Concurrent Algebras for VLSI Design

T. S. Balraj

CUCS-413-88

Concurrent Algebras for VLSI Design

T. S. Balraj

5 August 1988
Revised 31 October 1988

1 Introduction

As the size and complexity of VLSI chips increases, designers are beginning to rely more and more on automated chip design systems to help layout, route, or even design circuits. Silicon compilers convert the functional description of a system to a mask level design of a chip that implements the system. In order to ease the task of describing the system, and to help analyse and verify its working, the description languages are based on algebraic systems. A typical circuit has a number of actions occurring at any given time. So we use concurrent algebras as the basis for the description languages.

In this paper, we survey algebras that enable the description and analysis of concurrent systems. We examine them particularily from the point of view of using them to implement systems in VLSI. We therefore concentrate on the basics of each algebra, and omit features that are not readily implementable, such as recursion.

We will look at four algebras : trace theory, path expressions, Milner's calculus of communicating systems (CCS), and an algebra of finite events (CAFE). We choose the first three since each has been used in some form of silicon compiler or other automated hardware design system, and together they demonstrate all the features found in higher level description systems for hardware. The fourth is an algebra that we are developing to address the problems of describing systems of events of finite duration.

In chapter 2 we introduce an informal net notation and the concept of observers, which we use in the next four chapters to to describe each algebra briefly. In chapter 7, we compare the algebras in terms of their treatment of independence, the type of parallel composition they use, and the inter-event dependencies they allow. We end by explaining the relative advantages and disadvantages of the algebras in various situations.

The goal hoped that this comparative discussion of the algebras is to aid in the design of process description languages to be used in silicon compilers.

Research supported in part by DARPA contract #F-29601-87-C-0074

Concurrent Algebras for VLSI Design

T. S. Balraj

5 August 1988
Revised 31 October 1988

1 Introduction

As the size and complexity of VLSI chips increases, designers are beginning to rely more and more on automated chip design systems to help layout, route, or even design circuits. Silicon compilers convert the functional description of a system to a mask level design of a chip that implements the system. In order to ease the task of describing the system, and to help analyse and verify its working, the description languages are based on algebraic systems. A typical circuit has a number of actions occurring at any given time. So we use concurrent algebras as the basis for the description languages.

In this paper, we survey algebras that enable the description and analysis of concurrent systems. We examine them particularily from the point of view of using them to implement systems in VLSI. We therefore concentrate on the basics of each algebra, and omit features that are not readily implementable, such as recursion.

We will look at four algebras : trace theory, path expressions, Milner's calculus of communicating systems (CCS), and an algebra of finite events (CAFE). We choose the first three since each has been used in some form of silicon compiler or other automated hardware design system, and together they demonstrate all the features found in higher level description systems for hardware. The fourth is an algebra that we are developing to address the problems of describing systems of events of finite duration.

In chapter 2 we introduce an informal net notation and the concept of observers, which we use in the next four chapters to to describe each algebra briefly. In chapter 7, we compare the algebras in terms of their treatment of independence, the type of parallel composition they use, and the inter-event dependencies they allow. We end by explaining the relative advantages and disadvantages of the algebras in various situations.

The goal hoped that this comparative discussion of the algebras is to aid in the design of process description languages to be used in silicon compilers.

2 Basic Concepts

2.1 Terminology

The stated purpose of the algebras we shall discuss is to describe the behaviours of "concurrent processes". In the literature, the term *concurrent* is used loosely to express two distinct concepts. The primary meaning of concurrence is *causal independence.* But the term is also used to express *simultaneity*, which is the most interesting result of independence. However, forced simultaneity is actually a form of temporal dependence. So we must keep the distinction between the two meanings of concurrency in mind while we examine the literature. In this paper, we will strictly limit the use of the term to mean independence. We shall also use the phrase *occurs in parallel* to imply concurrency rather than simultaneity.

2.2 An Informal Net Notation

Throughout this paper, we represent processes by drawing them as nets. In this section, we describe the notation we use. We use this diagrammatic representation of processes rather than an expressional form to avoid confusion with the representation of processes in the algebras we discuss.

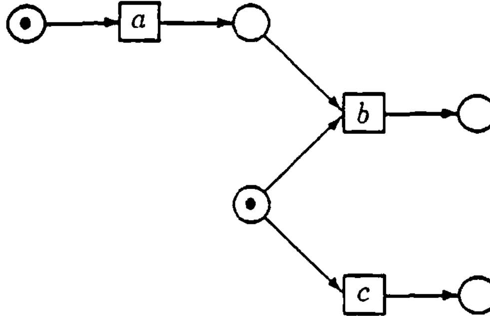
We use a restricted form of Petri nets. A Petri net is a 4-tuple (C, E, F, M_0) where C and E are disjoint sets of *conditions* and *events* respectively, F is a binary relation $\subseteq (C \times E) \cup (E \times C)$ called the *flow relation*, and M_0 is a non-null subset of C called the *initial marking*. We draw conditions as circles, events as boxes, and the flow relation as arrows from circles to boxes and vice versa. We represent the marking of the net by drawing *tokens* in the appropriate conditions. We represent a process by drawing the corresponding net with its initial marking.

If $(c, e) \in F$ for some condition c and event e, then c is called a *pre-condition* of e; if $(e, c) \in F$ c is a *post-condition* of e. We represent the sets of pre- and post-conditions of an event e by $\mathsf{pre}(e)$ and $\mathbf{post}(e)$ respectively.

An event occurs, or *fires*, when all its pre-conditions are marked (have tokens), and all its post-conditions are unmarked. After an event fires, its pre-conditions are unmarked, but its post-conditions are marked. If two events have one or more pre-conditions in common, they are said to be in *conflict*, and only one of them can fire at a time.

Two undesirable situations can arise with Petri nets. The first occurs when all the pre-conditions of an event are marked, but some its post-conditions are also marked. This is called a *contact situation*. In this situation, the definition of Petri nets prohibits the event from occurring. If this prohibition were absent, so that in the net ⊙→[a]→⊙, we allow the event a to fire giving ○→[a]→⊙, then in the situation ⊙→[a]→⊙→[b]→○, if each event occurs once, either ○→[a]→⊙→[b]→⊙ or ○→[a]→○→[b]→⊙ results, depending on which event occurs first. If the events occur simultaneously, it is unclear what the resultant marking should be. We would like therefore to limit ourselves to *contact-free* nets, in which contact situations cannot arise.

The other situation which we would like to avoid is called *confusion.* In the net



it is not obvious whether b and c are in conflict: if c fires before a, there is no conflict, while if a fires first, b and c are in conflict. In a hardware implementation of the net, if a and c are simultaneous, an anomalous firing of b may occur.

We therefore define a restricted form of nets, called *simply matched nets*, that is both contact-free and confusion-free. This allows us to clearly represent and readily identify branching and concurrency.

We formalise the concept of sequentiality by defining a precedence relation. Intuitively, a condition c precedes an event e, written $c < e$, if there exists a sequence of events in which c holds before e fires, and if e never fires while c holds. Similarily, $e < c$ if there exists a sequence of events in which c holds after e fires, and e never fires while c holds.

Definition 2.1 *We define a binary relation* $\subseteq (C \cup E) \times (E \cup C)$, precedes, *as follows: For conditions c and c', and events e and e'.*

1. $e < c \Leftrightarrow (c \in \mathbf{post}(e)) \vee (\exists c', e' \cdot e < c' \wedge c' < e' \wedge e' < c)$.
2. $c < e \Leftrightarrow (c \in \mathbf{pre}(e)) \vee (\exists c', e' \cdot c < e' \wedge e' < c' \wedge c' < e)$.
3. $c < c' \Leftrightarrow \exists e \cdot c < e \wedge e < c'$.
4. $e < e' \Leftrightarrow \exists c \cdot e < c \wedge c < e'$.

To describe processes, we use the restricted form of nets obtained by adding the following restrictions to Petri nets:

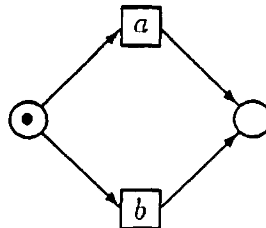
1. There exists a condition $c_0 \in C$ called the *initial condition* such that the only initial marking allowed is a single token in c_0.

2. If a condition c is a post-condition of two events e and e', then there exists a condition c' such that $c' < e$ and $c' < e'$, and for all $c'' \neq c'$ such that $c'' < e$ or $c'' < e'$, either $c' < c''$ or $c'' < c'$.

3. If two conditions c and c' are pre-conditions of an event e, then there exists an event e' such that $e' < c$ and $e' < c'$, and for all $e'' \neq e'$ such that $e'' < c$ or $e'' < c'$, either $e' < e''$ or $e'' < e'$.

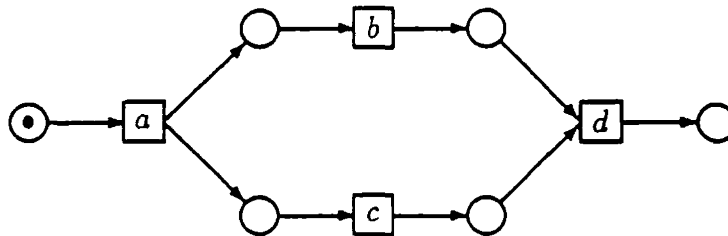
Restriction 2 above ensures that the alternate sequences in a branching section cannot interact until the end of the branch. Similarily, restriction 3 ensures that concurrent sections can not interact. These two taken together prohibit confusion.

Multiple arrows from or to a circle represent the start or end of a branching section in which one of several alternate sequences could occur. Thus,



represents a process in which either a or b (but not both) might occur.

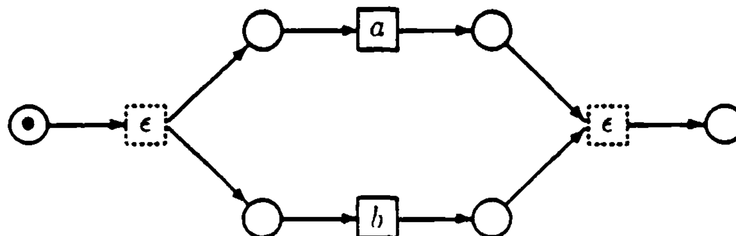
Multiple arrows leading from or to a box represent the start or end of a concurrent section, in which several sequences occur in parallel. In



a and d are separated by concurrent occurrences of b and c.

Branching preserves the number of tokens in the net. The start of a concurrent section multiplies tokens, generating one for each concurrent sequence, while the end of the concurrent section restores the original number of tokens. Since by restriction 1 there is only one token initially, we see that there is at most one token in each sequential portion of the net, and the net is therefore contact-free.

We sometimes need the null event ϵ to represent certain concurrent sequences. For example, to represent a process in which a and b can occur concurrently, we use

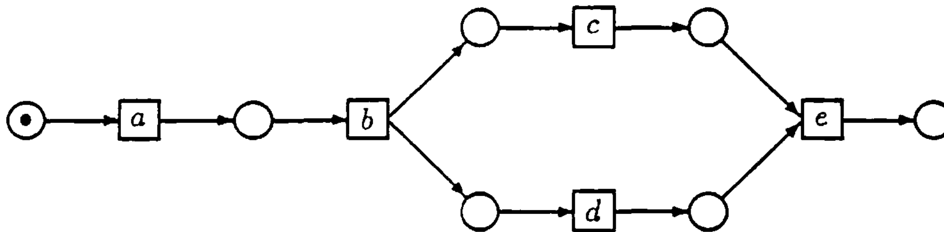


We use dashed boxes to represent the null events to indicate that they are artifacts of our representation, and have no significance in the actual process.

2.3 Observers

We will consider the behaviour of a process to be significant only as far as it interacts with the external world. The least intrusive interaction of the environment with a process is observation. An *observer process* records the events that occur during an execution of a process, subject to its own limitations. Thus, the observation reflects not only the behaviour of the observed process, but also the characteristics of the observer. The set of observations of all possible execution sequences of the process constitute a *complete observation* or *image*.

Observer processes may differ widely in their characteristics, and thus may yield images that also vary widely. For example, an observer process may be able to detect the occurrence of any event, but be unable to perceive the time at which they occurred. If such an observer encountered the process



the image produced would be simply the event set $\{a, b, c, d, e\}$. Observers may also be limited in the events that they can detect. An observer that is unable to detect the event d, but can detect all others, would see the above process as

One particularily interesting class of observer processes is that of *sequential observers*. A sequential observer can detect only one event at a time. If two or more events occur simultaneously in an execution of the observed process, the sequential observer arbitrarily assigns an ordering on them. Every execution of a process is seen as a sequential string of event occurrences, which is called a *trace* of the execution. We will usually use regular expressions to represent sequential observations of a process.

The set of events that an observer can detect is called its *event alphabet*. Occurrences of events not in this set are ignored by the observer. We shall represent a sequential observer with event alphabet A as $[A]$. If P is a process, we represent the complete observation of P by $[A]$ as $P \uparrow A$.

A sequential observer maps the set of all processes to the set of sequential processes on its event alphabet; sequential processes are mapped onto themselves.

Property 2.2 *For any process* P, $P \uparrow A \subseteq A^*$.

Property 2.3 $P \uparrow A = P \Leftrightarrow P \subseteq A^*$.

Sequential observers with larger event sets are in general more accurate observers than those with smaller event sets.

Property 2.4 *For any process P and event sets A and B, $A \subseteq B \Rightarrow (P \uparrow A) \uparrow B = P \uparrow B$.*

Property 2.5 *For any process P and event sets A and B, $(P \uparrow A) \uparrow B = P \uparrow (A \cap B)$.*

Definition 2.6 *The set inclusion operator on event alphabets defines a partial order on sequential observers*

$$[A] \sqsubset [B] \Leftrightarrow A \subset B.$$

If $[A] \sqsubset [B]$, we say that the observation of a process by $[A]$ is more accurate *than an observation by $[B]$.*

An observer with a larger event alphabet not only detects more events and therefore gives us a better approximation of the process, but it can also tell us what events do not occur in the process.

If the observation of a process yields an empty trace set, then it is obvious that none of the events in the event alphabet of the observer occur in the process. More generally, if any event of the event alphabet of an observer is missing in the observation of a process by that observer, then that event does not occur in the process.

Property 2.7 $P \uparrow A \subseteq B^* \Leftrightarrow P \uparrow (A \setminus B) = \emptyset$.

In our discussion of concurrent algebras, we shall use observers of varying capabilities to map the set of real processes onto the set of processes that the algebra can describe.

2.4 References

For the treatment of nets, we used [25]. The concept of observers was inspired by Milner's experiments on acceptors in chapter 1 of [21].

3 Trace Theory

Trace theory began as a tool for analyzing the behaviour of Petri nets. The algebras of strings and sets of strings had been very successful in the analysis of sequential systems. Trace theory attempted to describe concurrent systems in a manner which could allow the use of classical string oriented methods and techniques of sequential system theory.

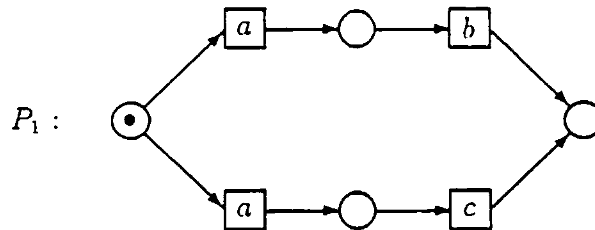
Today, trace theory forms a separate field in its own right, with links to the study of temporal logic, and the theory of graph grammars. In this chapter, we present a brief overview of basic trace theory, concentrating especially on those aspects most relevant to VLSI implementation. Our primary reference for this chapter is van de Snepscheut's thesis [27].

3.1 Observer

We shall use the class of sequential observers discussed in section 2.3. The sequential observer sees each execution of a process as a string of events — a trace. The complete observation is a set of such traces, called a *trace set*.

3.2 Basic Constructs

If an observation of a process P by a sequential observer $[A]$ gives a trace set R, then the process is said to be *approximated* by the ordered pair $< R, A >$, which is called a *trace structure*. A trace structure approximating a process contains not only the trace set of observations of the process, but also information about which events do not occur in the process. We shall denote by $\underline{t}T$ the trace set of a trace structure T, and by $\underline{a}T$ the event alphabet of T.



If P_1 is approximated by $< \{ab, ac\}, \{a, b, c, d\} >$, then $P_1 \uparrow \{a, b, c, d\} = \{ab, ac\}$ and $P_1 \uparrow \{d\} = \emptyset$.

Definition 3.1 *If for a trace structure P there exists an event A such that $\underline{t}P \uparrow \{a\} = \emptyset$, then P is said to be* a-restrictive.

In the above example, $P_1 \uparrow \{a, b, c, d\}$ is d-restrictive.

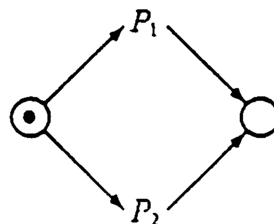
Note that the trace structure $< \{ab, ac\}, \{a, b, c\} >$ also approximates P_1, but is not d-restrictive.

Often, we have several observations of the same process that we wish to put together to form a better approximation of the process. These observations are usually observations of sub-processes that are to be composed to yield a description of the entire process.

Definition 3.2 (Concatenation) *If a process P consists of two sub-processes P_1 and P_2 such that P_1 occurs entirely before P_2,*

and if T_1 and T_2 are the observations of P_1 and P_2 by some sequential observers, then P is approximated by the sequential composition *of T_1 and T_2 written as $T_1 \cdot T_2$ given by* $< \{rs \mid r \in \underline{t}T_1 \land s \in \underline{t}T_2\}, \underline{a}T_1 \cup \underline{a}T_2 >$.

Definition 3.3 (Union) *If a process P consists of two alternate sub-processes P_1 and P_2 such that in any given execution either P_1 or P_2 is performed,*



and if T_1 and T_2 are the observations of P_1 and P_2 by some sequential observers, then P is approximated by the union *of T_1 and T_2 written as $T_1 + T_2$ given by $< \underline{t}T_1 \cup \underline{t}T_2, \underline{a}T_1 \cup \underline{a}T_2 >$.*

$\uparrow$ distributes over $+$ and $\cdot$, and $\cdot$ distributes over $+$.

Property 3.4 $(T_1 \cdot T_2)\uparrow A = (T_1 \uparrow A) \cdot (T_2 \uparrow A)$

Property 3.5 $(T_1 + T_2)\uparrow A = (T_1 \uparrow A) + (T_2 \uparrow A)$

Property 3.6 $T_1 \cdot (T_2 + T_3) = T_1 \cdot T_2 + T_1 \cdot T_3$

These two operators correspond to the concatenation and union of trace sets.

3.3 Parallel Composition

Parallel composition of trace structures is defined in such a way that the descriptions of the process given by the trace structures are merged.

Definition 3.7 (Weave) *If T_1 and T_2 are the observations of some process P by two different sequential observers, then P is better approximated by the* weave *of T_1 and T_2 written $T_1 \,\underline{w}\, T_2$ given by $< \{r \mid r \in A^* \wedge r \uparrow \underline{a}T_1 \in \underline{t}T_1 \wedge r \uparrow \underline{a}T_2 \in \underline{t}T_2\}, \underline{a}T_1 \cup \underline{a}T_2 >$.*

Note that T_1 is an approximation of $T_1 \,\underline{w}\, T_2$.
Weaving is symmetric, idempotent, and associative.

Property 3.8 $T_1 \,\underline{w}\, T_2 = T_2 \,\underline{w}\, T_1$

Property 3.9 $T_1 \,\underline{w}\, T_1 = T_1$

Property 3.10 $(T_1 \,\underline{w}\, T_2) \,\underline{w}\, T_3 = T_1 \,\underline{w}\, (T_2 \,\underline{w}\, T_3)$

Event restriction is preserved by weaving.

Theorem 3.11 *If a trace structure P is e-restrictive for some event e, then for any trace structure Q, $P \,\underline{w}\, Q$ is also e-restrictive.*

Proof : $\underline{t}P \uparrow \{e\} = \emptyset$
By property 2.7, $\underline{t}P \subseteq (\underline{a}P \setminus \{e\})^*$
From the definition of weaving, $\underline{t}(P \,\underline{w}\, Q) \uparrow \underline{a}P \subseteq (\underline{a}P \setminus \{e\})^*$
By property 2.7, $\underline{t}(P \,\underline{w}\, Q) \uparrow \{e\} = \emptyset$.

We define one additional parallel composition operator.

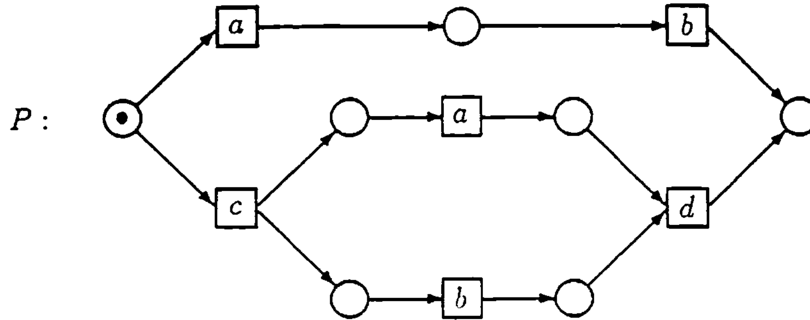
Definition 3.12 (Blend) *If T_1 and T_2 are the observations of a process by two different sequential observers, then the* blend *of T_1 and T_2 written $T_1 \,\underline{b}\, T_2$ is given by*

$$(T_1 \,\underline{w}\, T_2) \uparrow ((\underline{a}T_1 \cup \underline{a}T_2) \setminus (\underline{a}T_1 \cap \underline{a}T_2)) \quad .$$

The blend of two trace structures corresponds to composition of the sub-processes represented by the trace structures, together with the elimination of all common events. These events are *internal* events, and so are hidden from further external observation or interaction.

Blending is symmetric, but not idempotent or associative.

3.4 Example



The following are all approximations of of P

$T_1 =< \{a, cad\}, \{a, c, d\} >$
$T_2 =< \{b, cb\}, \{b, c\} >$
$T_3 =< \{cd\}, \{c, d\}$
$T_4 =< \{b, bd\}, \{b, d\} >$
$T_5 =< \{ab, cab, cba\}, \{a, b, c\} >$
$T_4 \, \underline{w} \, T_2 =< \{b, cbd\}, \{b, c, d\} >$
$T_5 \, \underline{w} \, T_4 \, \underline{w} \, T_2 =< \{ab, cabd, cbad\}, \{a, b, c, d\} >$
$T_4 \, \underline{w} \, T_1 =< \{ab, ba, bcad, cbad, cabd\}, \{a, b, c, d\} >$
$T_5 \, \underline{w} \, T_4 \, \underline{w} \, T_1 =< \{ab, cbad, cabd\}, \{a, b, c, d\} >$
$T_6 = a \cdot b + c \cdot (a \, \underline{w} \, b) \cdot d$, where we represent the singleton trace structure of an event e, $< \{e\}, \{e\} >$, by the event.

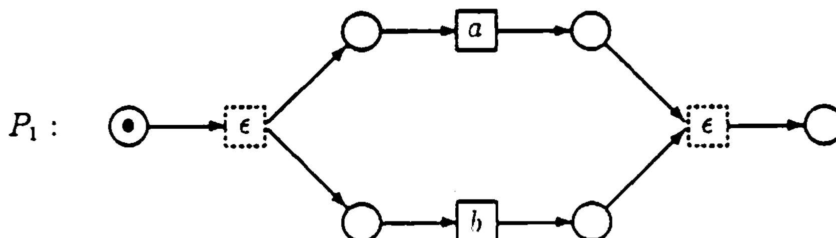
3.5 Implementation

Implementation of processes described by trace theory is relatively straightforward. If we limit trace sets to regular sets, any scheme for implementing finite automata can be used. Van de Snepscheut suggests an implementation scheme in his thesis, but this is complicated by several additions to basic trace theory that we have not considered in this overview.

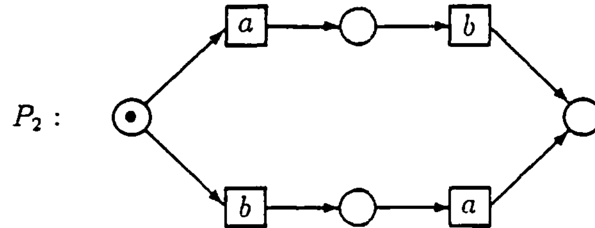
One problem of using a straightforward finite automata implementation is the exponential blow-up in the number of states due to enumeration of all possible execution sequences of independent events. We are working on a scheme based on Anantharaman et al [1] that will allow us to implement the weave operation directly without enumeration.

3.6 Limitations and Extensions

In trace theory, the process



is indistinguishable from the process



This is a serious flaw, since in P_1, a and b are independent, while in P_2, they are mutually exclusive — one of the most common forms of dependence.

This problem stems from the fact that sequential observation imposes mutual exclusion on the observed events. Thus information about true mutual exclusion is lost, and independence and mutual exclusion become indistinguishable.

Several attempts at remedying this problem have been made. However, all lead to much more complex algebras and lose the basic advantage of trace theory — simplicity. Mazurkiewicz [18] uses a structure that contains dependency information in the form of a dependency graph for the events in the structure. This scheme has several advantages, the chief being the accurate representation of independence. Traces are defined only upto equivalence under independence. Thus, if a and b are independent, $cabd \equiv cbad$. The corresponding trace is the equivalence class $[cabd]$.

However, while the weaving operator remains more or less unchanged, it is not clear how the union and concatenation (sequential composition) operations should be defined to preserve local independence information.

Black [3] points out that the Trace Theory limited to finite traces lacks the expressive power to specify the notion of fairness. He suggests extending Trace Theory to infinite traces, utilising the theory developed by Park [23] and others.

3.7 References

The treatment of trace structures broadly follows chapter 1 of van de Snepscheut's thesis [27]. Theorem 3.11 is previously unpublished.

4 Path Expressions

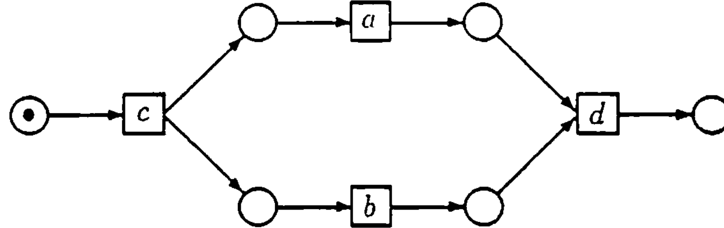
The use of path expressions in the description and analysis of concurrent systems was first suggested by Campbell and Habermann [5,10,9]. Since then, the growth of the field has been sporadic. There was initial interest in using path expressions to describe and analyze software systems [4,26], but most such efforts were abandoned when less limited algebras became available. More recently, path expressions have been used to describe systems for VLSI implementation [1].

4.1 Observers

Path expression theory is closely related to trace theory. Again we use the concept of sequential observers.

4.2 Basic Constructs

A path is a sequential observation of a process by an observer whose event alphabet does not contain any mutually independent events. This restriction ensures that no independence information is lost during observation. We write a path as **path** R **end**, where R is a regular expression representing the set of sequences produced by the observation.



In the process P shown above, (a, b) is the only pair of independent events. Some paths describing P are

$P \uparrow \{a, c, d\}$: **path** cad **end**

$P \uparrow \{b, d\}$: **path** bd **end**

$P \uparrow \{b, c\}$: **path** cb **end**

Note that an observation by $[\{a, b, c\}]$ is not a path, since the event alphabet contains independent events. The resultant trace set is $\{cab, cba\}$ which incorrectly shows a and b to be mutually exclusive.

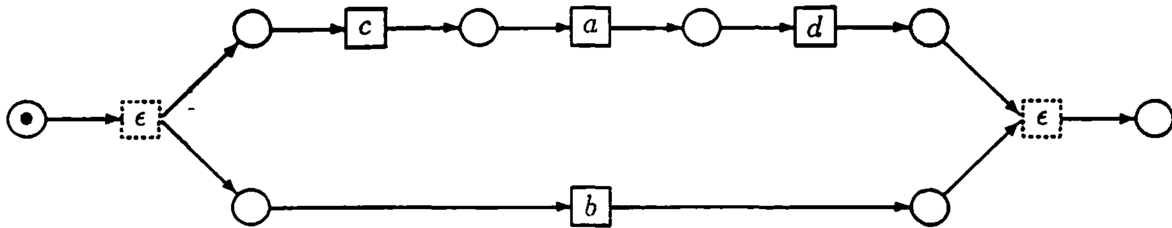
There are no equivalents of the concatenation and union operations for path expressions. The only composition of path expressions possible is parallel composition.

4.3 Parallel Composition

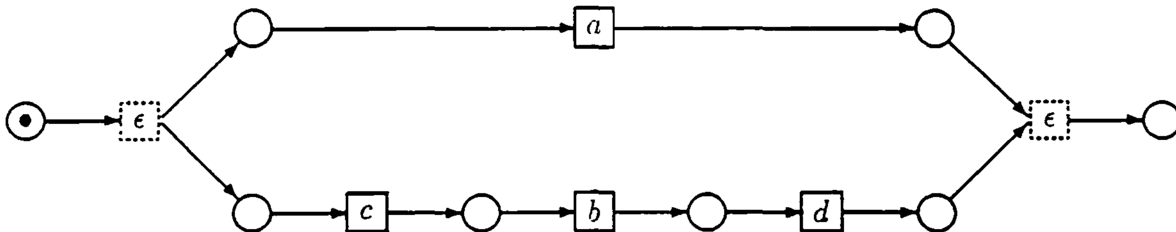
Path expression theory does not contain an explicit composition operator. This is because the parallel composition of two path expressions in general is not a sequential process, and cannot be described by a single path expression. Instead, we represent the composite process simply by writing the path expressions together. The process P of the example in the previous section can be described by the path expression system

path cad **end**
path cbd **end**

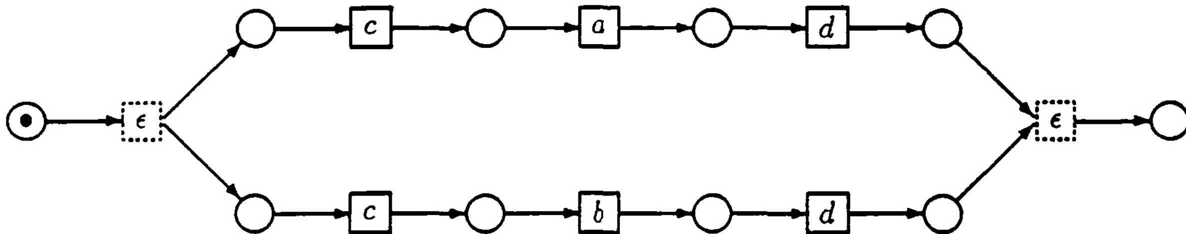
One way to understand the composition of path expressions is to consider each path expression S which is added to a path expression system as adding a restriction on the events in $\underline{a}S$. In the example above, we start initially with a totally unconstrained set of events $\{a, b, c, d\}$. The first path **path** cad **end** taken by itself restricts the events $\{a, c, d\}$ to occur in a specific order, but leaves b unconstrained.



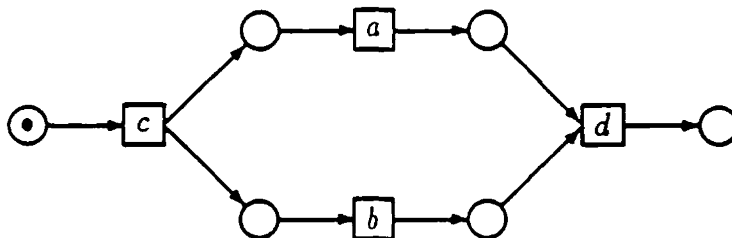
The second path taken by itself restricts only $\{b, c, d\}$, leaving a unconstrained.



When the two paths are composed, the restrictions are added. As a result, a and b are each constrained with respect to c and d, but are unconstrained with respect to each other. This gives,



or simplifying,



The addition of a third path path ab end to the path expression system would further constrain it to give

4.4 Example

In this section, we present an extended example illustrating how a system description in terms of path expressions is produced.

The system we consider consists of two processors, a memory unit, and an I/O device connected by a bus as shown in Fig. 1.

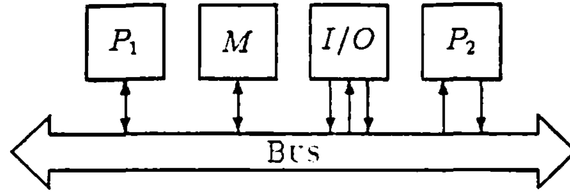


Figure 1:

A processor can read from or write to the memory unit, and input or output data on the I/O device. We represent by $R_i, W_i, I_i, O_i, i \in \{1,2\}$ the read, write,input, and output operations performed by processor P_i.

The memory unit can only handle one action at a time. The I/O device can handle one output action and any number of input actions simultaneously. P_1 is connected to the bus by a single two-way data port, and can therefore perform only one action at a time. P_2 is connected to the bus via 2 one-way ports, one in each direction. For simplicity, we assume that the bus has no restrictions on the number of actions that can occur simultaneously on it.

Each processor runs a pair of concurrent processes. One inputs data from the I/O device and stores it in memory, while the other takes data from the memory unit and outputs it on the I/O device.

To describe this system, we start from a set of totally unconstrained events, and add constraints in the form of paths for each processor or device.

The event set is $\{R_1, W_1, I_1, O_1, R_2, W_2, I_2, O_2\}$.

Constraints on the memory unit :

$$\textbf{path } (R_1 + R_2 + W_1 + W_2)^* \textbf{ end}$$

Constraints on the I/O unit :

$$\textbf{path } (O_1 + O_2 + I_1)^* \textbf{ end}$$
$$\textbf{path } (O_1 + O_2 + I_2)^* \textbf{ end}$$

Note that I_1 and I_2 remain independent.

Constraints on on P_1 :

$$\textbf{path } (R_1 + W_1 + I_1 + O_1)^* \textbf{ end}$$
$$\textbf{path } (I_1 W_1)^* \textbf{ end}$$
$$\textbf{path } (R_1 O_1)^* \textbf{ end}$$

Constraints on P_2 :

$$\textbf{path } (R_2 + I_2)^* \textbf{ end}$$
$$\textbf{path } (W_2 + O_2)^* \textbf{ end}$$
$$\textbf{path } (I_2 W_2)^* \textbf{ end}$$
$$\textbf{path } (R_2 O_2)^* \textbf{ end}$$

This completes the description of the system.

In general, each device has several paths that enforce mutual exclusion, while each processor has paths that enforce mutual exclusion or indicate sequences of operations.

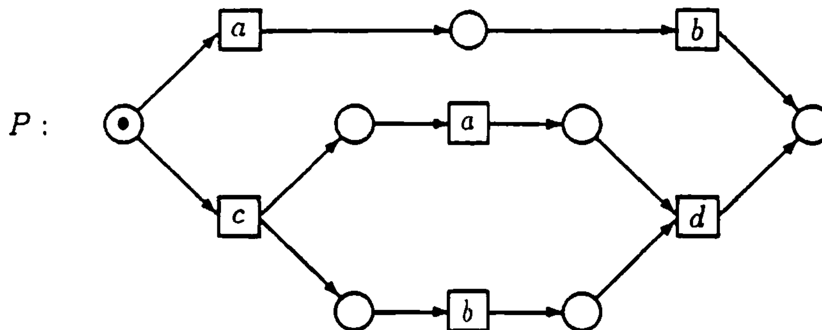
4.5 Implementation

Path expressions are relatively easy to implement in hardware. A system of path expressions consists essentially of a set of regular expressions operating in parallel. A general implementation scheme would therefore be to use a standard implementation scheme for regular expressions, and add additional circuitry to force parallel operation.

Several methods of implementing regular expressions have been described in the literature [7,27,8]. Li and Lauer [17] use PLA's to implement the state machine corresponding to each path, and add an additional PLA to handle synchronization. Foster's technique of using cells organized in a tree-like structure is particularily compact and amenable to our purposes. The Miss Manners synchronizer generator [2], based on the scheme described in [1], compiles a set of path expressions into a set of trees, together with an arbiter and additional circuitry to enforce parallelism.

4.6 Limitations

Since parallel composition is the only operation allowed on path expressions, the theory has difficulty describing correctly processes that are the union or concatenation of two sub-processes.



For example, consider the process P shown above. The path expression system

$$\textbf{path } a + cad \textbf{ end}$$
$$\textbf{path } b + cbd \textbf{ end}$$

correctly describes the portion of P in which a and b are independent, but at the cost of losing the sequence information in the other portion. Adding the path

$$\textbf{path } ab \textbf{ end}$$

provides this information, but now a and b are no longer independent in the other portion.

No path expression system can correctly describe P, since the sequence ab can only be ensured by having a path containing both a and b, which prevents a and b from being

independent. However, we can represent a sequential approximation of P by using the trace theory method of enumeration.

$$\text{path } ab + cabd + cbad \text{ end}$$

4.7 References

Our primary references for this chapter are [2.1].

5 A Calculus of Communicating Systems

Since Milner described CCS in [21], the calculus has been extensively studied, and a vast literature has been built up. In this section, we overview very briefly a small portion of basic CCS. In its standard form, CCS is better suited to the analysis of software systems than to the implementation of systems in hardware. However, several concepts introduced by CCS have relevance to our study, and we shall concentrate on these features.

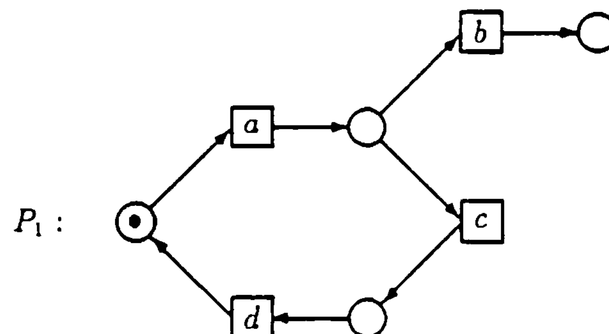
5.1 Observer

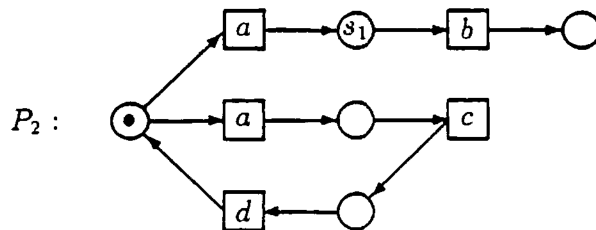
We extend the purely passive observers of the last two chapters to an *interactive observer*. This observer does not merely watch and record event occurrences, but instead conducts experiments on the process to determine its behaviour. It does so by repeatedly asking the process to perform specific events, and then noting either the occurrence or non-occurrence of the event. We shall see a little later that this allows us to distinguish between processes that are indistinguishable to a passive observer.

We shall specifically use the *interactive sequential observer*, which is the interactive version of the *sequential observer*.

An interactive sequential observer with events alphabet A experiments on a process P as follows. The observer requests P to perform some event $a \in A$. If P can perform the event in its current state, it does so and the observer notes the outcome. If P is unable to perform a in its current state, it does nothing and informs the observer accordingly. The observer then tries other events in A. We shall assume that P can be reset to its initial state and rerun as often as necessary to determine its behaviour completely.

Consider the processes P_1 and P_2:





A passive sequential observer with event alphabet $\{a, c, d\}$ would see both processes as $(acd)^*a$. An interactive sequential observer with the same event alphabet observing P_1 will always be able to successfully request some event. When it observes P_2, however, it may sometimes find that no request can be satisfied — when the token is in s_1. So, the interactive observer is able to distinguish between the two proceses which differ only in their internal structure.

In practice, this difference may be critical in the implementation of the process if it is required to communicate with other processes. If the process is implemented as in P_2, it may *deadlock* under certain conditions.

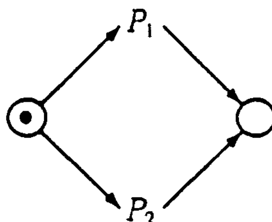
5.2 Basic Constructs

It must be kept in mind throughout this chapter that the observations of processes by interactive sequential observers are not sets of strings, since the distributive law $a(b + c) = ab + ac$ does not hold. We shall, however, use a notation similar to that of regular expressions to describe observations.

Definition 5.1 (Concatenation) *If a process P consists of two sub-processes P_1 and P_2 such that P_1 occurs entirely before P_2,*

and if T_1 and T_2 are the observations of P_1 and P_2 by some interactive sequential observers, then P is approximated by the concatenation of the sets T_1 and T_2 written as T_1T_2 given by $\{rs \mid r \in T_1 \wedge s \in T_2\}$.

Definition 5.2 (Union) *If a process P consists of two alternate sub-processes P_1 and P_2 such that in any given execution either P_1 or P_2 is performed,*



and if T_1 and T_2 are the observations of P_1 and P_2 by some interactive sequential observers, then P is approximated by the union of the sets T_1 and T_2 written as $T_1 + T_2$.

$+$ is associative and commutative.

5.3 Parallel Composition

Consider two processes that interact in a manner such that each behaves like an interactive observer of the other. Each process can request the other to perform some action, which the other does if possible. We would like to then consider the composition of the two processes as a single process, and investigate its behaviour.

We assume a set of events $a, b, c, d, \ldots$ and a disjoint set of co-events $\bar{a}, \bar{b}, \ldots$ such that the $(\bar{\cdot})$ operator is bijective. For any event a, $\bar{a}$ corresponds to a request for a. We correspondingly expand the capabilities of interactive observers to be able to detect and satisfy event requests.

Let P_1 and P_2 be two processes such that P_1 can perform a and P_2 can perform $\bar{a}$. P_1 can perform a in response to a request either from P_2, or from the observer. Similarily, a request $\bar{a}$ from P_2 can be satisfied either by P_1 or by the observer. If the processes perform the events in response to each other, the observer is unable to observe either event or response (since it was not involved).

Let P_1 consist of an occurrence of a followed by a sub-process P_1', and P_2 an occurrence of $\bar{a}$ followed by P_2' : $P_1 = aP_1'$ and $P_2 = \bar{a}P_2'$.

The composition $P_1 \mid P_2$ can have three possibilities:

1. P_1 performs a in response to a request from the observer

$$a(P_1' \mid (\bar{a}P_2'))$$

2. P_2 has the request a satisfied by the observer

$$\bar{a}((aP_1') \mid P_2')$$

3. P_1 and P_2 interact directly. We represent this unobservable communication by τ.

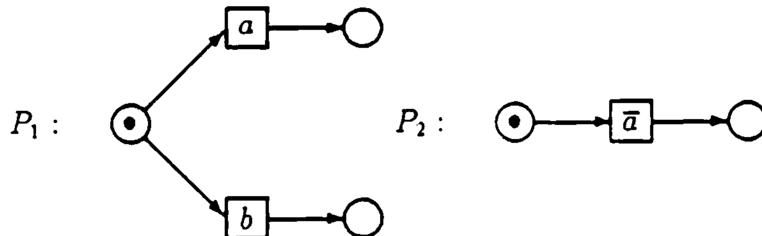
$$\tau(P_1' \mid P_2')$$

To complete the definition of $\mid$, we add, for $b \neq \bar{a}$,

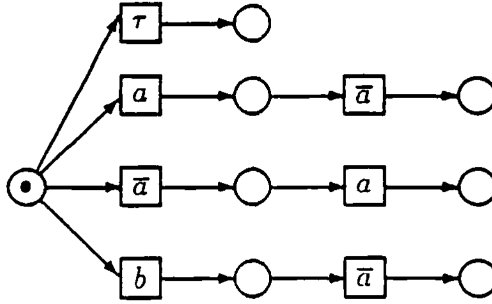
$$(aP_1') \mid (bP_2') = a(P_1' \mid (bP_2') + b((aP_1') \mid P_2')$$

Although τ is not directly observable, it is in fact detectable by the effects of the change in state of the process.

If P_1 and P_2 are as shown,



then $P_1 \mid P_2$ is



If τ occurs, the observer will be denied all further requests for events.

Parallel composition is symmetric and associative, but not idempotent.

Property 5.3 $P_1 \mid P_2 = P_2 \mid P_1$

Property 5.4 $(P_1 \mid P_2) \mid P_3 = P_1 \mid (P_2 \mid P_3)$

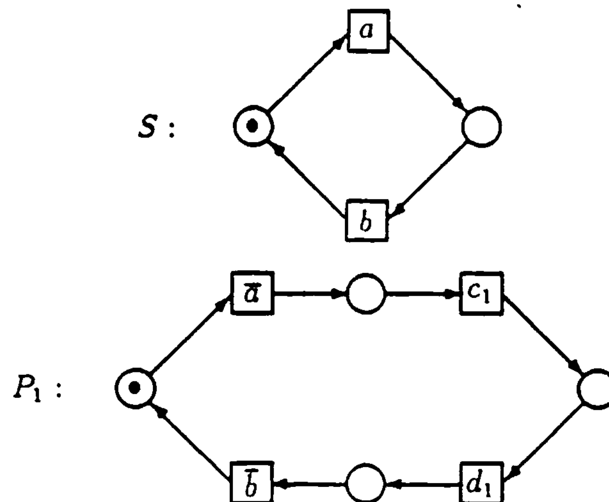
If in this example, a and $\bar{a}$ were events that were only used to communicate between P_1 and P_2, we would like to prevent other processes from interacting with them. We introduce the operation of *restriction* ($\backslash$).

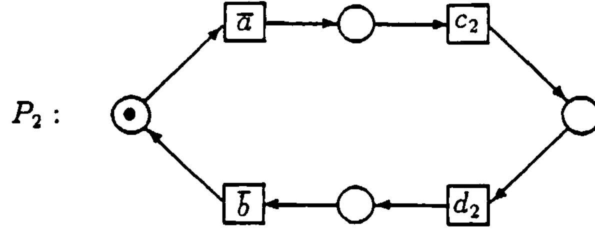
If P is a process and B is an event set, $P \setminus B$ restricts the events in B (and their co-events) making them unobservable. Since events can only occur through interaction with requests, restriction in effect prevents unpaired events or requests from occurring.

In the example above, if a and $\bar{a}$ are restricted, they cannot occur, and each of the three sequences that contain either of these is prohibited since it contains impossible events. Thus, $(P_1 \mid P_2) \setminus \{a\} = \tau$.

The combination of parallel composition and restriction is closely related to the blending operation of trace theory, but separating the two allows parallel composition to be associative.

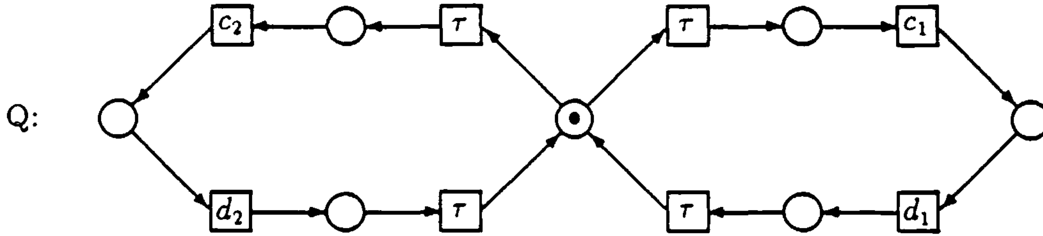
5.4 Example





$S = abS$, $P_1 = \bar{a}c_1d_1\bar{b}P_1$, and $P_2 = \bar{a}c_2d_2\bar{b}P_2$.
We compose all three processes,

$$
\begin{aligned}
Q &= (S \mid P_1 \mid P_2) \setminus \{a, b\} \\
&= \tau((c_1d_1\bar{b}P_1 \mid P_2 \mid bS) \setminus \{a, b\}) \\
&\quad + \tau((P_1 \mid c_2d_2\bar{b}P_2 \mid bS) \setminus \{a, b\}) \\
&= \tau c_1 d_1 \tau((\bar{b}P_1 \mid P_2 \mid bS) \setminus \{a, b\}) \\
&\quad + \tau c_2 d_2 \tau((P_1 \mid \bar{b}P_2 \mid bS) \setminus \{a, b\}) \\
&= \tau c_1 d_1 \tau((P_1 \mid P_2 \mid S) \setminus \{a, b\}) \\
&\quad + \tau c_2 d_2 \tau((P_1 \mid P_2 \mid S) \setminus \{a, b\}) \\
&= \tau c_1 d_1 \tau Q + \tau c_2 d_2 \tau Q
\end{aligned}
$$



S behaves as a binary semaphore, with a and b being the request and release operations. An n-bounded semaphore can be obtained simply by composing n copies of S. This example is from Milner [21].

5.5 Implementation

Implementation schemes for systems described in CCS present several problems. One chief problem is that since the distribution law does not hold ($a(b+c) \neq ab+ac$), expressions of CCS cannot be regarded as sets of strings. Thus, standard finite automata implementation techniques cannot be used.

5.6 Limitations and Extensions

The choice operator + in CCS has unclear semantics. It exhibits a mixture of two forms of non-deterministic behaviour — often referred to as *internal* and *external non-determinism* [11].

External non-determinism is exhibited in the process $aP_1 + bP_2$. If the observer (or another interacting process) requests a, the process will subsequently behave as described

by P_1, while a request of b causes the process to continue as P_2. (Note that since we use sequential observers, the process cannot receive requests for both a and b simultaneously.)

Internal non-determinism is exhibited in $aP_1 + aP_2$ and $aP_1 + \tau P_2$. In the first, after a request for a, the process will continue as either P_1 or P_2. In the second, a request for a may sometimes be granted and sometimes denied.

De Nicola and Hennessy point out that the need for τ to represent internal operations which should be invisible is counterintuitive. They suggest replacement of $+$ and τ with two combinators, $\oplus$ to represent internal non-determinism, and $\Box$ to represent external non-determinism. While the resulting algebra has simpler operational semantics, certain concepts of CCS, notably *observational equivalence*, cannot be expressed adequately in it.

Costa and Stirling [6] describe a variation of CCS that allows only fair execution sequences. While the algebra is of interest, it is considerably bulkier and more awkward that Milner's CCS.

5.7 References

Our primary references are Milner's CCS papers [22,19,21,20].

6 A Concurrent Algebra for Finite Events

In our discussion of concurrent processes, we have hitherto made the implicit assumption that events occur instantaneously — they have an infinitesimal duration. This assumption is a good approximation if we are dealing with purely sequential systems (as in Trace Theory and CCS), or if the duration of the events is negligible compared to the inter-event interval. In practice, real events have distinct start and end points and may have durations that are large compared to the inter-event time. This is especially true when we are dealing with micro-events such as the raising or lowering of individual lines. To be able to describe systems for VLSI implementation, we should be able to deal adequately with such micro-events, and hence it is important to be able to describe and analyze systems of events of finite duration (or *finite events*).

In this chapter, we describe briefly an algebra, called CAFE, to deal with finite events. We derive a minimal set of relations and present a language that uses them to describe processes.

6.1 Observer

Algebras of instantaneous events (or *point events*) have only two possible relations : precedence, and simultaneity. In contrast to this, occurrences of finite events can be related to each other in several ways. We will first list all possible combinations of two finite events, and then derive a minimal set of relations that can describe all the cases. We shall use an observer that detects relationships between the start and end times of events, which we call a *finite observer*. For an event a we represent its start and end time by a_s and a_e. We

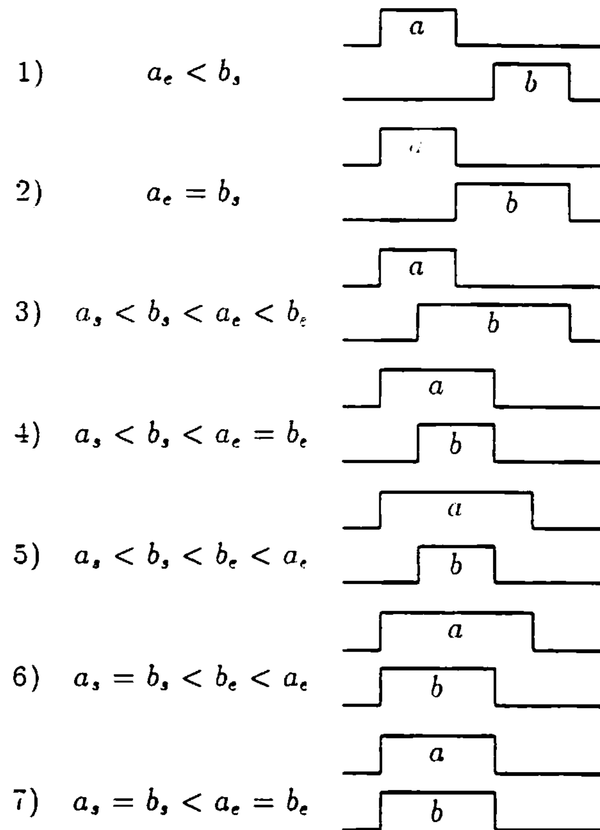


Figure 2:

shall use the relations <, >, and = to indicate relationships between these points. Thus, $a_e < b_s$ represents the fact that a ends before b starts.

When two finite events occur, one of the conditions shown in Fig.2 holds. We have omitted another set of six cases which are obtained by interchanging a and b. We represent all 13 possible cases in Fig. 3. A node numbered by x where $x > 7$ represents the condition obtained by interchanging a and b in condition $14 - x$ of Fig. 2. Nodes are joined by a line if they differ by one step in any direction.

A finite observer that could detect each of these 13 cases individually would be much too cumbersome to use. We would like to derive a smaller set of relations that can produce any set of cases through intersection and union.

A relation would be represented on this chart as a region enclosing one or more nodes. In the interests of implementability, we will restrict ourselves to continuous and convex relations. A continuous relation is one which corresponds to a single connected region. A convex relation corresponds to a convex region. Thus, the relation containing nodes $\{2, 3, 4, 8\}$ is continuous, but not convex. To make it convex, we could add node 7. We want to find a set of relations that allow us to refer to any node by superimposing one or more relations. In terms of regions on the chart, we want to find a set of partitions of the

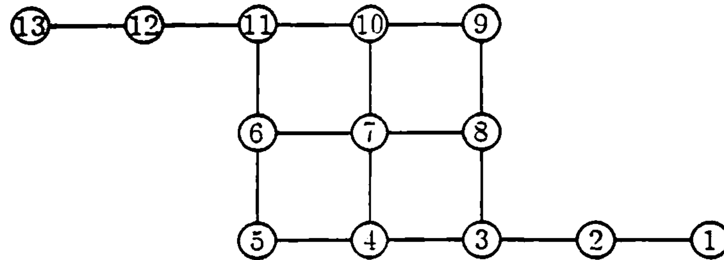


Figure 3:

graph that can isolate every node.

Theorem 6.1 *Six continuous and convex relations are necessary and sufficient to extract every node.*

Proof Outline : *The relation* $\{1\}$ *must be included, since this is the only one that separates nodes 1 and 2.*

Similarily, either $\{1, 2\}$ *or* $\{2\}$ *must be included, to separate nodes 2 and 3.*

We need a relation to separate the pairs $(3, 4), (7, 8), (9, 10)$ *and another to separate* $(3, 8), (4, 7), (5, 6)$.

Finally, we need relations to extract $(6, 7, 8)$ *and* $(4, 7, 10)$.

Figs 4 and 5 give the set of relations that we shall use. This particular set was chosen because each relation has a straightforward semantics.

$<_p$ is read ***strictly precedes***, $\leq_p$ is *precedes*, $\lll_s$ is *starts strictly within*, $\lll_e$ is *ends strictly within*, and $\lhd_s$ and $\lhd_e$ are *starts within* and *ends within*.

When describing systems for VLSI implementation, we are usually concerned with producing ***delay-insensitive*** descriptions. In this case, it is no longer meaningful to talk about simultaneity. So we use a *delay-insensitive observer* which can detect only the three strict relations $<_p \lll_s$, and $\lll_e$.

6.2 Basic Constructs

As with path expressions, we use the idea of starting with an initially unrestricted set of events, and adding restrictions in the form of relations.

	○	$b <_p a$	$b \leq_p a$	$b \lll_s a$	$b \lhd_s a$	$b \lll_e a$	$b \lhd_e a$	○
	●	$a <_p b$	$a \leq_p b$	$a \lll_s b$	$a \lhd_s b$	$a \lll_e b$	$a \lhd_e b$	●
1	$a_s < a_e < b_s < b_e$	●	●					
2	$a_s < a_e = b_s < b_e$		●		○		●	
3	$a_s < b_s < a_e < b_e$			○	○	●	●	
4	$a_s < b_s < a_e = b_e$			○	○		●○	
5	$a_s < b_s < b_e < a_e$			○	○	○	○	
6	$a_s = b_s < b_e < a_e$				●○	○	○	
7	$a_s = b_s < a_e = b_e$				●○		●○	
8	$a_s = b_s < a_e < b_e$				●○	●	●	
9	$b_s < a_s < a_e < b_e$			●	●	●	●	
10	$b_s < a_s < a_e = b_e$			●	●		●○	
11	$b_s < a_s < b_e < a_e$			●	●	○	○	
12	$b_s < a_s = b_e < a_e$		○		●		○	
13	$b_s < b_e < a_s < a_e$	○	○					

Figure 4:

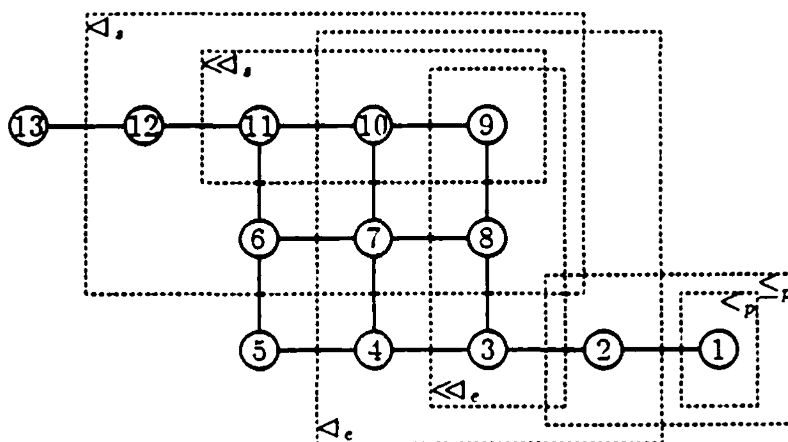
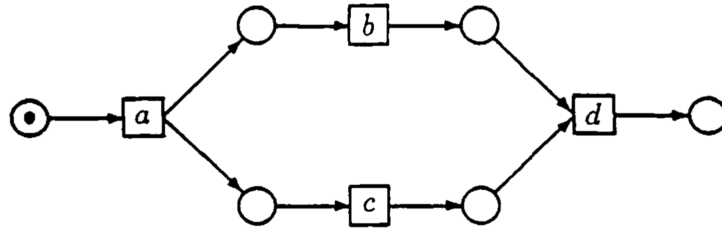


Figure 5:



A description of the process shown above is built up in steps as follows:

1. $\{a, b, c, d\}$
2. $(a <_p b, a <_p c : \{a, b, c, d\})$
3. $(b <_p d, c <_p d : (a <_p b, a <_p c : \{a, b, c, d\}))$
   If c occurred entirely within b, we add
4. $(c \lll_s b, c \lll_e b : (b <_p d, c <_p d : (a <_p b, a <_p c : \{a, b, c, d\})))$

$+$ has its usual significance of union.
We also use $a \,\mathbf{rel}\, b$ as a short hand for $(a \,\mathbf{rel}\, b : \{a, b\})$.

6.3 Parallel Composition

As with path expressions, there is no explicit parallel composition operator. Events that are not mutually restricted are concurrent. However, unlike path expressions, the composition of two sub-processes can be expressed by a single expression.

$$\{a, b\} = a <_p b + a \lhd_e b + a \lhd_s b + b <_p a$$

6.4 Example

We consider the case of a processor P doing a memory write using a 4-phase, 2-line bus protocol. The following sequence occurs :

1. P raises the request line to gain control of the bus.
2. The bus controller raises the acknowledge, granting the bus.
3. P puts the data and address on the bus
4. P raises the write line to write the data into memory.
5. P lowers write.
6. P releases the data and address buses.
7. P lowers request
8. The bus controller lowers acknowledge.

The events we use are linked to the states of the lines, rather than state transitions. Thus, r,a, and w represent the request, acknowledge, and write lines in the raised state, while d represents the activation of the data and address buses.

We list the relations that hold :

1. $a \lll_s r$
2. $d \lll_s a$
3. $w \lll_s d$
4. $w \lll_e d$
5. $d \lll_e a$
6. $r \lll_e a$

So, we can describe the process by :

$$(a \lll_s r, r \lll_e a : (d \lll_s a, d \lll_e a : (w \lll_s a, w \lll_e a : \{r, a, w, d\})))$$

6.5 Implementation

We are working on an implementation scheme for a subset of CAFE based on that of [1] for the path expression language. The basic scheme is to implement the sequential portions using the tree structure, and then to add on extra cells to enforce the other relations.

6.6 Limitations

We have not fully developed the algebra yet. At present, it is rather cumbersome.

6.7 References

The material in this chapter is previously unpublished. But we were influenced in the developement of the algebra by the work on the use of partial orders in describing concurrency of Janicki [12,13], Knuth [14], and Pratt [24], and by Lamport's papers on mutual exclusion [15,16].

7 Discussion

In this section, we compare the algebras that we have described in terms of their treatment of independence, parallel composition, and inter-event dependency. It is to be noted that path expressions and trace theory are closely related, and differ only in their treatment of independence. We give a theorem describing this relationship in section 7.1.

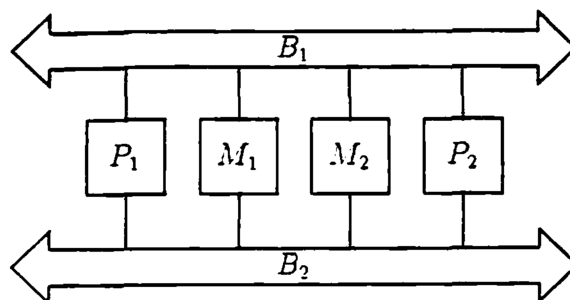


Figure 6:

7.1 Independence

We say two events are independent when each can occur without regard for the occurrence or non-occurrence of the other. In some systems, a pair of events can be independent under some conditions and interdependent under others. Two events are said to be globally independent when they are independent under all conditions — when each occurrence of one is independent of every occurrence of the other. When certain occurrences of the events are independent but not others, the events are said to be locally independent.

Consider the system shown in Fig. 6. Two processors, P_1 and P_2 are connected via two buses B_1 and B_2 to a pair of memory units M_1 and M_2. We will denote by R_{ij} a read operation on M_j by P_i, i and j $\in \{1, 2\}$. The processors, memory units, and buses can each handle only one operation at a time.

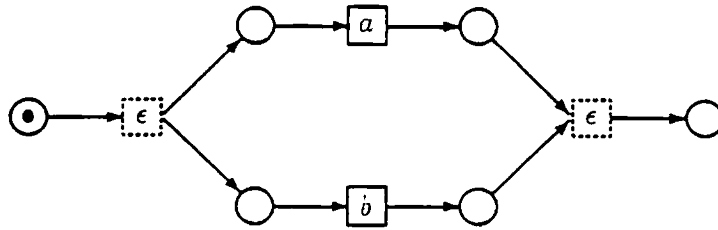
Then, (R_{11}, R_{22}) and (R_{21}, R_{12}) are pairs of globally independent events. These are the only independent pairs of events, since all other pairs are made mutually exclusive by restrictions on the processors or memory units.

Now, consider the same system, but assume bus B_1 is not always available, either due to hardware faults, or to pre-emption by some other part of the system. When both buses are available, R_{11} and R_{22} are independent, but when only one bus is available, they are mutually exclusive. Thus, R_{11} and R_{22} are now locally independent.

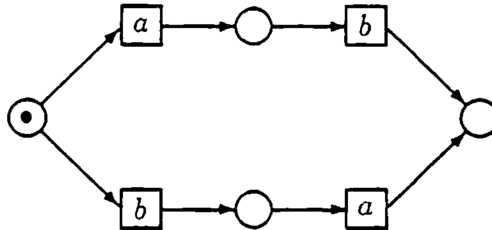
The implementation and analysis of systems containing local independence are in general hard problems. Most concurrent algebras avoid this problem in one way or the other.

Since path expressions allow parallel composition only on the top level, only global independence can occur. This restriction leads to the inability of path expression to adequately describe processes having local independence as described in section 4.6.

Both trace theory and CCS can describe systems containing local independence. However, they deal with it by using the *interleaving semantics*. In this model, two events that occur independently are considered to occur one at a time in an arbitrary order. Thus, in this semantics, the process



in which a and b are independent is equivalent to the process



in which a and b occur in sequence. If two processes occur independently, their events occur in sequence in some arbitrary interleaving.

This model of independence allows the preservation of local independence information, but at the cost of making independence indistinguishable from mutual exclusion (refer section 3.6).

CAFE also allows the description of systems containing local independence but does not use the interleaving semantics. Instead, the independence information is handled in a way analogous to the way path expressions handle global independence. This leads to an increase in the complexity of analysis, which requires the use of labelled partial ordered sets [18,24].

The primary difference between trace theory and path expressions is the use of the interleaving semantics by trace theory to model independence. We end this section with a theorem that describes the relationship between a path expression and the trace theoretic descriptions of a system.

If S is a path, let $\underline{a}S$ denote the events in the path, and $\underline{t}S$ the set of traces described by the path. Then, each path expression S corresponds to a trace structure $\mathcal{T}(S) = <\underline{t}S, \underline{a}S>$. If W is a system of path expressions, let $W \uparrow A$ denote the trace structure produced by the observation by the sequential observer $[A]$ of the system described by W. In general, the resultant trace structure does not correspond to a path in W.

Theorem 7.1 *If a process is represented by a system of path expressions W, and if A is the set of all events in W, then the observation by $[A]$ of the system described by W is equal to the weave of the individual observations of every path in W.*

Or, if $W = \{p_1, p_2, \ldots, p_n\}$ and $A = \underline{a}p_1 \cup \underline{a}p_2 \cup \ldots \cup \underline{a}p_n$,
then $W \uparrow A = \mathcal{T}(p_1) \,\underline{w}\, \mathcal{T}(p_2) \,\underline{w}\, \ldots \,\underline{w}\, \mathcal{T}(p_n)$

Proof Outline : *If a pair of events (a, b) are mutually restricted by W, then they must occur together in some path p in W. They therefore occur in sequence in some trace of*

$\underline{t}T(p)$. This sequence is preserved by weaving, and therefore appears in the resultant of the RHS.

If a and b are mutually unrestricted by W, then they never occur in the same path. Hence, they occur together in a trace on the RHS only through weaving, and therefore $RHS\P\{a,b\} = ab + ba = LHS\P\{a,b\}$.

7.2 Parallel Composition

The major difference between CCS and the other three algebras is their treatment of parallel composition. In CCS, the composition of two sub-processes represents the process formed by connecting the two subprocesses and allowing them to interact. In other words, in CCS parallel composition is very closely related the physical construction of the system from independent modules. In each of the other algebras, parallel composition corresponds to the interaction of two descriptions of the *same* global process. Trace theory, path expressions, and CAFE describe each sub-process in terms of global events. Hence parallel composition corresponds to the adding of restrictions on the occurrences of the global events.

This distinction is easily seen by comparing the effects of composing and event a with itself in the two systems. In CCS, $a \mid a$ refers to the composition of two sub-proceses each of which performs a once. So the resultant process is aa. In the other three algebras, composition is idempotent since no extra restrictions are generated. For example, in trace theory, $a \; \underline{w} \; a = a$. We use the term *co-incidence* to refer to the latter form of parallel composition, events in the composed descriptions co-incide in the resultant.

Parallel composition in CCS, *synchronization*, is a one-to-one operation. One event request merges with one event to produce τ. As a result, CCS allows us to control the number of times events can occur, as in the binary semaphore example of section 5.4. This is not possible with co-incidence. Co-incidence can however be simulated in CCS, but at the cost of adding a large number of dummy events. Some path expression languages provide some numerical control by using additional constructs such as *flags* in [2] and *states* in [7].

7.3 Dependency

The chief factor that distinguishes CAFE from the other three algebras is its ability to describe dependencies between events of finite duration. In CCS, path expressions, and trace theory, the only inter-event dependency possible is precedence. For CCS and trace theory, no other relationship is possible between events due to the use of the interleaving semantics. CAFE allows the description of all possible inter-event relationships assuming that each event has a distinct beginning and end.

7.4 Comparison

In fig 7 we present a comparison of the four algebras we have discussed in terms of the main features that distinguish them.

	Type of independence handled	Interleaving semantics for independence?	Type of parallel composition	Inter-event dependencies
Trace Theory	local	yes	co-incidence	precedence
Path Expressions	global	no	co-incidence	precedence
CCS	local	yes	synchronization	precedence
CAFE	local	no	co-incidence	6 relations

Figure 7:

If the events that we are dealing with are of short duration, or do not overlap in time (i.e. no two events are active at any moment), then the interleaving semantics is an adequate approximation of independence. In this case, using either CCS or trace theory has the advantage that analysis of the system is simplified. If events overlap, then the interleaving semantics cannot be used. Path expressions are simple to implement but are useful only if no complex relations exist between events. CAFE allows us to describe in detail relationships between overlapping events. It is our belief that this ability will make it useful in the description and analysis of systems for VLSI implementation.

References

[1] T. S. Anantharaman, E. M. Clarke, M. J. Foster, and B. Mishra. Compiling path expressions into VLSI circuits. *Distributed Computing*, 1(3), 1986.

[2] T. S. Balraj and M. J. Foster. Miss Manners: a specialized silicon compiler for synchronizers. In *Proceedings of the Fourth MIT Conference on Advanced Research in VLSI*, MIT, April 1986.

[3] D. L. Black. *On the Existence of Delay-Insensitive Fair Arbiters : Trace Theory and its Limitations*. Technical Report CMU-CS-85-173, Carnegie-Mellon University, October 1985.

[4] B. Bruegge and P. Hibbard. Generalized path expressions : a high-level debugging mechanism. *The Journal of Systems and Software*, 3:265–276, 1983.

[5] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In G. Goos and J. Hartmanis, editors, *LNCS 16 : International Symposium on Operating Systems, VIII*, pages 89–102, Springer Verlag, 1974.

[6] G. Costa and C. Stirling. A fair calculus of communicating systems. In M. Karpinski, editor, *LNCS 158 : Foundations of Computation Theory, XI*, pages 94–105, Springer-Verlag, 1983.

[7] R. W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *JACM*, 29(3):603–622, July 1982.

[8] M. J. Foster. *Specialized Silicon Compilers for Language Recognition.* PhD thesis, Carnegie-Mellon University, July 1984.

[9] A. N. Habermann. *Implementation of Regular Path Expressions.* Technical Report ANH7902, Carnegie-Mellon University, 1975.

[10] A. N. Habermann. *Path Expressions.* Technical Report ANH7901, Carnegie-Mellon University, 1975.

[11] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[12] R. Janicki. A characterization of concurrency-like relations. In G. Kahn, editor, *LNCS 70 : Semantics of Concurrent Computation, VI*, pages 109–122, Springer-Verlag, 1979.

[13] R. Janicki. A method for developing concurrent systems. In C. Girault and M. Paul, editors, *LNCS 167 : International Symposium on Programming, VI*, pages 155–166, Springer-Verlag, 1984.

[14] E. Knuth. Cycles of partial orders. In J. Winkowski, editor, *LNCS 64 : Mathematical Foundations Of Computer Science, X*, pages 315–325, Springer-Verlag, 1978.

[15] L. Lamport. The mutual exclusion problem : part i - a theory of interprocess communication. *Journal of the Association for Computing Machinery*, 33(2):313–326, April 1986.

[16] L. Lamport. The mutual exclusion problem : part ii - statement and solutions. *Journal of the Association for Computing Machinery*, 33(2):327–348, April 1986.

[17] W. Li and P. E. Lauer. *A VLSI Implementation for COSY.* Technical Report ASM/121, Computing Laboratory, The University of Newcastle Upon Tyne, January 1984.

[18] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *LNCS 255: Advances in Petri Nets, Part II. X*, pages 279–324, Springer-Verlag, 1986.

[19] R. Milner. An algebraic theory for synchronization. In K. Weihrauch, editor, *LNCS 67 : Theoretical Computer Science, VII*, pages 27–35, Springer-Verlag, 1979.

[20] R. Milner. Lectures on a calculus of communicating systems. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *LNCS 197 : Seminar on Concurrency, X*, pages 268–280, Springer-Verlag, 1985.

[21] R. Milner. *LNCS 92 : A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[22] R. Milner. Synthesis of communicating behaviours. In J. Winkowski, editor, *LNCS 64 : Mathematical Foundations of Computer Science, X*, pages 71–83, Springer-Verlag, 1978.

[23] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *LNCS 104 : Theoretical Computer Science, VII*, pages 167–183, Springer-Verlag, 1981.

[24] V. Pratt. The pomset model of parallel processes : unifying the temporal and the spatial. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *LNCS 197 : Seminar on Concurrency, X*, pages 180–196, Springer-Verlag, 1985.

[25] W. Reisig. *Petri Nets*. Springer-Verlag, 1985.

[26] M. W. Shields. Adequate path expressions. In G. Kahn, editor, *LNCS 70 : Semantics of Concurrent Computation, VI*, pages 249–265, Springer-Verlag, 1979.

[27] J. L. A. van de Snepscheut. *LNCS 200 : Trace Theory and VLSI Design*. Springer-Verlag, 1985.