/or output) ports $\beta_1\dots\beta_n$. A renaming $\{R\} = \{\alpha_i/\beta_i\}$ is a bijection $R:L \to R(L)$ over the ports L of P, i.e. the β_i the ports of P, and the α_i are distinct. Dummy substitutions will be omitted, so that $\{\} = \{\alpha_i/\alpha_i\}$.

[{}] $P\{\} = P$

[{}{}] $P\{R\}\{S\} = P\{S \circ R\}$

[{}\] $(P\backslash\alpha)\{R\} = (P\{R,\ \beta/\alpha\})\backslash\beta$
if $\alpha \in$ ports(P) and $\beta \notin$ range(R)

[{}|] $(P|Q)\{R\} = (P\{R'\})|(Q\{R''\})$
where R' = R restricted to ports(P)
and R'' = R restricted to ports(Q)

To distribute {R} over χ we actually perform a syntactic substitution:

$$[\{\}\chi] \qquad (\prod_{i\in I} T_i)\{\alpha_j/\beta_j\} = \prod_{i\in I}(T_i[\alpha_j/\beta_j])$$

Example:

$$(\alpha\ \beta:\ \alpha \rightsquigarrow \beta\ \chi\ \beta \rightsquigarrow \alpha)\{\alpha/\beta,\ \beta/\alpha\} = \beta\ \alpha:\ \beta \rightsquigarrow \alpha\ \chi\ \alpha \rightsquigarrow \beta$$

The algebraic laws we have presented so far form what we shall call an analog algebra. These laws can be grouped into two categories: external laws (relating |, \α and {R}: [||], [|], [\], [\\], [\|], [{}], [{}{}], [{}\] and [{}|]) concerning the synthesis of processes from simpler processes, and internal laws (all the others) concerning the inner structure of processes. The external laws are just those of Milner's flow algebras [Milner 79]. Flow algebras are extended in [Milner 78] by a set of internal laws for communicating processes, and are then called behaviour algebras. Our internal laws are quite different from Milner's ones, but they seem to fit very well in the general framework of flow algebras, even if the meaning of |, \α and {R} is radically different.

4.4 A Denotational Model

In the rest of this chapter we shall study a particular analog algebra, built within the framework of denotational semantics. This will allow us to study the exact meaning of processes just by computing their semantics and observing their input-output behaviour. The denotational semantics will also prove useful in discussing some delicate situations arising from feedback loops and recursively defined signals.

Processes are collections of transitions; in particular $P_{L,L'}$ is the domain of processes with L inputs and L' outputs, namely associations of transitions with L inputs to the output ports L':

$P_{L,L'} = L' \rightarrow T_L$

Here L,L' range over finite subsets of PLab, the set of port labels, and T_L is the domain of transitions with L inputs (and one output). The domain P of processes is given by:

$P \triangleq \Sigma_{L,L'} P_{L,L'}$

A transition with L inputs is a function taking |L| input signals and producing an output signal, hence:

$T_L \triangleq S^L \rightarrow S$

where S is a domain of signals.

Signals are functions from time to a domain of values. We can have several types of signals, like boolean signals, real signals, etc.

$S \triangleq K \rightarrow V$

where K is the flat domain of positive real numbers, and V is a given data domain which is an abelian monoid $\langle V,\phi,\vee\rangle$ with $\vee$ strict (i.e. $\bot \vee x = \bot$). We define:

$\bot(t) \triangleq \phi$

$(s' \uplus s'')(t) \triangleq s'(t) \vee s''(t)$

for all $t \in K$ and $s',s'' \in S$. This definition will make [$\uplus\uplus$], [$\uplus$] and [$\bot$] hold when we give the semantics.

We need some notation for elements in these domains; λ-notation will be used for signals $s \in S = K \rightarrow V$. Elements of S^L will be denoted by expressions like:

$[a_1{:}s_1, \ldots, a_n{:}s_n]$ for $a_1..a_n \in L$, $s_1..s_1 \in S$

which are meant to be unordered tuples of labelled signals $a_i{:}s_i$ with the additional property:

$[\ ..\ a{:}s',\ a{:}s''\ ..\ ] = [\ ..\ a{:}s' \uplus s''\ ..\ ]$

and operations:

$\backslash\alpha : S^L \rightarrow S^{L\backslash\{\alpha\}}$

$.\alpha : S^L \rightarrow S$

$\uplus : S^L \times S^{L'} \rightarrow S^{L\cup L'}$

defined as:

$[\alpha_i:s_i]\backslash\alpha = [\alpha_j:s_j]$ with $i\in I$, $j\in\{i\in I \mid \alpha_i\neq\alpha\}$

$[\alpha_i:s_i].\alpha = \uplus\{s_k \mid \alpha_k=\alpha\}$ where $\uplus\{\} = \bot$

$[\alpha_i:s_i] \uplus [\alpha_j':s_i'] = [\alpha_i:s_i,\ \alpha_j':s_i']$

Elements of $T_L = S^L \rightarrow S$ of the form:

$\lambda x.\ \dots\ x.\alpha_1\ \dots\ x.\alpha_n\ \dots$

will be abbreviated (with a change of font) as:

$\lambda[a_1\ \dots\ a_n].\ \dots\ a_1\ \dots\ a_n\ \dots$

where $[a_1\ \dots\ a_n]$ is an unordered tuple of variables. Notice that this notation allows for unordered application by label names (i.e. call-by-keyword), as in:

$(\lambda[a_1\ a_2].\ a_1 * a_2)[\alpha_2:3,\ \alpha_1:5] = 5*3$

Finally, processes $p \in P_{L,L'} = L' \rightarrow T_L$ of the form:

$\lambda x.\ (x=\alpha_1) \Rightarrow t_1;\ \dots\ ;\ (x=\alpha_n) \Rightarrow t_n;\ (\lambda[].\ \bot)$

(where "$a\Rightarrow b;c$" means "if a then b else c") will be abbreviated as:

$\{t_1 \rightsquigarrow \alpha_1;\ \dots\ ;\ t_n \rightsquigarrow \alpha_n\}$

There are three semantic evaluation functions:

$\mathbb{T}$: terms $\times$ ports $\times$ vars $\rightarrow$ T	for term expressions
$\mathbb{S}$: signals $\times$ ports $\rightarrow$ S	for signal expressions
$\mathbb{P}$: processes $\times$ ports $\rightarrow$ P	for process expressions

with two kinds of environments:

vars = Ide $\rightarrow$ V

ports = L $\rightarrow$ S

We shall first discuss the semantics of process expressions, then the semantics of signal expressions, giving the syntax at the same time. We shall not treat the semantics of terms, as term expressions will always have an evident meaning.

The following is the semantics of a very simple process, consisting of a single transition:

$\mathbb{P}[\![\alpha_i : S \rightsquigarrow \beta]\!]\sigma =$

$\quad Y\, \lambda P.\ \{\lambda[a_i].\ \mathbb{S}[\![S]\!]\sigma[(a_i \uplus P(\alpha_i)[\alpha_j : a_j])/\alpha_i] \rightsquigarrow \beta\}$

(note that $P(\alpha_i)[\alpha_j:a_j] = \bot$ if $\alpha_i \neq \beta$).

The fixpoint and the join operation are needed just in case β is equal to one of the α_i, i.e. when there is a feedback. Otherwise the previous expression reduces simply to:

$\quad \{\lambda[a_i].\ \mathbb{S}[\![S]\!]\sigma[a_i/\alpha_i] \rightsquigarrow \beta\}$

In case of feedback, say $\alpha_3=\beta$, the input to α_3 is a_3 (the input to process P) joined to what comes out of β, which is $P(\alpha_3)[\alpha_i:a_i]$. In fact $P(\alpha_3)$ is the transition associated with $\alpha_3=\beta$, which receives as input the same input of the process: $[\alpha_j:a_j]$.

The same idea is used in giving the semantics of composition, in which the component processes may feed each other in complex ways. The composition operation on processes is defined as:

$$\begin{aligned}
p|q = {} & \text{let } p = \{s_i \rightsquigarrow \gamma_i\} \text{ where } s_i = \lambda[a_h].\ M_i \\
& \quad \text{and } q = \{r_j \rightsquigarrow \delta_j\} \text{ where } r_j = \lambda[b_k].\ N_j \text{ in} \\
& Y\, \lambda R.\ \{\lambda[a_h b_k].\ s_i[\alpha_h:(a_h \uplus R(\alpha_h)[\alpha_h:a_h,\ \beta_k:b_k])] \rightsquigarrow \gamma_i\} \\
& \quad \uplus \{\lambda[a_h b_k].\ r_j[\beta_k:(b_k \uplus R(\beta_k)[\alpha_h:a_h,\ \beta_k:b_k])] \rightsquigarrow \delta_j\}
\end{aligned}$$

and we have the evident semantics:

$\quad \mathbb{P}[\![P|Q]\!]\sigma = \mathbb{P}[\![P]\!]\sigma \mid \mathbb{P}[\![Q]\!]\sigma$

This composition is commutative ([|] holds); to prove associativity ([||]) we had to assume absorption of $\uplus$, i.e. $s \uplus s = s$ (which also implies $P|P = P$; we do not know whether this is a necessary condition). The other laws of analog algebras are easily verified, if we complete the definition of $\mathbb{P}$ by the following equations:

$$\mathbb{P}[\![NIL]\!]\sigma = \lambda x.\ (\lambda[].\ \perp)$$

$$\mathbb{P}[\![T_1 \between \dots \between T_n]\!]\sigma = \mathbb{P}[\![T_1]\!]\sigma \mid \dots \mid \mathbb{P}[\![T_n]\!]\sigma$$

$$\mathbb{P}[\![P\backslash a]\!]\sigma =$$
$$\lambda\beta\varepsilon L'.\ \lambda x\varepsilon S^{L\backslash\{a\}}.\ (\mathbb{P}[\![P\varepsilon P_{L,L'}]\!]\sigma)(\beta)(x\backslash a \uplus [a:\perp])$$

$$\mathbb{P}[\![P\{\beta_i/a_i\}]\!]\sigma =$$
$$\text{let } p = \mathbb{P}[\![P]\!]\sigma$$
$$\text{in } \lambda\gamma.\ \lambda[b_i].\ \gamma=\beta_1 \Rightarrow p(a_1)[a_i:b_i];\ \dots;\ \gamma=\beta_n \Rightarrow p(a_n)[a_i:b_i];\ \perp$$

We now consider signals; a simple way to specify them is to describe their value at any instant of time, using a sort of λ-notation, where "∂t" is read "at time t" and t is the only variable (if any) free in V:

$$\mathbb{S}[\![\partial t.V]\!]\sigma = \lambda x.\ \mathbb{T}[\![V]\!]^{\varepsilon}_{\wedge}[x/t] \qquad (\varepsilon \text{ is the empty environment})$$

for example ∂t. 3*sin t. We have the equivalences $\perp = \partial t.\ \phi$ and $a \uplus b = \partial t.\ a(t) \vee b(t)$. The notation $\uparrow V$ will be used as an abbreviation for ∂t. V, when t is not a free variable in V, like in $\uparrow 3 = \partial t.\ 3$.

Signals can also be defined by recursion:

$$\mathbb{S}[\![\mu a.\ S]\!]\sigma = Y\ \lambda a.\ \mathbb{S}[\![S]\!]\sigma[a/a]$$

like in

$$\mu a.\ \partial t.\ t<1 \Rightarrow \phi;\ a(t-1) \ \equiv\ \perp$$

Two other useful abbreviations are conditional signals and delays:

$$S \Rightarrow S'\ ;\ S'' \ =\ \partial t.\ S(t) \Rightarrow S'(t)\ ;\ S''(t)$$

$$S' \,\Delta\, S'' \ = \partial t.\ t<S''(t) \Rightarrow \phi\ ;\ S'(t-S''(t))$$

A simple example of delay is $S\ \Delta \uparrow 3$ which is the signal S constantly delayed by 3 units of time, yielding ϕ during the first three units of time. This notation also allows us to express variable delays.

Notice that the ∂-notation has too great an expressive power, being able for example to define a signal in terms of the "future" of another signal (or even of itself; e.g. μα. ∂t. α(t+1)), but we might impose syntactic restrictions to avoid that, leaving Δ as a primitive.

Summarising the syntax, we have terms V, signals M and processes P. Terms are boolean expressions and conditionals with at most one free variable t ranging over reals.

```
V ::= 'true' | 'ϕ' | BooleanExpression |
      V '⇒' V ';' V | Port '(' t ')'

M ::= '∂t.' V | '↑' V | '∸' | Port |
      M 'Ʉ' M | 'μ' Port '.' M |
      M '⇒' M ';' M | M 'Δ' M |
      M ('+' | '-' | '*' | '/' | '=' ) M

P ::= {Port}1 ':' {M '⇝' Port / 'X'}1 |
      NIL | P '|' P | P '\' Port |
      P '{' {Port '/' Port / ','} '}'
```

4.5 Feasibility

Great care has been put into the definition of the algebraic laws and of the denotational semantics, in order to be able to treat circularities. The simplest example of feedback can be found in the following fast loop process:

α: α ⇝ α

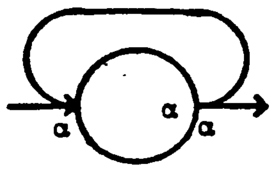


Figure 4.3 Fast loop

This process has an input port α, whose input is mixed to the output coming from the output port α. This process has no internal delay, and the output at any instant t depends on the input at the same instant t, which depends again on the output at time t. Computing the semantics:

$$p \triangleq \mathbb{P}[\![\alpha:\ a \rightsquigarrow \alpha]\!]\sigma$$
$$= Y\ \lambda P.\ \{\lambda[a].\ \mathbb{S}[\![a]\!]\sigma[a \uplus P(\alpha)[\alpha{:}a]/a] \rightsquigarrow \alpha\}$$
$$= Y\ \lambda P.\ \{\lambda[a].\ a \uplus P(\alpha)[\alpha{:}a] \rightsquigarrow \alpha\}$$

It is not immediately clear what p does, but we can try to understand its behaviour by applying some input. We first extract the transition we are interested in (there is only one in this case) applying it to the output port α:

$$p(\alpha) = \lambda[a].\ a \uplus p(\alpha)[\alpha{:}a]$$

Then we apply an input signal to see what is the response of the transition:

$$p(\alpha)[\alpha{:}a] = a \uplus p(\alpha)[\alpha{:}a] = \bot$$

the result is $\bot$, because of strictness of $\uplus$.

Here we have a first example of a clearly "infeasible" process, which denotes $\bot$, the undefined element. We can also see that a slow loop is not mapped to $\bot$ and is well-defined everywhere. Set

$$p \triangleq \mathbb{P}[\![\alpha:\ a\ \Delta \uparrow 1 \rightsquigarrow \alpha]\!]\sigma$$
$$= Y\ \lambda P.\ \{\lambda[a].\ \lambda t.\ t<1 \Rightarrow \phi;\ (a \uplus P(\alpha)[\alpha{:}a])(t-1) \rightsquigarrow \alpha\}$$
$$p(\alpha)[\alpha{:}a] = \lambda t.\ t<1 \Rightarrow \phi;\ (a \uplus p(\alpha)[\alpha{:}a])(t-1)$$

There are also processes whose output signals are only partially undefined; an example is the Zeno loop:

α: $\alpha\ \Delta\ (\partial t.\ t<1 \Rightarrow 1-t;\ 0) \rightsquigarrow \alpha$

This is a feedback loop which increases its speed, and at a finite point in time reaches an infinite speed (i.e. a zero delay). The output of the Zeno loop for a nosignal input is $\lambda t.\ t<1 \Rightarrow \phi;\ \bot$.

As a general principle, the output of a feedback loop is defined as long as the delay in the loop is greater than zero. This may look trivial, but feedback loops appear in almost any interesting process, and this simple fact has several intriguing consequences. We are going now to look at some of these.

4.6 Expressibility

We have seen that we can express several physically infeasible processes. This suggests that our formalism has too great an expressive power, and we might try to impose some constraints in order to exclude unwanted processes. However it would be wrong to think that we can express anything we like. In particular there are several processes which, we conjecture, cannot be exactly expressed, and yet admit approximations up to an arbitrary degree of accuracy. We shall call such inexpressible processes **perfect**, and shall call their expressible approximations **imperfect**.

Consider for example the following (naive) **memory** cell:

$\alpha\ \beta$: $\alpha \mho \beta\ \Delta\ \uparrow 1 \rightsquigarrow \beta$

To work properly as a (write once) memory cell, this process must receive a set impulse of length 1 on α. Then this impulse enters the loop and is "remembered". This memory cell presents two main defects: it will not work properly (i) if the set impulse is longer than 1 as it will overwrite itself, or (ii) if the set impulse is shorter than 1, as it will not fill the loop period. We can solve the first problem by the following (improved) **memory** cell:

$\alpha\ \beta$: $(\alpha{=}\bot \Rightarrow \alpha\ ;\ \beta)\ \Delta\ \uparrow 1 \rightsquigarrow \beta$

This process will cut off its α line after having received a signal different from $\perp$ for one unit of time. But the second problem still remains; if the α signal differs from $\perp$ for less than one unit of time, the output β is not constant. The same problem occurs when the set impulse changes its value during the setting time; then a varying signal is recorded into the feedback loop and the output of the memory cell oscillates: we get a (quench free) metastable state.

In effect what we really want is a perfect memory cell which stores constantly the value of an instantaneous setting spike, so that there can be no indeterminacy due to fluctuations of the input signal. Notice that starting from our improved memory cell we can get better and better approximations to a perfect cell, simply by reducing the delay in the feedback loop. Unfortunately if we reduce the delay to zero, we do not get a perfect storage device, but only an undefined output. Hence there seems to be no expression denoting a perfect memory cell (which yet exists inside our semantics domains) because there seems to be no way of defining a storing device without the use of feedbacks.

Therefore, expressible memory cells are imperfect. It is important to notice that many useful processes have memory cells (or their equivalent) as basic building blocks, and such processes must take into account this imperfection and are likely to be themselves imperfect. In general an imperfect process works "correctly" under some classes of input signals, but in certain critical circumstances there is no way to guarantee its intended operation.

4.7 Indeterminacy

Consider the problem of designing a process which determines the time of occurrence of an event, or which measures the value of a signal when some event (e.g. "measure it now") occurs. First we must agree on a definition of determining or measuring, and a sensible one seems to be storing constantly for an unlimited amount of time. We shall not go into the details of such design because it is very similar to the problem of producing a perfect memory cell. In fact it is not difficult to see that perfect determination is impossible, just because perfect storage devices are infeasible.

A well known case of indeterminacy is arbitration, where a device attempts to determine which of two events arrives first. A simple way of implementing an arbiter is to use a decider and a memory cell. The decider tells at any instant whether the first, the second or both signals are arriving, and the memory cell tries to remember the first decision of the decider. But memory cells are imperfect and so are arbiters based on memory cells. If the two signals arrive too close, the decider changes its decision while the memory cell is storing it, and the output of the cell is unstable.

If we had a perfect memory cell we could build a perfect arbiter this way:

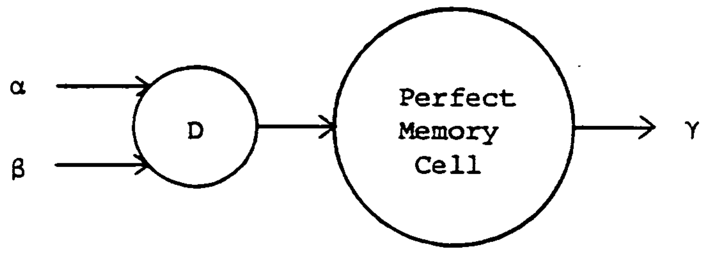


Figure 4.4 An arbiter

where the decider D is

$D \triangleq \alpha\ \beta$:

$(\alpha = \perp) \Rightarrow (\beta = \perp) \Rightarrow \perp$; "$\beta$ first";

$(\beta = \perp) \Rightarrow$ "α first";

"α and β together"

$\rightsquigarrow \gamma$

which at any instant outputs one of four different messages: $\perp$, "α first", "β first" or "α and β together". The perfect cell then remembers the first (arbitrarily short) decision different from $\perp$.

An alternative way of building an arbiter is by using two detectors to determine the time of occurrence of two events, and then compare these times. But detectors are imperfect because time is a continuously changing quantity which cannot be stored instantaneously, hence arbiters built in this way are imperfect.

In general the order or coincidence in time of two events cannot be determined. The order cannot be determined when the signals are too close, and the coincidence cannot be determined when the simultaneous signals are too short.

4.8 Flip-Flops

In this last section we analyse a particular analog process, showing in detail how its behaviour can be derived from its semantics. Here V={true,false,$\perp$}, ϕ=false and $\bigvee$=or.

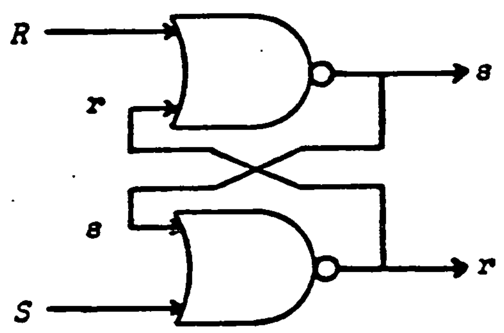


Figure 4.5 Flip-Flop

This is an SR flip-flop. In one of its steady state conditions we have the following values on the ports:

R = S = s = false;   r = true

Starting from this condition and applying a set pulse to the port S we get s = true and r = false. Another set pulse has no effect. Then applying a reset pulse to the port R we change the output back to s = false and r = true. Another reset pulse has no effect. Applying both a set and a reset signal, the output signals oscillate between true and false, and this is called a metastable state. The actual behaviour of a real flip-flop in a metastable state can be rather different from the one described above [Chaney 73]. We believe it can be modelled by introducing some "quench", but we shall not undertake this analysis here.

The SR can be synthesized from smaller components:

$$
\begin{aligned}
\text{OR} &= \text{in1 in2: (in1 or in2)} \ \Delta \ \uparrow d' \rightsquigarrow \text{out} \\
\text{NOT} &= \text{in: (not in)} \ \Delta \ \uparrow d'' \rightsquigarrow \text{out} \\
\text{OR1} &= \text{OR } \{R/\text{in1},\ r/\text{in2},\ w1/\text{out}\} \\
\text{OR2} &= \text{OR } \{S/\text{in1},\ s/\text{in2},\ w2/\text{out}\} \\
\text{NOT1} &= \text{NOT } \{w1/\text{in},\ s/\text{out}\} \\
\text{NOT2} &= \text{NOT } \{w2/\text{in},\ r/\text{out}\}
\end{aligned}
$$

$$\text{SR} = (\text{OR1} \mid \text{NOT1} \mid \text{OR2} \mid \text{NOT2})\backslash w1 \backslash w2$$

It is an easy exercise to show that this is equivalent to:

$$
\begin{aligned}
\text{SR} = \text{S R s r:}& \\
&\text{not(R or r)} \ \Delta \ \uparrow d \rightsquigarrow s \quad \chi \\
&\text{not(S or s)} \ \Delta \ \uparrow d \rightsquigarrow r
\end{aligned}
$$

where d = d'+d". Unfortunately if we try to switch on the flip-flop without supplying any signal (i.e. supplying false on all the inputs) we immediately get a metastable state. This happens because starting with false on all the inputs, we are not in the steady state condition. To enforce a well defined start, we supply true to r for the first d seconds. At that time the signal from S reaches r and the system is ready to work. Hence we redefine:

```
SR  =  S R s r:
            not(R or r) Δ ↑d ⤳ s  Ⅹ
            (not(S or s) Δ ↑d) Ʉ (∂t. t<d) ⤳ r
```

Computing the semantics:

```
SR = ℙ⟦SR⟧σ
   = Y λSR.
        {λ[S R s r]. λt.
            t<d ⇒ false;
            not(R(t-d) or r(t-d) or SR(r)[S:S,R:R,s:s,r:r](t-d))
         ⤳s;
         λ[S R s r]. λt.
            t<d ⇒ true;
            not(S(t-d) or s(t-d) or SR(s)[S:S,R:R,s:s,r:r](t-d))
         ⤳r}
```

and extracting the output transitions:

```
SR(s)  =  Y λT. [λ[S R s r]. λt. t<d ⇒ false;
                  not(R(t-d) or r(t-d) or
                   (t<2d ⇒ true;
                    not(S(t-2d) or s(t-2d) or
                      T[S:S,R:R,s:s,r:r](t-2d)))

SR(r)  =  ...
```

We look at the output signals in absence of input:

```
SR(s)[S:⊥, R:⊥, s:⊥, r:⊥]
  = Y λS. λt. t<2d ⇒ false; S(t-2d)
  = λt. false

SR(r)[S:⊥, R:⊥, s:⊥, r:⊥]
  = λt. true
```

This means that for S = ↑false we obtain s = ↑false, r = ↑true; we are in the steady state condition. Now we supply a pulse (λt. t<π) of an unspecified length π:

```
SR(s)[S:(λt. t<π), R:⊥, s:⊥, r:⊥]
  = Y λS. λt. t<2d ⇒ false; t<2d+π ⇒ true; S(t-2d)
```

There are two cases: (i) the length of the set pulse is $\pi \geq 2d$; then the flip-flop is properly set (the expression above reduces to λt. t<2d ⇒ false; true)

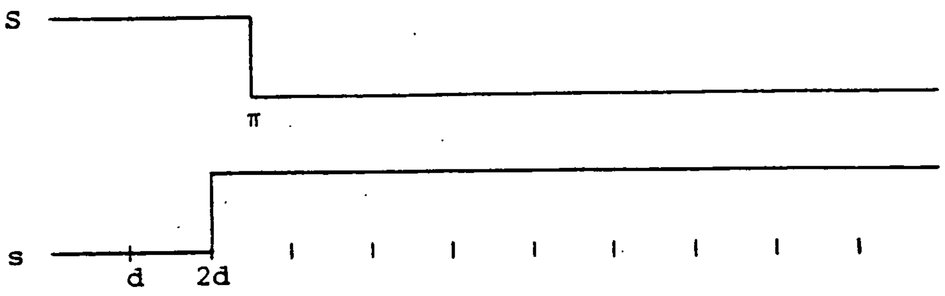


Figure 4.6 Stable state

or (ii) the length of the set pulse is $\pi<2d$; then the flip-flop is in a metastable state and the output signal oscillates between true and false.

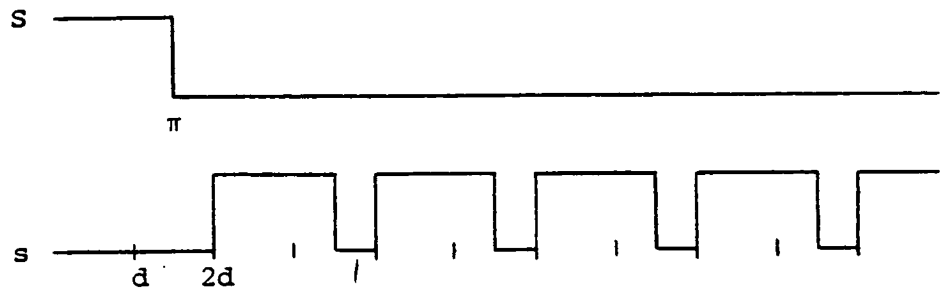


Figure 4.7 Metastable state

4.9 Conclusions

We have shown how analog processes can be studied from a semantic point of view. The proof techniques for equivalence (a process is a simplified form of another one) and correctness (a process implements a given transition) of analog processes are reduced to the standard proof techniques used in denotational semantics.

Direct "execution" of the semantic equations of a process provides a simulation technique. If we wish to know the output of a port at time t, we apply t to the output signal of the corresponding transition; the value is computed recursively backwards in time until (hopefully) a base value is found near time 0. In this sense it would be possible to devise an implementation for the language we have described.

Several semantic problems need further investigation, expecially regarding the relations between the formal semantics and our intuitions about analog processes.

5. Real Time Agents

Without trying to make any final assessment of the structure of the physical world, one might take the view that at an appropriate level of abstraction there are entities which act and influence each other's behaviour through a continuous interaction. These entities are called here agents and their interactions are assumed to happen in real time. The picture becomes particularly interesting when we allow our agents to behave nondeterministically both in the actions they can perform and in the time they take to do it. The ability to express nondeterministic systems is the major difference between this chapter and the previous one, deeply influencing the semantic techniques we use.

5.1 Introduction

This chapter is inspired by Milner's approach to synchronous processes, as reported in [Milner 81]. The main differences are the use of a continuous time domain and a continuous-nondeterminism operator. Milner has shown that many of the characteristics of concurrent processes can be modelled and, more importantly, manipulated in an algebraic framework tailored to synchronous discrete interaction. Although much can be done in a discrete-time model by reducing the grain of discreteness to the desired level, we think it is interesting to see what can be gained in a continuous-time framework and what additional difficulties arise.

5.1.1 Methodology

We begin with a general presentation of the operational approach to the semantics of concurrent systems [Plotkin 81].

There is a set of agents $p \in P$ which may perform actions $a \in A$. The semantics of agents is given by a set of binary relations $\xrightarrow{a}$ over P (for all $a \in A$). When $p \xrightarrow{a} p'$ we say that the agent p

performs the action a and becomes the agent p'.

The set P is defined as the free algebra over a signature Σ, i.e. P is the set of syntactic expressions for agents which are built from a set of operators in Σ. Some structure is usually imposed on the set A, e.g. an abelian monoid or group.

An operational semantics is defined which specifies the relations $\xrightarrow{a}$. These relations are expressed in a syntax-directed way: for every op ε Σ we say how to derive the reductions "$op(p_1,\ldots,p_n) \xrightarrow{a} q$" of p from the reductions of $p_1, \ldots ,p_n$.

A congruence relation "$\sim$" is defined over P together with some useful proof method for proving properties like $p \sim q$. This congruence relation defines a Σ-algebra $P/\sim$ which is the semantics of agents.

A set of algebraic laws holding in $P/\sim$ is derived. This set of laws is particularly interesting when it is complete for finite expressions in P, i.e. when the congruence $\sim$ is the same as the congruence generated by the laws. This means that two finite agents p,q ε P <u>are</u> equivalent if and only if they can be <u>proved</u> equivalent using the laws. Gordon Plotkin remarked that this property does not hold in general for infinite agents (e.g. recursive agents) but it can lead to a powerful proof system when coupled with an induction theorem.

5.1.2 The Action Monoid

Agents progress by performing actions. Actions are denoted by the letters a,b,c and d, and the set of all the actions is A. Actions can be performed concurrently, so we denote by $a \cdot b$ (or simply ab) the simultaneous occurrence of actions a and b. We also admit a <u>neutral</u> action 1, so that $\langle A, \cdot, 1 \rangle$ is an abelian monoid, i.e.:

Unit: $a\ 1 = a$

Commutativity: $a\ b = b\ a$

Associativity: $a\ (b\ c) = (a\ b)\ c$

Communication between agents can be modelled by requiring A to be a commutative group $\langle A, \cdot, 1, \bar{\ } \rangle$, where

Inverse: $a\ \bar{a} = 1$

We may require A to be a free group over a set N of atomic actions (generators) denoted by greek letters $\alpha, \beta, \gamma, \delta$.

A successful communication between two agents is represented by the matching of two actions a and $\bar{a}$. The fact that $a\bar{a} = 1$ means that communication involves exactly two agents, that the respective communication capabilities are consumed during the process and that an external observer is unable to tell which communication took place (he can only observe 1). Note that communication here means simple synchronisation, and does not involve the passage of values.

5.1.3 Time

The central idea in real time agents is the explicit use of time information when expressing the behaviour of agents. Time is assumed to be dense, i.e. for every two instants t_0, t_1 it is always possible to find an instant t such that $t_0 < t < t_1$. The real numbers are the obvious choice for a dense domain of time, but rational numbers will also do.

We shall formalise the idea of observing a real time system <u>during</u> intervals of time, (i.e. not observing <u>at</u> time instants) and we want to rule out the possibility of observing zero-length actions. Hence the variables denoting time will range over a dense domain $\mathbb{K}$ (for Kronos) = $\mathbb{R}^+$, that is the set of <u>strictly</u> positive real numbers. The letters t,u,v,w,x,y,z will range over $\mathbb{K}$.

5.2 Deterministic Agents

We first examine agents which are deterministic, in the informal sense that every agent has a unique possible development in time. A formal property corresponding to the idea of determinism will be examined later.

5.2.1 Signature

We start with a very simple set of operators to form our expressions. This set will be gradually expanded making clear what results extend from the smaller signatures to the larger ones.

Our initial signature Σ^D (where D stands for deterministic) consists of: a constant $\mathbf{1}$ representing the neutral agent always performing the neutral action 1; a unary prefix operator a[t]: which represents the act of performing the action a for an interval of time t; and the binary infix operator $\times$ representing the synchronous composition (coexistence) of two agents.

$$\begin{array}{lll} \mathbf{1} & \in \Sigma^D_0 & \\ a[t]: & \in \Sigma^D_1 & \text{for all } a\in A \text{ and } t\in\mathbb{K} \\ \times & \in \Sigma^D_2 & \end{array}$$

Finally an agent (denoted by p,q,r,s) is an expression built over the signature $\Sigma^D \triangleq \{\Sigma^D_0, \Sigma^D_1, \Sigma^D_2\}$. The set of agents P^D is the free algebra over Σ^D.

5.2.2 Operational Semantics

Now we shall specify how our agents <u>behave</u>, by defining a set of binary relations $\xrightarrow[t]{a}$ (for $a\in A$ and $t\in\mathbb{K}$) over P^D. We read $p\xrightarrow[t]{a}q$ as "p moves to q performing a for an interval t", or "p takes t to move under a to q".

The reduction rules for deterministic agents are as follows:

[**1**→]	$\mathbf{1} \xrightarrow[t]{1} \mathbf{1}$
[a[]→]	$a[t]{:}p \xrightarrow[t]{a} p$
[a[]a[]→]	$a[t{+}u]{:}p \xrightarrow[t]{a} a[u]{:}p$
[X→]	$\dfrac{p \xrightarrow[t]{a} p' \qquad q \xrightarrow[t]{b} q'}{p \times q \xrightarrow[t]{ab} p' \times q'}$

Rule [**1**→] asserts that **1** moves under 1 for an arbitrary interval t to produce **1** again.

Rule [a[]→] says that a[t]:p takes t to move under a to p, with t>0.

Rule [a[]a[]→] has to do with the density of time; it says that after an interval t, a[t+u]:p has only reached a[u]:p. Note that it is possible to split actions at arbitrary points, but this is done consistently so that the final outcome remains the same.

Rule [X→] gives meaning to the coexistence of two agents: if p takes t to move under a to p' and q takes t to move under b to q', then pXq takes t (the same t) to move under a·b to p'Xq'. Note that if q is of the form b[t+u]:q", we can use [a[]a[]→] to get a t-derivation of q, so that we can use [X→].

This set of operational rules enjoys two fundamental properties:

Lemma 5.1 (Density Lemma) $p \xrightarrow[t+u]{a} r \Rightarrow \exists q.\ p \xrightarrow[t]{a} q,\ q \xrightarrow[u]{a} r$

Proof Induction on the structure of the derivation of $p \xrightarrow[t+u]{a} r$ □

Lemma 5.2 (Persistency Lemma) $\forall p,t.\ \exists p_1..p_n, a_1..a_n, t_1..t_n.$

$\Sigma_i t_i = t$ and $p \xrightarrow[t_1]{a_1} p_1 \ldots \xrightarrow[t_n]{a_n} p_n$

Proof Induction on the structure of p. The case $p=p'Xp''$ needs the Density Lemma $\square$

We shall abandon the persistency lemma later, but density is fundamental for all the systems we study. When adding a new operator to our signature, most of the results for the old signature extend to the new one, provided that density is preserved.

5.2.3 Observation

Agents will be observed by considering the sequences of actions they can perform. If the agents p and q are in the relation $p \xrightarrow[t]{a} q$, and q and r are in the relation $q \xrightarrow[u]{b} r$, then we can consider the composition of the relations $\xrightarrow[t]{a}$ and $\xrightarrow[u]{b}$ (denoted $\xrightarrow[t]{a}\circ\xrightarrow[u]{b}$) so that p and r are in the relation $p\ (\xrightarrow[t]{a}\circ\xrightarrow[u]{b})\ r$.

Definition 5.1

$\xrightarrow[t]{a}\circ\xrightarrow[u]{b} \triangleq \{\langle p,r\rangle \mid \exists q.\ \langle p,q\rangle \varepsilon \xrightarrow[t]{a} \text{ and } \langle q,r\rangle \varepsilon \xrightarrow[u]{b}\}$ $\square$

We write $\xrightarrow[(t_1\ldots t_n)]{(a_1\ldots a_n)}$ for $\xrightarrow[t_1]{a_1}\circ\ldots\circ\xrightarrow[t_n]{a_n}$ $(n>0)$. Moreover a sequence of actions is denoted by

$$\tilde{a} \triangleq (a_1,\ldots,a_n) \quad \text{with} \quad \#\tilde{a} \triangleq n$$

and a sequence of time intervals by

$$\tilde{t} \triangleq (t_1,\ldots,t_n) \quad \text{with} \quad \#\tilde{t} \triangleq n \quad \text{and} \quad \Sigma\tilde{t} \triangleq \Sigma_{1\le i\le n} t_i.$$

We want to observe actions in such a way that, for example, the sequences

$$\xrightarrow[(1,1)]{(a,a)} \quad \text{and} \quad \xrightarrow[(2)]{(a)}$$

are indistinguishable. This can be done by considering similar sequences in the following informal sense:

$$\xrightarrow[(2,2,2,2)]{(a,b,b,b)} \text{ is similar to } \xrightarrow[(1,1,3,3)]{(a,a,b,b)}$$

$$\xrightarrow[(1,2)]{(a,b)} \text{ is } \underline{\text{not}} \text{ similar to } \xrightarrow[(2,1)]{(a,b)}$$

Definition 5.2 Similarity is the least equivalence relation, $\simeq$, between relations, $\xrightarrow[\tilde{t}]{\tilde{a}}$, such that:

(i) If $a_1 = \ldots = a_n = b_1 = \ldots = b_m$ and $\Sigma\tilde{t} = \Sigma\tilde{u}$

then $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[\tilde{u}]{\tilde{b}}$

(ii) If $\xrightarrow[\tilde{t}']{\tilde{a}'} \simeq \xrightarrow[\tilde{u}']{\tilde{b}'}$ and $\xrightarrow[\tilde{t}'']{\tilde{a}''} \simeq \xrightarrow[\tilde{u}'']{\tilde{b}''}$

then $\xrightarrow[\tilde{t}']{\tilde{a}'} \circ \xrightarrow[\tilde{t}'']{\tilde{a}''} \simeq \xrightarrow[\tilde{u}']{\tilde{b}'} \circ \xrightarrow[\tilde{u}'']{\tilde{b}''}$

□

Note that if $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[\tilde{u}]{\tilde{b}}$ then $\Sigma\tilde{t} = \Sigma\tilde{u}$.

The following abbreviation will be used:

Definition 5.3 $p \xrightarrow[\tilde{t}]{\tilde{a}}{}^{s} q \Leftrightarrow \exists \xrightarrow[\tilde{t}']{\tilde{a}'} \simeq \xrightarrow[\tilde{t}]{\tilde{a}}$ such that $p \xrightarrow[\tilde{t}']{\tilde{a}'} q$ □

We can also talk about finer and coarser sequences and the meet of two similar sequences:

Definition 5.4 $\xrightarrow[\tilde{t}]{\tilde{a}}$ is finer than $\xrightarrow[\tilde{u}]{\tilde{b}}$ when $\xrightarrow[\tilde{t}]{\tilde{a}} \leq \xrightarrow[\tilde{u}]{\tilde{b}}$, where $\leq$ is the least relation satisfying:

(i) $\xrightarrow[(t_1 \ldots t_n)]{(a \ldots a)} \leq \xrightarrow[\Sigma_i t_i]{a}$

(ii) If $\xrightarrow[\tilde{t}']{\tilde{a}'} \leq \xrightarrow[\tilde{u}']{\tilde{b}'}$ and $\xrightarrow[\tilde{t}'']{\tilde{a}''} \leq \xrightarrow[\tilde{u}'']{\tilde{b}''}$

then $\xrightarrow[\tilde{t}']{\tilde{a}'} \circ \xrightarrow[\tilde{t}'']{\tilde{a}''} \leq \xrightarrow[\tilde{u}']{\tilde{b}'} \circ \xrightarrow[\tilde{u}'']{\tilde{b}''}$

□

Definition 5.5 $\xrightarrow[\tilde{t}]{\tilde{a}}$ is coarser than $\xrightarrow[\tilde{u}]{\tilde{b}}$ (written $\xrightarrow[\tilde{t}]{\tilde{a}} \geq \xrightarrow[\tilde{u}]{\tilde{b}}$) iff $\xrightarrow[\tilde{u}]{\tilde{b}} \leq \xrightarrow[\tilde{t}]{\tilde{a}}$ □

Theorem 5.1

The relation $\leq$ defines a partial order over the set of transition relations $\xrightarrow[\tilde{t}]{\tilde{a}}$. Moreover:

(i) If $\xrightarrow[\tilde{t}]{\tilde{a}} \leq \xrightarrow[\tilde{u}]{\tilde{b}}$ then $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[\tilde{u}]{\tilde{b}}$

(ii) If $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[(u)]{(b)}$ then $\xrightarrow[\tilde{t}]{\tilde{a}} \leq \xrightarrow[(u)]{(b)}$

(iii) The meet (greatest lower bound) of two similar sequences $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[\tilde{u}]{\tilde{b}}$ (written $\xrightarrow[\tilde{t}]{\tilde{a}} \wedge \xrightarrow[\tilde{u}]{\tilde{b}}$) exists and is unique.

Proof Directly from the definitions $\Box$

Finally the Density Lemma implies the following:

Lemma 5.3 (Refinement Lemma)

If $p \xrightarrow[\tilde{t}]{\tilde{a}} q$ and $\xrightarrow[\tilde{u}]{\tilde{b}} \leq \xrightarrow[\tilde{t}]{\tilde{a}}$ then $p \xrightarrow[\tilde{u}]{\tilde{b}} q$ $\Box$

Remark: the Refinement Lemma can also be expressed as:

$$\xrightarrow[\tilde{u}]{\tilde{b}} \leq \xrightarrow[\tilde{t}]{\tilde{a}} \text{ implies } \xrightarrow[\tilde{t}]{\tilde{a}} \subseteq \xrightarrow[\tilde{u}]{\tilde{b}}$$

Lemma 5.4 (Similarity Lemma)

(1) If $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[t]{1}$ then $\mathbf{1} \xrightarrow[\tilde{t}]{\tilde{a}} \mathbf{1}$

(2) If $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[t]{a}$ then $a[t]:p \xrightarrow[\tilde{t}]{\tilde{a}} p$

(3) If $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[t]{a}$ then $a[t+u]:p \xrightarrow[\tilde{t}]{\tilde{a}} a[u]:p$

(4) If $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[t]{a}$ and $\xrightarrow[\tilde{t}]{\tilde{b}} \simeq \xrightarrow[t]{b}$ then $\exists \xrightarrow[\tilde{t}]{\tilde{c}} \simeq \xrightarrow[t]{ab}$.

$$\frac{p \xrightarrow[\tilde{t}]{\tilde{a}} p' \qquad q \xrightarrow[\tilde{t}]{\tilde{b}} q'}{p \chi q \xrightarrow[\tilde{t}]{\tilde{c}} p' \chi q'}$$

Proof Trivial, except that (4) uses the Refinement Lemma $\Box$

5.2.4 Equivalence

Informally, the behaviour of agents is given by their reduction chains, and we want to regard as equivalent agents which have the "same" reduction chains (i.e. which perform the "same" actions) even if they are syntactically different as members of P^D. After having defined a congruence relation ~ over P^D so that p~q iff they perform the same actions, we can then take the equivalence class of p in $P^D/\sim$ as the semantics of p.

We are going to define the following equivalence:
p is equivalent to q iff every time that p can reduce under a sequence of actions $\xrightarrow[\tilde{t}]{\tilde{a}}$ to p', then q can reduce by a similar sequence $\xrightarrow[\tilde{t}]{\tilde{a}}{}^{s}$ to some q' equivalent to p' (and vice versa). This equivalence is called smooth equivalence because it ignores the "density" of individual actions and only considers their coarse result.

We first define a formula $\mathbb{D}(\approx)$ parametrically in an arbitrary relation $\approx$ over P^D:

Definition 5.6

$\mathbb{D}(\approx) \triangleq$

$p \approx q$ iff $\forall\ a \in A,\ t \in \mathbb{K}.$

both $p \xrightarrow[t]{a} p' \Rightarrow \left(\exists q'.\ q \xrightarrow[t]{a}{}^{s} q' \text{ and } p' \approx q'\right)$

and $q \xrightarrow[t]{a} q' \Rightarrow \left(\exists p'.\ p \xrightarrow[t]{a}{}^{s} p' \text{ and } p' \approx q'\right)$

□

Definition 5.7 **Smooth equivalence** (~) is the maximal fixpoint of the equation $\approx\ = \mathbb{D}(\approx)$ in the lattice of binary relations over P^D □

Theorem 5.2 (Park's Induction Principle [Park 81])

$p \sim q$ iff $\exists\ R \subseteq P^D \times P^D.$

(i) $\langle p,q\rangle \in R$

(ii) $R \subseteq \mathbb{D}(R)$

□

Condition (ii) can be written more explicitly as:

$\langle p,q\rangle \in R \Rightarrow$

(ii') $\forall p \xrightarrow[t]{a} p'.\ \exists\langle p',q'\rangle \in R.\ q \xrightarrow[t]{a}{}^{s} q'$

(ii") $\forall q \xrightarrow[t]{a} q'.\ \exists\langle p',q'\rangle \in R.\ p \xrightarrow[t]{a}{}^{s} p'$

Theorem 5.3

(i) $\sim$ is an equivalence relation.

(ii) $\sim$ is a congruence with respect to $\Sigma^D = \{\mathbf{1},\ a[t]:,\ \chi\}$.

(iii) $P^D/\sim$ is a Σ^D-algebra.

Proof

(i) Easily verified.

(ii) We have to show that for every Σ^D-context $C[x]$:

$p \sim q \Rightarrow C[p] \sim C[q]$

It is enough to show (by Park's induction) that:

(1) $p \sim q \Rightarrow a[t]:p \sim a[t]:q$.

(2) $p \sim q \Rightarrow p\chi r \sim q\chi r$ and $r\chi p \sim r\chi q$.

For (1) take $R \triangleq \{\langle a[t]:p,a[t]:q\rangle \mid p \sim q\rangle\} \cup \sim$.

(1.base) $\langle a[t]:p,a[t]:q\rangle \in R$ by definition;

(1.step) if $r\in R$ because $r\in\sim$ then $r\in\mathbb{D}(R)$ by definition of $\sim$;

if $\langle a[t]:p,a[t]:q\rangle\in R$ where $p\sim q$, suppose $a[t]:p\xrightarrow[u]{b}P$:

it may only have been derived from $[a[]\rightarrow]$ or $[a[]a[]\rightarrow]$.

(1.step.$[a[]\rightarrow]$) $a[t]:p\xrightarrow[t]{a}p$ with $b=a$, $u=t$, $P=p$.

By $[a[]\rightarrow]$: $a[t]:q\xrightarrow[t]{a}q$ with $\langle p,q\rangle \in R$ by hypothesis.

(1.step.$[a[]a[]\rightarrow]$) $a[t]:p\xrightarrow[u]{a}a[t-u]:p$ with $b=a$, $u<t$, $P=a[t-u]:p$.

By $[a[]a[]\rightarrow]$: $a[t]:q\xrightarrow[u]{a}a[t-u]:q$ with $\langle a[t-u]:p,b[t-u]:q\rangle \in R$.

The rest is symmetric, for $a[t]:q\xrightarrow[u]{b}Q$

For (2) the proof follows the same theme, with $R \triangleq \{\langle p\chi r,q\chi r\rangle \mid p \sim q\rangle\} \cup \sim$ (and symmetrically in the second case), using the similarity lemma, and hence depending on the density lemma.

(iii) This is a standard algebraic result, based on (ii).

$\square$

5.2.5 Algebraic Laws

The following holds:

$[\times\mathbf{1}]$	$p \times \mathbf{1} \sim p$
$[\times]$	$p \times q \sim q \times p$
$[\times\times]$	$p \times (q \times r) \sim (p \times q) \times r$
$[1[]\mathbf{1}]$	$1[t]:\mathbf{1} \sim \mathbf{1}$
$[a[]a[]]$	$a[t]:a[u]:p \sim a[t+u]:p$
$[a[]\times]$	$a[t]:p \times b[t]:q \sim ab[t]:(p \times q)$

All the laws can be proved smoothly by Park's induction. Both the congruence property for $\times$ and the factorisation law $[a[]\times]$ depend only on the density lemma; whenever we modify our signature we need only to make sure that the density lemma still holds.

The following results tell us that our set of laws is rich and consistent:

Definition 5.8 Let us denote by $\equiv$ the congruence defined by the set of laws $[\times\mathbf{1}]$ ... $[a[]\times]$. We say that p is convertible to q iff $p \equiv q$ □

Theorem 5.4 (Soundness)

$p \equiv q \Rightarrow p \sim q$

Proof

Induction on the derivation of $p \equiv q$, using the fact that $\sim$ is a congruence and the laws are valid □

Definition 5.9 $S_{i\leq n}\ a_i[t_i]:p \ \triangleq\ a_1[t_1]:\ldots a_n[t_n]:p \ (n\geq 0)$ □

Definition 5.10 An agent is in sequence form if it is of the form

$S_{i\leq n}\ a_i[t_i]:\mathbf{1}$ □

Definition 5.11 An agent is in normal form if it is in sequence form $S_{i\leq n}\ a_i[t_i]:\mathbf{1}$ with $(n>0 \Rightarrow a_n \neq 1)$ and $(n\geq 2 \Rightarrow \forall i<n.\ a_i \neq a_{i+1})$ □

Theorem 5.5 (Normal Forms)

(i) Every agent is convertible to a sequence form.

(ii) Every sequence form is convertible to a normal form.

(iii) Every agent has a unique normal form.

Proof Simple inductions on the structure of terms □

Theorem 5.6 (Completeness)

$$p \sim q \Rightarrow p \equiv q$$

Proof

First prove that for p', q' in normal form, $p' \sim q' \Rightarrow p' \equiv q'$ by induction on the structure of p' and q' (this is easy because of the simple structure of normal forms: we even have $p' \sim q' \Rightarrow p' = q'$). In general, by the normal form theorem, p and q have respective normal forms p' and q' (so that $p \equiv p'$ and $q \equiv q'$). By soundness $p' \sim p \sim q \sim q'$. So by the first part of the proof $p' \equiv q'$. Hence $p \equiv p' \equiv q' \equiv q$ □

5.2.6 Determinacy

We said that our agents are deterministic; in fact there are very strong properties that agents must obey in reductions. The most important ones are collected in the following action lemmas:

Lemma 5.5 (Action Lemmas)

If $\mathbf{1} \xrightarrow[(t)]{(a)}{}^{s} p$ then $a = 1$, $p = \mathbf{1}$

If $a[t]{:}p \xrightarrow[u]{b} q$ then $u \leq t$

If $a[t]{:}p \xrightarrow[(u)]{(b)}{}^{s} q$ then $b = a$

If $a[t]{:}p \xrightarrow[(u)]{(b)}{}^{s} q$ and $u > t$ then $p \xrightarrow[(u-t)]{(b)}{}^{s} q$

If $a[t]{:}p \xrightarrow[(t)]{(a)}{}^{s} q$ then $p = q$

If $a[t]{:}p \xrightarrow[(u)]{(b)}{}^{s} q$ and $u < t$ then $q = a[t-u]{:}p$

If $p' \chi p'' \xrightarrow[t]{a} q$ then $\exists a', a'', q', q''.$

$p' \xrightarrow[t]{a'} q'$, $p'' \xrightarrow[t]{a''} q''$, $a = a'a''$, $q = q' \chi q''$

$\square$

These action lemmas imply, by simple structural induction, the following important properties:

Theorem 5.7 (Vertical Determinacy)

$p \xrightarrow[t]{a} q$ and $p \xrightarrow[u]{b} r \Rightarrow a = b$ $\square$

Theorem 5.8 (Horizontal Determinacy)

(i) If $p \xrightarrow[\tilde{t}]{\tilde{a}} q$, $p \xrightarrow[\tilde{u}]{\tilde{b}} r$ and $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[\tilde{u}]{\tilde{b}}$ then $q = r$

(ii) If $p \sim q$, $p \xrightarrow[\tilde{t}]{\tilde{a}} p'$, $q \xrightarrow[\tilde{u}]{\tilde{b}} q'$ and $\xrightarrow[\tilde{t}]{\tilde{a}} \simeq \xrightarrow[\tilde{u}]{\tilde{b}}$ then $p' \sim q'$

$\square$

In this formal sense, our agents are completely deterministic, and we can also see that it is possible to introduce two orthogonal kinds of nondeterminism. This will be done in the next section.

5.3 Nondeterministic Agents

5.3.1 Signature

Let us now consider the signature

$0 \quad \varepsilon\ \Sigma_0^{ND}$

$a(t): \quad \varepsilon\ \Sigma_1^{ND}$ for all $a\varepsilon A$ and $t\varepsilon \mathbb{K}$

$+ \quad \varepsilon\ \Sigma_2^{ND}$

The agent 0 has no actions, not even neutral actions. When a system reaches the state 0, a catastrophe occurs and time ceases to flow; hence 0 is called a disaster.

The prefix operator a(t): represents the act of performing the action a for a positive interval of length <u>at most</u> t; we shall say that this operator introduces horizontal continuous nondeterminism in the sense that arrows can be stretched horizontally according to the duration of a(t):.

The binary operator + represents the choice of two possible behaviours, and it introduces vertical discrete nondeterminism; the sense of these adjectives may be made clear by the following diagram, where the action monoid is on the vertical axis and time is on the horizontal axis. The behaviour of an agent is then a (possibly discontinuous) trajectory in this space.

Figure 5.1

5.3.2 Operational Semantics

There are no axioms for 0.

The agent a(t):p takes time $v \leq t$ to move under a to p, and a(t+u):p takes time $v \leq t$ to move under a to p + a(u):p. Hence a(t):p can choose at any move to shorten its life span by some amount;

moreover at any point in time it can stop its a-action and start executing p.

If p takes t to move under a to p', then p+q may move under a to p' taking time t, or else if q takes u to move under b to q', then p+q may move under b to q' taking time u.

$[a()\rightarrow]$	$a(t):p \xrightarrow[v]{a} p$	$v \leq t$
$[a()()\rightarrow]$	$a(t+u):p \xrightarrow[v]{a} p + a(u):p$	$v \leq t$
$[+\rightarrow]$	$\dfrac{p \xrightarrow[t]{a} p'}{p+q \xrightarrow[t]{a} p'}$,	$\dfrac{q \xrightarrow[u]{b} q'}{p+q \xrightarrow[u]{b} q'}$

5.3.3 **Algebraic Laws**

Applying the same definition of smooth equivalence to the new signature and operational semantics, we obtain the following holding in P^{ND}:

$[+0]$	$p + 0 \sim p$
$[+p]$	$p + p \sim p$
$[+]$	$p + q \sim q + p$
$[++]$	$p + (q + r) \sim (p + q) + r$
$[a()+]$	$a(t+u):p \sim a(t+u):p + a(t):p$
$[a()a()]$	$a(t+u):p \sim a(t):(p+a(u):p)$

5.3.4 **Combined Calculus**

We now merge the two signatures into $\Sigma^0 \triangleq \Sigma^D \cup \Sigma^{ND}$ with P^0 being the free Σ^0-algebra. We have to abandon the persistency lemma, because of the presence of 0. The density lemma, however, still hods:

Lemma 5.6 (Density Lemma) $p \xrightarrow[t+u]{a} r \Rightarrow \exists q.\ p \xrightarrow[t]{a} q,\ q \xrightarrow[u]{a} r$ □

Extending the usual definition of equivalence to Σ^0:

Theorem 5.9

(i) $\sim$ is an equivalence relation.

(ii) $\sim$ is a congruence with respect to Σ^0

(iii) $P^0/\sim$ is a Σ^0-algebra $\Box$

We obtain a new set of laws describing the interactions between the two smaller signatures:

[$\times$0] $p \times 0 \sim 0$

[$\times$+] $p \times (q + r) \sim (p \times q) + (p \times r)$

[1()1] $1(t){:}\mathbf{1} \sim \mathbf{1}$

This does not give us a complete set of laws; we lack the distributivity of a(t): over $\times$ and some law relating a(t): to a[t]:.

Laws relating a(t): and $\times$ are called factorisation theorems. (The operator $\downarrow B$ used below is explained in the next section; the laws [FT2] and [FT4] hold also with all the $\downarrow B$ elided.)

[FT1] $(a(t){:}p \times b(t){:}q)\downarrow B \sim 0$ if $ab \notin B$

[FT2] $(a(t){:}p \times b(t){:}q)\downarrow B \sim (ab(t){:}(p \times q))\downarrow B$

if either $\forall u{<}t.\ (p\times(q+b(u){:}q))\downarrow B \sim (p\times q)\downarrow B$

or $\forall u{<}t.\exists v{\le}u.\ (p\times(q+b(u){:}q))\downarrow B \sim (p\times q+a(v){:}p\times b(v){:}q)\downarrow B$

and either $\forall u{<}t.\ ((p+a(u){:}p)\times q)\downarrow B \sim (p\times q)\downarrow B$

or $\forall u{<}t.\exists v{\le}u.\ ((p+a(u){:}p)\times q)\downarrow B \sim (p\times q+a(v){:}p\times b(v){:}q)\downarrow B$

and either $\forall u{<}t.\ ((p+a(u){:}p)\times(q+b(u){:}q))\downarrow B \sim (p\times q)\downarrow B$

or $\forall u{<}t.\exists v{\le}u.\ ((p+a(u){:}p)\times(q+b(u){:}q))\downarrow B \sim (p\times q+a(v){:}p\times b(v){:}q)\downarrow B$

and either $\forall u{<}t.\ (p\times q+a(u){:}p\times b(u){:}q)\downarrow B \sim (p\times q)\downarrow B$

or $\forall u{<}t.\exists v{\le}u.\ (p\times q+a(u){:}p\times b(u){:}q)\downarrow B \sim ((p+a(v){:}p)\times(q+b(v){:}q))\downarrow B$

[FT3] $(a(t){:}p \times b[t]{:}q)\downarrow B \sim 0$ if $ab \notin B$

[FT4] $(a(t):p \between b[t]:q)\downarrow B \sim (ab[t]:(p \between q))\downarrow B$

if $\forall u<t.\ (a(u):p \between b[u]:q)\downarrow B \sim (p \between b[u]q)\downarrow B$

and $\forall u<t. \exists v \leq u.\ (a(u):p \between b[u]:q)\downarrow B \sim ((p+a(v):p) \between b[u]:q)\downarrow B$

These laws constitute a major departure from the equational style we have observed up to now, and may be an indication that we have not chosen the best possible set of primitive operators. On the other hand they seem to reflect rather faithfully the complex relationships between a synchronous deterministic world (Σ^D) and an asynchronous nondeterministic one (Σ^{ND}), and we could not devise a simpler formulation. The factorisation theorems can usually be much simplified in practical situations (e.g. replacing "$\forall u<t$" by "$\forall u$"), and they turn out to be very useful in proving equational laws of interesting derived operators, as we shall see later.

5.4 Communication

In order to model communication, our action monoid A will be assumed to be an abelian group $\langle A, \cdot, 1, \bar{\ } \rangle$ freely generated by a set of **names** N. For $B \subseteq A$ we define $\bar{B} \triangleq \{\bar{a} \mid a \in B\}$; then $\bar{N}$ is the set of **conames** and $L \triangleq N \cup \bar{N}$ is the set of labels or **atomic actions**.

Communication occurs when two complementary actions occur together, like in

$$a[t]:p \times \bar{a}[t]:q \sim a\bar{a}[t]:(p \times q) \sim 1[t]:(p \times q)$$

In a composition $p \times q$ we implicitly establish communication channels between all the complementary actions of p and q. Since this connections are implicit in the naming conventions of actions, we need some operator to control this naming activity, so that we can prepare agents for purposeful compositions.

5.4.1 Restriction

The **restriction** operator $\downarrow B$, for $B \subseteq A$ and $1 \in B$ is used to extract a subset of the possible actions of an agent, inhibiting the rest of the actions.

$$[\downarrow\longrightarrow] \quad \frac{p \xrightarrow[t]{a} q}{p\downarrow B \xrightarrow[t]{a} q\downarrow B} \quad \text{if } a \in B$$

Thus $p\downarrow B$ can only perform actions which are in B. The action 1 is never inhibited by definition; it represents the possible anonymous occurrence of a communication event inside p.

It should be stressed that restriction is not a hiding of some internal actions, but it represents their inhibition, the impossibility of their occurrence in isolation (they may occur if complemented). Restriction can be used to drive and determine the internal behaviour of an agent, as in the following example:

$p \triangleq a[t]:\mathbf{1} + b[t]:\mathbf{1}$

$p{\downarrow}\{1,a\} \sim a[t]:\mathbf{1}$

$p{\downarrow}\{1,b\} \sim b[t]:\mathbf{1}$

where in each case one of the two sides of + is forced. This idea can be used to channel communication, as in:

$p \triangleq (a[t]:\mathbf{1} + b[t]:\mathbf{1}) \times \bar{a}\bar{b}[t]:\mathbf{1}$

$p{\downarrow}\{1,\bar{a}\} \sim \bar{a}[t]:\mathbf{1}$

$p{\downarrow}\{1,\bar{b}\} \sim \bar{b}[t]:\mathbf{1}$

where in the first case a b-communication, and in the second case an a-communication, are forced.

Restriction can induce disaster:

$(a[t]:p){\downarrow}\{1\} \sim 0$

but can also avert disaster:

$(a[t]:0 + b[u]:\mathbf{1}){\downarrow}\{1,b\} \sim b[u]:\mathbf{1}$

Here are the laws of restriction:

[↓] $\quad p{\downarrow}B \sim p \quad$ if $p \in P_B$

[↓a[]:] $$(a[t]:p){\downarrow}B \sim \begin{cases} a[t]:(p{\downarrow}B) & \text{if } a \in B \\ 0 & \text{otherwise} \end{cases}$$

[↓a():] $$(a(t):p){\downarrow}B \sim \begin{cases} a(t):(p{\downarrow}B) & \text{if } a \in B \\ 0 & \text{otherwise} \end{cases}$$

[↓+] $\quad (p + q){\downarrow}B \sim p{\downarrow}B + q{\downarrow}B$

These laws are also valid if we only assume A to be a monoid, but note the absence of a law for $\times$. This is better studied in the case of the next operator we examine.

The delabelling operator $p\backslash\alpha$ is a particular case of restriction. It is used to restrict over a set B in which some atomic action α and its complement $\bar{\alpha}$ never appear as factors; then $p{\downarrow}B$ means hereditarily removing all the α-communications capabilities of p.

Definition 5.12 p has sort B (or p has B) if whenever $p \xrightarrow[t_0]{a_0} \ldots \xrightarrow[t_n]{a_n} p'$ $(n \geq 0)$ and $p' \xrightarrow[t]{a} p''$ then $a \varepsilon B$ □

Definition 5.13 If $B \subseteq L$ then B^* is the submonoid of A generated by B □

We can now define delabelling as:

$$p\backslash\alpha \triangleq p{\downarrow}(L-\{\alpha,\bar{\alpha}\})^*$$

with laws:

[\]	$p{\downarrow}\alpha \sim p$	if p has B and $\alpha,\bar{\alpha} \notin B$
[a[]:\]	$(a[t]:p)\backslash\alpha \sim \begin{cases} 0 \text{ if } \alpha \text{ or } \bar{\alpha} \text{ is a factor of } a \\ a[t]:(p\backslash\alpha) \text{ otherwise} \end{cases}$	
[a():\]	$(a(t):p)\backslash\alpha \sim \begin{cases} 0 \text{ if } \alpha \text{ or } \bar{\alpha} \text{ is a factor of } a \\ a(t):(p\backslash\alpha) \text{ otherwise} \end{cases}$	
[X\]	$(p \times q)\backslash\alpha \sim p\backslash\alpha \times q\backslash\alpha$	if p has B, q has C and $\alpha,\bar{\alpha} \notin B\cap\bar{C}$
[+\]	$(p + q)\backslash\alpha \sim p\backslash\alpha + q\backslash\alpha$	
[\\]	$p\backslash\alpha\backslash\beta \sim p\backslash\beta\backslash\alpha$	

5.4.2 Morphisms

We need a way of renaming actions, so that we can easily set up communication channels. The most general form of renaming is called a morphism $p\{\phi\}$ where $\phi: A \to A$ is a monoid homomorphism:

$$[\{\phi\}\to] \quad \frac{p \xrightarrow[t]{a} p'}{p\{\phi\} \xrightarrow[t]{\phi(a)} p'\{\phi\}}$$

We shall write $p\{\alpha_i/\beta_i\}$ for the unique monoid morphism renaming the generators β_i to α_i in p and leaving the other generators unchanged. Here are the laws for morphisms:

[{}]	$p\{\} \sim p$
[{ϕ}]	$p\{\phi\} \sim p\{\phi'\}$ if $p \varepsilon P_C$ and $\phi(a)=\phi'(a)$ for all $a \varepsilon C$
[{ϕ}{ϕ'}]	$p\{\phi\}\{\phi'\} \sim p\{\phi' \circ \phi\}$
[a[]:{ϕ}]	$(a[t]:p)\{\phi\} \sim \phi(a)[t]:(p\{\phi\})$
[a():{ϕ}]	$(a(t):p)\{\phi\} \sim \phi(a)(t):(p\{\phi\})$
[X{ϕ}]	$(p \times q)\{\phi\} \sim p\{\phi\} \times q\{\phi\}$
[+{ϕ}]	$(p + q)\{\phi\} \sim p\{\phi\} + q\{\phi\}$
[↓{ϕ}]	$p{\downarrow}B\{\phi\} \sim p\{\phi\}{\downarrow}\phi(B)$

5.4.3 Delays

We want to be able to model agents in which the actions of an (output) port are the delayed copy of the actions of another (input) port. It is not enough to have a delay operator which delays a whole agent, because this means delaying all the (input and output) ports by the same amount.

Hence we define the operator $\Delta_t M \varepsilon \Sigma_1$ for any $M \subseteq N$ containing 1:

[$\Delta_t M \longrightarrow$] $$\frac{p \xrightarrow[t]{ab} q}{p\Delta_{t+u}M \xrightarrow[t]{a} (q\Delta_{t+u}M) \times (1[u]:b[t]:1)}$$

if $\text{factors}(a) \subseteq M \cup \bar{M}$ and $\text{factors}(b) \cap (M \cup \bar{M}) = \emptyset$

where factors(a) is the set of prime factors (generators) of a, i.e. not including 1.

Here are the laws for delays:

$[\Delta_t M]$	$p\Delta_t M \sim p$	if p has M
$[\Delta_t M\ a[u]:]$	$(ab[t]:p)\Delta_u M \sim a[t]:(p\Delta_u M) \times 1[u]:b[t]:\mathbf{1}$	
	if $\text{factors}(a) \subseteq M \cup \bar{M}$ and $\text{factors}(b) \cap (M \cup \bar{M}) = \emptyset$	
$[\Delta_u M \times]$	$(p \times q)\Delta_t M \sim p\Delta_t M \times q\Delta_t M$	
$[\Delta_u M+]$	$(p + q)\Delta_t M \sim p\Delta_t M + q\Delta_t M$	
$[\Delta_t M \Delta_u M]$	$p\Delta_t M\Delta_u M \sim p\Delta_{t+u} M$	
$[\Delta_t M \Delta_u M']$	$p\Delta_t M\Delta_u M' \sim p\Delta_u M'\Delta_t M$	

We shall show some example involving delays after having defined recursive agents.

5.5 Recursion

A recursive definition facility will now be introduced in our language. Its general form for a single recursive definition is:

$x \Leftarrow r$

where x is a variable and r is a context, i.e. a term possibly containing variables. We have the operational rule:

$$[\Leftarrow] \quad \frac{r \xrightarrow[t]{a} p}{x \xrightarrow[t]{a} p}$$

The effect of $\Leftarrow$ is equivalent to the introduction of a new constant $x \in \Sigma_0^D$, like in

$x \Leftarrow \mathbf{1} + a[t]:x$

To satisfy this definition, it is sufficient to find a p such that

$p \sim \mathbf{1} + a[t]:p$

because all our laws are valid up to equivalence. In fact it is easy to show that $[\Leftarrow]$ implies $x \sim p$.

But we still need to specify which particular x we want, when several of them are available, like in the definition $x \Leftarrow x$. To avoid this problem we restrict our admissible definitions to those having a unique solution up to equivalence; thus there is no doubt about which x we mean. We shall do so by imposing syntactic restrictions on the form of our definitions, or more precisely on the form of our sets of definitions (to take into account mutual recursion).

Definition 5.14 A **definition set** is a set of pairs $\{\langle x_i, r_i \rangle\}$, written $\{x_i \Leftarrow r_i\}$ or $\tilde{x} \Leftarrow \tilde{r}$, where the x_i are variables and the r_i are contexts □

Definition 5.15 $r\{\tilde{p}/\tilde{x}\}$ is the result of simultaneously replacing each x_j by p_j in r □

Definition 5.16 A **1-step expansion** of a definition set $\tilde{x} \Leftarrow \tilde{r}$ is obtained by replacing $x_i \Leftarrow r_i$ by $x_i \Leftarrow r_i\{r_j/x_j\}$ (for some i and j) in $\tilde{x} \Leftarrow \tilde{r}$. A **finite expansion** $\tilde{x} \Leftarrow \tilde{r}'$ of $\tilde{x} \Leftarrow \tilde{r}$ is an expansion obtained by a finite number of 1-step expansions. □

Lemma 5.7 If $\tilde{x} \Leftarrow \tilde{r}'$ is a finite expansion of $\tilde{x} \Leftarrow \tilde{r}$, then for all $\tilde{p}$,

$$\tilde{p} \sim \tilde{r}\{\tilde{p}/\tilde{x}\} \Leftrightarrow \tilde{p} \sim \tilde{r}'\{\tilde{p}/\tilde{x}\} \quad □$$

Definition 5.17 A variable x is **guarded** in a context r if all the occurrences of x are in subterms of r of the form a[t]:r' or a(t):r'. A context r is **guarded** if all its variables are guarded □

In order to have unique solutions for our definition sets, we need to exclude definition sets which expand indefinitely but only approach a finite limit (i.e. such that the sum of the durations of an infinite chain of actions is finite). Definition sets in which every infinite reduction chain has an infinite duration are called **persistent**.

Definition 5.18 A definition set $\{x_i \Leftarrow r_i\}$ is **guarded** if there is a finite expansion $\{x_i \Leftarrow r_i'\}$ such that each r_i' is guarded □

Definition 5.19 A definition set $\{x_i \Leftarrow r_i\}$ is **persistent** if whenever $\tilde{p}\sim\tilde{r}\{\tilde{p}/\tilde{x}\}$ then for all j, $p_j \xrightarrow[t]{a} P$ implies that there exists a finite expansion r_j^* of r_j such that $r_j^* \xrightarrow[t]{a}{}^s r_j'$ with $r_j'\{\tilde{p}/\tilde{x}\}\sim P$ ▯

Lemma 5.8 Every persistent definition set is guarded ▯

Lemma 5.9 Every finite guarded definition set is persistent ▯

Remark I: the previous lemma becomes false if we introduce "time-shrinking" operators in our signature, like:

$$\frac{p \xrightarrow[t]{a} q}{\Delta p \xrightarrow[u]{a} \Delta q}$$

In fact, take $r=\Delta a[t]:x$ and $p=\Delta a[t]:\mathbf{1}$, with $a\neq 1$; we can show:

$$\Delta a[t]:\mathbf{1} \sim \Delta a[t]:\Delta a[t]:\mathbf{1}$$

by Park's induction. Hence $p\sim r\{p/x\}$ is a solution for $\{x\Leftarrow r\}$ and $p\xrightarrow[t]{a}\mathbf{1}$, but for any expansion r^* of r we can only have reductions of the form

$$r^* \xrightarrow[t]{a}{}^s r' = \Delta a[t]:\ldots\Delta a[t]:x \text{ with } r'\{p/x\} \not\sim \mathbf{1}$$

Remark II: the following infinite definition set is guarded but not persistent:

$$\{Z_n \Leftarrow 1[n]:Z_{n/2} \mid n\varepsilon\mathbb{K}\}$$

In fact $\forall n.\ p_n=\mathbf{1}$ is a solution (suggested by Matthew Hennessy) and $p_1=\mathbf{1} \xrightarrow[2]{1} \mathbf{1}$ but there is no finite expansion Z_1^* of Z_1 such that $Z_1^* \xrightarrow[2]{1}{}^s \mathbf{1}$; in fact all the expansions of Z_1 have the form $Z_1^* = 1[1]:1[1/2]:1[1/4]:1[1/8]:\ldots$ where the sum of the durations of the actions is always less than 2.

Theorem 5.10 (Recursion Theorem)

Every persistent definition set $\tilde{x} \Leftarrow \tilde{r}$ has a unique solution up to $\sim$, i.e.: $p_i \sim r_i\{\tilde{p}/\tilde{x}\}$ and $q_i \sim r_i\{\tilde{q}/\tilde{x}\} \Rightarrow p_i \sim q_i$

Proof Let $\approx\ \triangleq\ \{\langle C\{\tilde{p}/\tilde{x}\},C\{\tilde{q}/\tilde{x}\}\rangle \mid C \text{ is a context}\}$

- $p_i \approx q_i$ (take $C = x_i$).

- $C\{\tilde{p}/\tilde{x}\} \xrightarrow[t]{a} P$ may hold because:

  either $C \xrightarrow[t]{a} C'$ with $P = C'\{\tilde{p}/\tilde{x}\}$;

  then also $C\{\tilde{q}/\tilde{x}\} \xrightarrow[t]{a} Q = C'\{\tilde{q}/\tilde{x}\}$, and $Q \approx P$

  or x_j is not guarded in C and $p_j \xrightarrow[t]{a} P$;

  then because $\tilde{r}$ is persistent there is a finite expansion r^*_j with $r^*_j \xrightarrow[t]{a}{}^s r'_j$ and $r'_j\{\tilde{p}/\tilde{x}\} \sim P$.

  Then also $r^*_j\{\tilde{q}/\tilde{x}\} \xrightarrow[t]{a}{}^s r'_j\{\tilde{q}/\tilde{x}\}$, and since $q_j \sim r_j\{\tilde{q}/\tilde{x}\} \sim r^*_j\{\tilde{q}/\tilde{x}\}$, we have $q_j \xrightarrow[t]{a}{}^s Q \sim r'_j\{\tilde{q}/\tilde{x}\}$.

  Hence $C\{\tilde{q}/\tilde{x}\} \xrightarrow[t]{a}{}^s Q$ with $Q \approx P$

□

Let us try a simple example of recursive definition:

$$\mathbb{K}_a \Leftarrow a[1]:\mathbb{K}_a$$

The agent $\mathbb{K}_a$ produces a-actions indefinitely. Using the recursion theorem it is possible to show that the "1" in the definition of $\mathbb{K}_a$ is non critical:

$$\begin{aligned} & a[t]:\mathbb{K}_a \\ & \quad \sim a[t]:a[1]:\mathbb{K}_a \\ & \quad \sim a[t+1]:\mathbb{K}_a \\ & \quad \sim a[1]:a[t]:\mathbb{K}_a \end{aligned}$$

Hence the equation

$$x \sim a[1]:x$$

is satisfied both by $\mathbb{K}_a$ (by definition) and by $a[t]:\mathbb{K}_a$, and by the recursion theorem we can conclude that:

$$\mathbb{K}_a \sim a[t]:\mathbb{K}_a \text{ for any } t$$

and also that, for a=1:

$$\mathbb{K}_1 \sim \mathbf{1}$$

Similarly from the equation:

$$\begin{aligned} & \mathbb{K}_a \times \mathbb{K}_b \\ & \quad \sim a[1]:\mathbb{K}_a \times b[1]:\mathbb{K}_b \\ & \quad \sim ab[1]:(\mathbb{K}_a \times \mathbb{K}_b) \end{aligned}$$

we can deduce:

$$\mathbb{K}_a \between \mathbb{K}_b \sim \mathbb{K}_{ab}$$

Going back to the delay operator, we can define a not gate with delay z in the following way:

$$Not'_z \Leftarrow \alpha_0\bar{\beta}_1[z]:Not'_z + \alpha_1\bar{\beta}_0[z]:Not'_z$$
$$Not_z \Leftarrow (Not'_z\Delta_z\bar{\beta}_i) \between \bar{\beta}_0[z]:\mathbf{1}$$

This not gate is not completely satisfactory, because it assume that its input signal changes at multiples of z (otherwise a disaster occurs). We shall see in a later section how to solve this problem of unsynchronised input by using nondeterministic guards.

5.6 Indefinite Actions and Delays

We shall see that one frequently uses nondeterministic guards a(t): only to prove that the particular t we use in not really important. This situation can be made systematic by defining an operator a.p (indefinite action) performing an action a for an arbitrary amount of time:

$$a.p \Leftarrow a(1):(p + a.p)$$

This particular choice of unit delay in the above definition makes no difference, as we have:

$$\begin{aligned}
a(t):(p + a.p) &\sim a(t):(p + a.p + a.p)\\
&\sim a(t):(p + a.p + a(1):(p + a.p))\\
&\sim a(t+1):(p + a.p) && \text{by } [a()+]\\
&\sim a(1):(p + a.p + a(t):(p + a.p)) && \text{by } [a()+]\\
a.p &\sim a(1):(p + a.p)\\
&\sim a(1):(p + a.p + a.p)
\end{aligned}$$

Hence $a.p \sim a(t):(p + a.p)$

by recursion theorem.

Moreover a.p enjoys the laws:

[1.0] $1.0 \sim \mathbf{1}$

[1.**1**] $1.\mathbf{1} \sim \mathbf{1}$

[a.] $a.p \sim a.(p + a.p)$

[a.Xb.] $a.p \times b.q \sim ab.(p \times q + a.p \times q + p \times b.q)$

Note the importance of the law [a.Xb.]; it allows us to equationally factorise actions in horizontally nondeterministic agents, which we could not do for the 'a(t):' operator. The law is proved by the factorisation theorems, thereby demonstrating some of their power. The above laws can be proved as follows:

$1.0 \sim 1(1):(0 + 1.0) \sim 1(1):(1.0)$

$\mathbf{1} \sim 1(1):\mathbf{1}$

Hence $1.0 \sim \mathbf{1}$

$1.\mathbf{1} \sim 1(1):(\mathbf{1} + 1.\mathbf{1})$

$\mathbf{1} \sim 1(1):\mathbf{1} \sim 1(1):(\mathbf{1} + \mathbf{1})$

Hence $1.\mathbf{1} \sim \mathbf{1}$

$a.(p + a.p) \sim a(1):(p + a.p + a.(p + a.p))$

$a.p \sim a(1):(p + a.p) \sim a(1):(p + a.p + a.p)$

Hence $a.p \sim a.(p + a.p)$

$a.p \times b.q \sim a(1):(p+a.p) \times b(1):(q+b.q)$

$\quad \sim ab(1):((p+a.p) \times (q+b.q))$ (*)

$\quad \sim ab(1):(p \times q + a.p \times q + p \times b.q + a.p \times b.q)$

$ab.(p \times q + a.p \times q + p \times b.q)$

$\quad \sim ab(1):(p \times q + a.p \times q + p \times b.q + ab.(p \times q + a.p \times q + p \times b.q))$

Hence $a.p \times b.q \sim ab.(p \times q + a.p \times q + p \times b.q)$

The step leading to (*) uses a factorisation theorem ([FT2]); the four hypotheses of the theorem can be verified as follows (using the fact that $a.p \sim a(t):(p+a.p)$ and $b.q \sim b(t):(q+b.q)$):

1) $(p+a.p) \times (q+b.q+b(t):(q+b.q)) \sim (p+a.p) \times (q+b.q)$

2) $(p+a.p+a(t):(p+a.p)) \times (q+b.q) \sim (p+a.p) \times (q+b.q)$

3) $(p+a.p+a(t):(p+a.p)) \times (q+b.q+b(t):(q+b.q)) \sim (p+a.p) \times (q+b.q)$

4) $(p+a.p)\times(q+b.q) + a(t):(p+a.p)\times b(t):(q+b.q) \sim (p+a.p) \times (q+b.q)$

A closely related operator to a.p is **indefinite delay:**

$$\delta_a p \Leftarrow p + a.p$$

where the agent p may be activated immediately, or delayed indefinitely by an action a. The following laws can all be easily proved from the properties of a.p:

$$\delta_a 0 \sim \mathbb{K}_a$$
$$\delta_a \mathbb{K}_a \sim \mathbb{K}_a$$
$$\delta_a(\delta_a p) \sim \delta_a p$$
$$\delta_a p \times \delta_b q \sim \delta_{ab}(\delta_a p \times \delta_b q)$$
$$\delta_a p \times \delta_b q \sim \delta_{ab}(\delta_a p \times q + p \times \delta_b q)$$

5.7 Synchronising on Non-Synchronous Input

Suppose we want to express an agent I which takes an input on port a and produces the same value as output on port $\bar{b}$ without any delay. The simplest form of I, written I_{null} with null={nil}, accepts a single value nil and can be written:

$$I_{null} \Leftarrow a_{nil}\bar{b}_{nil}[1]:I_{null}$$

i.e. $I_{null} \sim \mathbb{K}_{a_{nil}\bar{b}_{nil}}$.

The next simplest form of I is I_{bool} with bool={l,h} (low and high) and, surprisingly, this cannot be written with deterministic guards. In fact the definition:

$$I_{bool} \Leftarrow a_l\bar{b}_l[1]:I_{bool} + a_h\bar{b}_h[1]:I_{bool}$$

will not work because the input I_{bool} may change at any time, while for example the guard $a_l\bar{b}_l[1]$: once selected must be taken to completion. The "1" in the definition of I_{bool} is critical, and cannot be replace by a different number without changing the behaviour of the agent. Computing the behaviour in case of unsynchronised input we obtain, for example:

$$(I_{bool} \times \bar{a}_h[0.5]:\mathbb{K}_{\bar{a}_l})\backslash a_l\backslash a_h \sim \bar{b}_h[0.5]:0$$

while we might expect the result to be $\bar{b}_h[0.5]:\mathbb{K}_{\bar{b}_l}$. The example above behaves correctly if we replace "1" by "0.5" in the definition of I_{bool}, but of course we can always pick up an input waveform so that the output degenerates to 0.

Let us now redefine I_{bool} by using nondeterministic guards:

$$I_{bool} \Leftarrow a_l\bar{b}_l(1):I_{bool} + a_h\bar{b}_h(1):I_{bool}$$

we can now prove that the choice of "1" in the definition is not critical:

$$I_{bool} \sim a_l\bar{b}_l(t):I_{bool} + a_h\bar{b}_h(1):I_{bool}$$
$$I_{bool} \sim a_l\bar{b}_l(1):I_{bool} + a_h\bar{b}_h(t):I_{bool}$$

Then we can prove the desired properties of I_{bool}, using the factorisation theorems [FT3] and [FT4] to relate the asynchronous behaviour of I_{bool} to a synchronous input:

$$(I_{bool} \times \bar{a}_h[t]:p)\backslash a_h \sim \bar{b}_h[t]:(I_{bool} \times p)\backslash a_h$$

$$(I_{bool} \times \bar{a}_l[t]:p)\backslash a_l \sim \bar{b}_l[t]:(I_{bool} \times p)\backslash a_l$$

The other factorisation theorems ([FT1] and [FT2]) are needed to prove the interactions of two asynchronous agents; for example in the proof of:

$$(I_{bool}\{c_i/b_i\} \times I_{bool}\{\bar{c}_i/a_i\})\backslash c_i \sim I_{bool} \quad (i \in \{l,h\})$$

5.8 An Asynchronous Rising Edge Counter

We now discuss an example of the application of nondeterministic guards. Suppose we have a boolean signal:

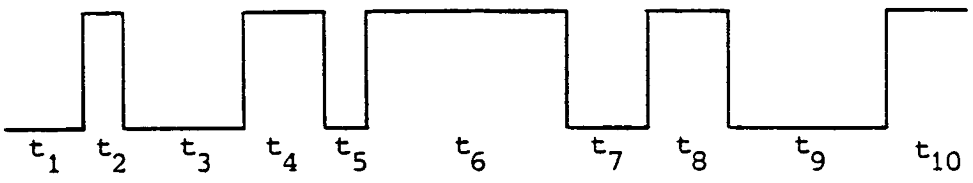


Figure 5.2

where the length of the segments t_i is completely arbitrary. The problem consists in counting the number of rising edges (i.e. transitions from low to high) which have occurred in the signal at any given time. It is pretty well evident that there can be no solution using deterministic guards as any proposal would be bound to fail for some input waveform.

The counter has two states: Low_n and $High_n$, and n is increased at any passage from Low to High (for simplicity, the count n is not supplied as an explicit output)

$$Low_n \Leftarrow l(1):Low_n + h(1):High_{n+1}$$

$$High_n \Leftarrow l(1):Low_n + h(1):High_n$$

Note how the guards "l" and "h" are programmed to last as long as their corresponding asynchronous inputs. As usual, we first have to prove some invariance lemmas:

$$Low_n \sim l(t):Low_n + h(1):High_{n+1}$$

$High_n \sim l(1):Low_n + h(t):High_n$

$Low_n \sim l(1):Low_n + h(t):High_{n+1}$

$High_n \sim l(t):Low_n + h(1):High_n$

The following equivalences state the correctness of the counter; the input signal is assumed to be a sequence of deterministic guards, and the equivalences can be proved by using [FT3] and [FT4].

$(Low_n \times \bar{l}[t]:p)\backslash l \sim 1[t]:(Low_n \times p)\backslash l$

$(High_n \times \bar{h}[t]:p)\backslash h \sim 1[t]:(High_n \times p)\backslash h$

$(Low_n \times \bar{h}[t]:p)\backslash h \sim 1[t]:(High_{n+1} \times p)\backslash h$

$(High_n \times \bar{l}[t]:p)\backslash l \sim 1[t]:(Low_n \times p)\backslash l$

5.9 Descriptive Operators

Some operators can be introduced in order to describe properties of agents, without adding any expressive power themselves.

Here is a very simple descriptive operator:

$$[\Pi \rightarrow] \quad \frac{p \xrightarrow[t]{a} q}{\Pi p \xrightarrow[t]{1} \Pi q}$$

Definition 5.20 An agent p is persistent if $\Pi p \sim \mathbf{1}$ □

The persistency operator allows us to distinguish agents which may end up in disaster from agents which carry on forever. This operator can help us if we want to exclude nonpersistent agents of any kind from the class of "physically existent" or "implementable" agents.

In order to talk about synchrony, we can introduce a synchronisation operator Γ, designed to "impose" a clock on an otherwise unsynchronised agent. We actually introduce an indexed family Γ_t of such operators, meaning that $\Gamma_t p$ synchronises p to a clock of period $t \in \mathbb{K}$.

$$[\Gamma_t\rightarrow] \qquad \frac{p \xrightarrow[t]{a} q}{\Gamma_t p \xrightarrow[t]{a} \Gamma_t q}$$

$$[\Gamma_{t+u}\rightarrow] \qquad \frac{\Gamma_t p \xrightarrow[u+v]{a} q}{\Gamma_t p \xrightarrow[u]{a} a[v]:q}$$

Rule $[\Gamma_t\rightarrow]$ says that $\Gamma_t p$ can perform "t-ticks" only if p can, i.e. p must be synchronisable to a clock of period t, otherwise $\Gamma_t p$ will stop.

Rule $[\Gamma_{t+u}\rightarrow]$ is introduced in order to preserve the density lemma.

Definition 5.21 An agent p is **t-synchronous** if $p \sim \Gamma_t p$ □

The definition of t-synchrony intends to capture the idea that all the "significant changes" (i.e. transitions from an a-action to a different b-action) in a t-synchronous agent occur at instants which are divisors of t. For example:

$p \Leftarrow a[2]:b[2]:p$

p is 2-synchronous, 1-synchronous, etc., but it is not 3-synchronous, 4-synchronous, etc. because p cannot produce any action longer than 2. Note that **1** is t-synchronous for all t.

Definition 5.22 An agent p is **non-synchronous** if it is not t-synchronous for any t □

An example of non-synchronous process is provided by a "bouncing ball" agent which is persistent and changes its output at a faster and faster rate:

$p_n \Leftarrow a[1/n]:b[1/n]:p_{n+1}$

If we eliminate the nondeterministic guard "a(t):" from our signature, and we replace "a[t]:" by "a[1]:" (abbreviated "a:"), than all the agents which can be expressed are 1-synchronous. The set of 1-synchronous agents corresponds exactly to the synchronous-CCS calculus [Milner 81], in the sense that the same set of laws holds.

Finally we can try to characterise some form of asynchronous behaviour by the following operator:

$$[\Delta\rightarrow] \quad \frac{p \xrightarrow[t]{a} q}{\Delta p \xrightarrow[t+u]{a} \Delta q}$$

which stretches by arbitrary amounts all the actions of an agent.

Definition 5.23 **An agent p is asynchronous iff** $p \sim \Delta p$ □

Note that this definition allows us to make a subtle distinction between non-synchronous or non t-synchronous agents (which are deterministic) and asynchronous ones (which are completely nondeterministic) and that many other behaviours lay in between.

6. Conclusions

6.1 Achievements and Future Work

This thesis has demonstrated how algebraic techniques can be naturally applied to several aspects of hardware description and verification, with particular emphasis on the syntax and semantics of VLSI circuits and design systems. Indeed, we might say that our effort was not to apply preconceived techniques to new problems, but rather that the problems themselves seemed to fit naturally in a environment which had developed for different (but after all, related) purposes.

In Chapter 1 we have introduced a notation for the structural description of networks, giving laws for net expressions which characterise a suitable kind of graphs. This work might be extended in several directions. Infinitary sorts might be useful in some applications; for example the sorts used in Chapter 3 are finite but, as explained in Section 1.10, they might be naturally regarded as uncountably infinite. An attempt could be made to axiomatise planar networks, and to prove completeness and initiality theorems with respect to that axiomatisation; we have taken the simpler approach of defining planar networks as a particular case of networks, without trying to characterise them (Section 1.7). Planar sorts and cycles might be extended to three-dimensional objects in order to express the incompenetrability of solids; this is briefly discussed in Section 1.10. Finally, the problem of deciding the equivalence of two net expressions (or equivalently the isomorphism of two net graphs) appears to be polynomial, but we need to study tight upper bounds and to provide good equivalence algorithms.

In Chapter 2 we have shown how a wide variety of levels of description of hardware circuits can be cast in similar formal frameworks, so that the passage between levels is facilitated. A formal semantics has been given for the topmost behavioural level which concerns synchronous systems; formal proofs concerning these systems seem to be well suited to mechanisation, but they also badly require mechanical aids. A more complex problem is the definition of viable semantics for non-synchronous systems; Chapters 4 and 5 attack this problem, but further work is needed.

A major problem left unsolved in this thesis is the definition of a satisfactory dynamic semantics of low-level hardware (i.e. below the gate level). Rather accurate informal models are discussed in Section 2.4 but difficulties arise in formalisation; certain semantic techniques (like those discussed in chapter 4) could be applied in principle, but they seem to give rise to intractable formal systems. The static CSA semantics we present seems instead rather satisfactory because it can model the context-dependent relaxation processes which are characteristic of low-level hardware, and can help in understanding the dynamic behaviour of circuits.

In the study of the translations between levels, two novel algorithms have been presented. One is an efficient stretching algorithm for grid structures, which simulates the two-dimensional stretching of matrices by the composition of stretching transformations. The second is an algorithm for the context-driven translation of purely topological planar stick diagrams (represented by textual expressions) into grid structures, and hence into layouts. Both algorithms need to be tested, expecially because the latter algorithm uses limited heuristics.

In Chapter 3 we have described an experimental system for the design of VLSI layouts, which uses algebraic concepts to abstract away from geometric details. The system is built around a functional higher-order language, which provides the necessary control and parameterisation structures. Interactive graphical feedback is used in the development of programs in order to better relate the textual representations to geometrical layouts. The system might be improved in several directions; in particular, planarity checks and design rule checks were not included in the implementation to allow for a deeper investigation of other innovative features.

Finally we have presented two different attempts towards the formalisation of real-time systems using, respectively, denotational and operational semantics techniques. The theories developed seem to give rise to satisfactory semantic models, but much theoretical and practical work has to be carried out in order to test these ideas on large scale applications. It is hoped that formal systems of this kind can be used to formulate and prove properties of low-level hardware; encouraging steps in this direction are described in [Gordon 81a, Gordon 81b]. Several intuitive properties of the analog processes formal system have been left as conjectures which we believe could be formally stated and proved in our framework.

6.2 The Future

6.2.1 VLSI

VLSI is going to become the single most important technology of the next 20 years, and probably longer. It is already the most sophisticated technology ever devised, and its potentialities are today too remote to be fully appreciated. Even its limitations are too remote, and it seems that for some time the main difficulties will consist in effectively exploiting the remarkable features which are presently available.

The shape of things to come in VLSI is usually expressed by Moore's Laws (so-called). The First Law is very optimistic (slightly more optimistic than reality) stating that the number of devices per silicon chip doubles every 2-3 years. This "law of nature" was discovered in the mid-sixties and the exponential rate of growth it forecasts has been essentially respected up to now, and it will also be roughly respected in the next 5-10 years. After that, some very basic physical limits of the present integration technology will be reached, although progress may continue in other directions.

Hence in about 10 years we shall be able to put something like 100 million transistors on a silicon wafer. We have to think about how to use them in interesting ways. Exciting possibilities have already been found which critically use the features of VLSI technology [Kung 80], and many more remain to be discovered.

Almost every aspect of computer science will have to cope with this new technology. Even the less technology-related disciplines, like complexity theory, semantics, formal languages, algorithms and software engineering are going to be deeply influenced by this new way of looking at computation. This is just the beginning, and we should carefully try to avoid repeating old mistakes.

6.2.2 Design Tools

Moore's Second Law is, instead, very pessimistic. It says that the design time of VLSI chips grows exponentially with the number of devices per chip (and that we are already close to the almost-vertical zone).

The biggest task for the design tool designers in the next few years will be to falsify this law. This cannot be done by linearly improving existing design systems and methodologies; totally new lines of attack are required. It is far too early now to guess what kind of design systems will prevail. It seems certain that translation techniques and effective graphic interaction will be useful, but it is not at all clear how.

One noticeable trend is towards very complex systems with data-bases maintaining multiple levels of descriptions of circuits, where the user can jump from level to level editing text and graphics and optimising subcircuits, and where the system preserves the overall integrity of the design by making expert autonomous decisions based on complex heuristics.

This is not our aim; we have tried to demonstrate that the problems involved can best be cast in a simple framework involving a few primitive concepts, and many interesting translations are almost completely algorithmic, using only limited heuristics or a few user interactions at critical points. As many examples in computer science have shown, sometimes only simple solutions are able to solve complex problems.

6.2.3 Semantics

Given the complexity of future hardware systems, and their widespread use in all aspects of human life, important security problems arise. How can we know that chips controlling critical systems, like power plants, airplanes, cars etc. will not contain fatal "bugs", or that they will be immune to catastrophic hardware failures? In the case of microcoded systems, or hardware-software combinations, we cope with the still noticeably unsolved problem of medium-large scale software verification. One (not yet well founded) hope here is that abundance of hardware will let us write software suitable for formal verification.

Further work is needed in the semantics and verification of hardware systems; at the most abstract level this reduces to the problem of giving tractable semantics to extensively concurrent systems. For the purposes of verification, one should be aware of formal systems which are completely satisfactory from the point of view of expressiveness and generality, but which do not allow us to carry out complex proofs because of technical clumsiness. In this respect equational approaches like [Milner 80] are promising, because they seem to be very suitable for mechanisation.

At lower levels (like the CSA level and below) not even semantics is well established. This is to be attributed to the fact that in electronic circuits the semantics of the whole is not a simple function of the semantics of the parts, and complex relaxation processes are involved. The main semantic techniques seem to come from the field of circuit simulation, and simulation is not satisfactory from a semantic point of view. Even simulators are often criticised on the ground of not being realistic, but this unrealism may be because they have to compromise between accuracy and efficiency; disregarding efficiency there may be a satisfactory semantic model.

In conclusion the current lack of flexible verification systems may be because verification is at the same time a very difficult problem in each of several distinct areas: mathematical foundations, artificial intelligence, semantics and software engineering. There is some indication that these areas are slowly converging towards viable solutions, and together with the steady increase in computational power we retain some hope for future success.

Appendix I. Syntax Description Notation

The following conventions are used to present grammars:

- strings between single quotes "'" are terminal symbols;
- "''" is the null string;
- identifiers are non-terminals;
- juxtaposition is concatenation;
- "|" is disjunction;
- "[...]" means zero or one times "...";
- "{ ... }n" means n or more times "..." (default n=0);
- "{ ... / --- }n" means n or more times "..." separated by "---" (default n=0);
- parenthesis "(...)" are used for precedence;
- "::=" is used for mutually recursive definitions.

As an example, the metanotation is described in terms of itself:

```
Grammar ::= {Identifier '::=' Term}


Term ::=
    '''' [ Characters ] '''' |
    Identifier |
    Term Term |
    Term '|' Term |
    '[' Term ']' |
    '{' Term [ '/' Term ] '}' [Integer] |
    '(' Term ')'
```

Appendix II. Table of Symbols

$\emptyset$	empty set
$\{ .. \}$	sets
$\{ .. \mid .. \}$	set descriptions
$a \in A$	set membership
$A \cup B$	set union
$A \cap B$	set intersection
$A \setminus B$	set difference
$A \oplus B$	symmetric difference ($A \setminus B \cup B \setminus A$)
$A \subseteq B$	A is a subset of B
$A \supseteq B$	A is a superset of B
$\lvert S \rvert$	cardinality of a set S
$A \longrightarrow B$	function space
B^A	function space $A \longrightarrow B$
$f: A \longrightarrow B$	f is a function from A to B
$f: a \longmapsto b$	f maps a into b
$\ulcorner f \urcorner$	domain of a function
$f \downarrow A$	function restricted to the domain A
id_A	identity function on A
$f \circ g$	function composition ($(f \circ g)(a) = f(g(a))$)
$f \# g$	function pairing ($(f \# g)\langle a,b \rangle = \langle f(a), g(b) \rangle$)
f^{-1}	inverse function
$f(a)$	function application
$A \times B$	cartesian product
$\langle a,b \rangle$	pair
$\downarrow_1$	left projection ($\langle a,b \rangle \downarrow_1 = a$)
$\downarrow_2$	right projection ($\langle a,b \rangle \downarrow_2 = b$)
A^*	set of finite lists over A
$[e_1; .. ; e_n]$	list ($n \geq 0$)

$\sim$	boolean not
$\wedge$	boolean and
$\vee$	boolean or
$\Rightarrow$	implies
$\Leftarrow$	implied
$\Leftrightarrow$	if and only if
$\forall$	forall
$\exists$	exists
a$\Rightarrow$b;c	if a then b else c
let $a_1=N_1$ and .. and $a_n=N_n$ in M	binds N_i to a_i in the scope M
M where $a_1=N_1$ and .. and $a_n=N_n$	binds N_i to a_i in the scope M
$\equiv$	equivalence
$=$	equality
$\simeq$	isomorphism
$\triangleq$	equal by definition
s	sort
S	set of sorts
$w \in S^*$	list of sorts
$\underline{\Sigma} = \langle S,\Sigma\rangle$	signature
$\alpha_{w,s} \in \Sigma_{w,s}$	operator symbol of rank w,s, arity w, sort s
$\underline{A} = \langle A^{\bullet},A\uparrow\rangle$	algebra with carriers $A^{\bullet}$ and operations $A\uparrow$
$\phi^{A}_{\alpha_{w,s}}$	operator of $\underline{A}$ named by $\alpha_{w,s}$
Types	net algebra types
PortNames	net algebra port names
a,b	port names
A,B,C	finite sets of port names
s: A -> Types	net algebra sort

e	net expression
$e : \{a_i : T_i\}$	syntax for sorts
$l \varepsilon \mathbb{L}$	literal
$\lambda(l)$	sort of a literal
$e\backslash a$	restriction
$e\{r\}$	renaming $e\{a_i\backslash b_i\}$ (a_i becomes b_i)
$\vert$	implicit composition
$e[r]e'$	explicit composition $e[a_i\text{--}b_i]e'$
$\sigma(e)$	sort of an expressions
$\equiv$	convertibility
$t \rightsquigarrow b$	clocked transitions
ϕ_1,ϕ_2	clock phases
$\langle 1:t_i \rightsquigarrow b_i\rangle$	phase-1 clusters
$\langle 2:t_i \rightsquigarrow b_i v_i\rangle$	phase-2 clusters
tt,ff	boolean true and false
0	CSA strong zero
1	CSA strong one
$\tilde{0}$	CSA weak zero
$\tilde{1}$	CSA weak one
U	CSA strong undefined
$\tilde{U}$	CSA weak undefined
Z	CSA floating
F	either 0 or $\tilde{0}$
T	either 1 or $\tilde{1}$
$\diamond$	CSA connection operation
GND	ground
VDD	power supply
$\lambda x.M$	lambda notation for functions

$M \rightsquigarrow \alpha$	analog process transition
χ	product of transitions
$\uplus$	join of signals
NIL	empty transition
$\perp$	nosignal
$\Pi_{i \varepsilon I} T_i$	indexed product of transitions
$\mid$	composition of analog processes
$\backslash \alpha$	restriction
$\partial t.V$	pointwise definition of signals
$\mu\alpha.M$	recursively defined signal
$M[N/\alpha]$	syntactic substitution (N replaces α)
$\{\alpha_i/\beta_i\}$	renaming (β_i replaces α_i)
L	finite set of labels
$P_{L,L'} = L' \longrightarrow T_L$	a domain of processes
$P = \Sigma_{L,L'} P_{L,L'}$	domain of all processes
$T_L = S^L \longrightarrow S$	a domain of transitions
$S = K \longrightarrow V$	a domain of signals
K	time (positive reals)
V	a domain of signal values
$\langle V, \phi, \vee \rangle$	signal monoid
$[\alpha_i : s_i]$	labelled tuples of signals
$[\alpha_i : s_i].\alpha_j = s_j$	field extraction
$\lambda[a_i].M[a_i]$	abbreviates $\lambda x.M[x.\alpha_i/a_i]$
$\{t_i \rightsquigarrow \alpha_i\}$	syntax for processes
Y	least fixpoint operator
$\mathbb{T}$	semantics of terms
$\mathbb{S}$	semantics of signals
$\mathbb{P}$	semantics of processes
$\uparrow V$	$\partial t.V$ when t does not occur in V
$S \Delta S'$	delay operator on signals
$\perp$	semantic bottom

References

[Barton 81] E.E.Barton: A Non-Metric Design Methodology for VLSI. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Batali 81] J.Batali, N.Mayle, H.Shrobe, G.Sussman, D.Weise: The DPL/Daedalus Design Environment. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Bryant 81] R.E.Bryant: A Switch Level Model of MOS Logic Circuits. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Buchanan 80] I.Buchanan: Modelling and Verification in Structured Integrated Circuit Design. Ph.D. Thesis, Dept. of Computer Science, University of Edinburgh, 1980.

[Burge 75] W.H.Burge: Recursive Programming Techniques. Addison Wesley, 1975.

[Burstall 77] R.M.Burstall, J.A.Goguen: Putting Theories Together to Make Specifications. Proc. of the 5th International Joint Conference on Artificial Intelligence, Cambridge, Mass. 1977.

[Cardelli 80] L.Cardelli: Analog Processes. Proc. 9th Symposium on Mathematical Foundations of Computer Science. Lecture Notes in Computer Science n.88, Springer-Verlag 1980.

[Cardelli 81a] L.Cardelli, G.Plotkin: An Algebraic Approach to VLSI Design. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Cardelli 81b] L.Cardelli: Sticks & Stones: An Applicative VLSI Design Language. Internal Report CSR-85-81, Dept of Computer Science, University of Edinburgh.

[Cardelli 82] L.Cardelli: Real Time Agents. Proc of the 9th International Colloquium on Automata, Languages and Programming, Aarhus 1982 (to appear).

[Chaney 73] T.J.Chaney, C.H.Molnar: Anomalous Behaviour of Synchronizers and Arbiter Circuits. IEEE Transaction on Computers, April 1973.

[Chazelle 81] B.Chazelle, L.Monier: Optimality in VLSI. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Clark 80] W.A.Clark: From Electron Mobility to Logical Structure: A View of Integrated Circuits. Computing Surveys 12(3) 1980.

[Conway 80] L.A.Conway, A.Bell, M.E.Newell: MPC 79: The Large-Scale Demonstration of a New Way to Create Systems in Silicon. Lambda 1, 2, 1980.

[Fairbairn 78] D.G.Fairbairn, J.A.Rowson: ICARUS: An Interactive Integrated Circuit Layout Program. Proc. 15th Design Automation Conference, 1978.

[Floyd 80] R.W.Floyd, J.D.Ullman: The Compilation of Regular Expressions into Integrated Circuits. In: Proc. 21st Annual Symposium on Foundations of Computer Science. IEEE Computer Society.

[Foster 80] M.J.Foster, H.T.Kung: The Design of Special-Purpose VLSI Chips: Example and Opinions. Computer 13(1).

[Foster 81] M.J.Foster, H.T.Kung: Recognize Regular Languages with Programmable Building-Blocks. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Goguen 78] J.A.Goguen, J.W.Thatcher, E.G.Wagner: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In: R.T.Yeh(ed.): Current Trends in Programming Methodology. Vol IV, Prentice-Hall, 1978.

[Gordon 79a] M.J.Gordon, R.Milner, C.P.Wadsworth: Edinburgh LCF. Lecture Notes in Computer Science, n.78. Springer-Verlag 1979.

[Gordon 79b] M.J.Gordon: Denotational Description of Programming Languages. Springer-Verlag, 1979.

[Gordon 81a] M.J.Gordon: A Very Simple Model of Sequential Behaviour of nMOS. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Gordon 81b] M.J.Gordon: Register Transfer Systems and Their Behaviour. Proc. of the 5th International Conference on Hardware Description Languages, 1981.

[Gratzer 79] G.Gratzer: Universal Algebra. Springer-Verlag, 1979.

[Hayes 81] J.P.Hayes: A Logic Design Theory for VLSI. Proc. 2nd Caltech Conference on VLSI. Caltech, January 1981.

[Hennessy 80] M.Hennessy, R.Milner: On Observing Nondeterminism and Concurrency. Proc. ICALP 80. Lecture notes in Computer Science n.85. Springer-Verlag 1980.

[Johannsen 79] D.Johannsen: Bristle Blocks: A Silicon Compiler. Proc. 16th Design Automation Conference. June 1979.

[Kung 80] H.T.Kung, C.E.Leiserson: Algorithms for VLSI Processor Arrays. In: C.A.Mead, L.A.Conway: Introduction to VLSI Systems. Addison Wesley, 1980.

[Lattin 81] W.W.Lattin, J.A.Bayliss, D.L.Budde, J.R.Rattner, W.S.Richardson: A Methodology for VLSI Chip Design. Lambda 2(2), 1981.

[Locanthi 78] B.Locanthi: LAP: A SIMULA Package for IC Layout. Caltech Display File #1862, 1978.

[MacQueen 79] D.MacQueen: Models for Distributed Computing. Report n.351, IRIA-Laboria, 1979.

[Masumoto 80] R.T.Masumoto: The design of a 16x16 Multiplier. Lambda 1(1), 1980.

[Mead 80] C.A.Mead, L.A.Conway: Introduction to VLSI Systems. Addison Wesley, 1980.

[Melen 80] R.Melen, H.Garland: Understanding CMOS Integrated Circuits. H.W.Sams&Co. 1980.

[Mikkelson 81] J.M.Mikkelson, L.A.Hall, A.K.Malhotra, S.D.Seccombe, M.S.Wilson: An NMOS VLSI Process for Fabrication of a 32b CPU Chip. IEEE International Solid State Circuits Conference, 1981.

[Milner 78] R. Milner: Synthesis of Communicating Behaviour. Proc. 7th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, n.64. Springer-Verlag 1978.

[Milner 79] R.Milner: Flowgraphs And Flow Algebras. Journal of the ACM, 26(4), 1979.

[Milner 80] R. Milner: A Calculus of Communicating Systems. Lecture Notes in Computer Science, n.92. Springer-Verlag 1980.

[Milner 81] R.Milner: On Relating Synchrony and Asynchrony. Internal Report CSR-75-80. Dept. of Computer Science, University of Edinburgh.

[Mosteller 81] R.C.Mosteller: REST - A Leaf Cell Design System. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Mudge 81] J.C.Mudge: VLSI Chips at the Crossroads. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Newman 79] W.M.Newman, R.F.Sproull: Principles of Interactive Computer Graphics. McGraw-Hill 1979.

[Park 81] D.M.R.Park: Concurrency and Automata on Infinite Sequences. Proc. GI Conference, 1981.

[Patil 74] S.S.Patil: Bounded and Unbounded Delay Synchronizers and Arbiters. Computation Structures Group Memo 103, M.I.T. June 1974.

[Piloty 81] R.Piloty: A Language System for Hardware Description. Proc. of the 5th International Conference on Hardware Description Languages, 1981.

[Plotkin 81] G.D.Plotkin: A Structural Approach to Operational Semantics. Report DAIMI FN-19, Dept. of Computer Science, University of Aarhus, 1981.

[Preparata 79] F.P.Preparata, J.Vuillemin: The Cube-Connected-Cycles: A Versatile Network for Parallel Computation. Proc. 20th Annual Symposium on Foundations of Computer Science, October 1979.

[Rem 81] M.Rem, C.Mead: A Notation for Designing Restoring Logic Circuitry in CMOS. Proc. 2nd Caltech Conference on VLSI. Caltech, January 1981.

[Rivest 80] R.R.Rivest: A description of a Single Chip Implementation of the RSA Cipher. Lambda 1(3) 1980.

[Rowson 80] J.A.Rowson: Understanding Hierarchical Design. Ph.D. Thesis, California Instituite of Technology, 1980.

[Rupp 81] C.R.Rupp: Components of a Silicon Compiler System. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[S.A. 77] Scientific American, September 1977. (Special issue on microelectronics)

[Sannella 81] D.Sannella: A New Semantics for Clear. Internal Report CSR-79-81, Dept. of Computer Science, University of Edinburgh, 1981.

[Scott 76] D.S.Scott: Data Types as Lattices. SIAM Journal of Computing, 4, 1976.

[Seitz 80] C.L.Seitz: System Timing. In: C.A.Mead, L.A.Conway: Introduction to VLSI Systems. Addison Wesley, 1980.

[Sequin 81] C.H.Sequin: Generalized IC Layout Rules and Layout Representations. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Smith 80] L.D.Smith: The Elementary Structural Description Language. CSR-53-80, Dept. of Computer Science, University of Edinburgh, 1980.

[Sproull 80] R.F.Sproull, R.F.Lyon: The Caltech Intermediate Form for LSI Layout Description. In: C.A.Mead, L.A.Conway: Introduction to VLSI Systems. Addison Wesley, 1980.

[Steele 80] G.L.Steele Jr., G.J.Sussman: Desing of a LISP-Based Microprocessor. Communications of the ACM 23, 11, 1980.

[Stoy 77] J.E.Stoy: Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory. M.I.T. Press, 1977.

[Tarolli 80] G.Tarolli: Chip Assembler. Proc. Computer Architecture Conference, 1980.

[Trimberger 79] S.Trimberger: A CAD System Combining Interactive Graphics and a Layout Language. Caltech SSP File #2499, 1979.

[Weste 81] N.Weste, B.Ackland: A Pragmatic Approach to Topological Symbolic IC Design. In: J.P.Gray(ed.): VLSI 81. Academic Press, 1981.

[Whitney 81] T.E.Whitney: A Hierarchical Design-Rule Checking Algorithm. Lambda 2, 1, 1981.

[Williams 78] J.D.Williams: Sticks - A Graphical Compiler for High Level LSI Design. Proc. NCC, May 1978