A Comparative Introduction to CSP, CCS and LOTOS

Colin Fidge
Software Verification Research Centre
Department of Computer Science
The University of Queensland
Queensland 4072, Australia

January 1994

Abstract

The language features and formal definitions of the three most popular process algebras, CSP, CCS and LOTOS, are introduced and compared.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages, methodologies*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*algebraic language theory*

Additional Key Words and Phrases: Concurrency, formal specification, formal description techniques, process algebras, Communicating Sequential Processes, CSP, CCS, LOTOS

Contents

1 Introduction

The specification languages collectively known as the process algebras are, at first glance, very similar. They all begin with notions of processes composed from atomic actions, have operational semantics, and include many operators in common. Underlying these similarities, however, are some subtle differences. Some syntactically identical operators have very different semantics. Conversely, some semantic requirements are expressed differently from one language to the next. The purpose of this article is to alert the reader to these and other differences between the algebras and thus allow an informed selection to be made among them for a particular application.

2 Background

The term *process algebra* refers to a family of specification techniques particularly well-suited to describing systems of concurrent, communicating components. More than mere documentation methods, however, they also incorporate equivalence theories that define algebraic laws, i.e., formal reasoning systems with variables representing processes. (Most recently, non-algebraic methods such as modal logic have been used as well, suggesting that a better name may be process *calculus* [Milner 1989, p.4].)

Many such languages, and dialects, currently exist. For the purposes of this study we consider the basic forms of the three most well-known languages:

- Communicating Sequential Processes (CSP) [Hoare 1985],
- the Calculus of Communicating Systems (CCS) [Milner 1989], and
- the Language of Temporal Ordering Specification (LOTOS) [ISO 1989].

No slight is intended on the technical merits of other established process algebras such as the Algebra of Communicating Processes (ACP) [Bergstra & Klop 1984], the RAISE Specification Language (RSL) [RAISE 1992] or the π-caclulus [Milner 1991]. Our restriction to the three languages above is based purely on the need to keep the study manageable and on a perception that newcomers to the field will be more likely to have heard of these three. The categories used herein for comparing the languages should be applicable to any process algebra and readers familiar with other methods are encouraged to see how their language fits in.

2.1 History

Figure 1 summarises major events in the evolution of the languages of interest here.

Hoare's CSP first came to prominence in 1978. At that stage it was not a fully-fledged process algebra, but a programming language fragment inspired by Dijkstra's guarded command language and Pascal. Due to the revolutionary emphasis placed on a minimal form of interprocess communication it became highly influential as a language for reasoning about concurrent and distributed systems. In the following years the concepts were refined and the language became more oriented towards specification, rather than implementation, culminating in its definitive process-algebraic form published in 1985 [Hoare 1985].

Milner's CCS has had a less obvious evolution, with most changes occurring in the underlying semantics. An initial version was published in 1980 but improvements to the semantic model used for defining equivalence continued to be made, inspired by the work of David Park, until the definitive form appeared in 1989 [Milner 1989]. (Milner now prefers the term "process calculus", but we will continue to use "CCS" due to its familiarity.) CSP and CCS influenced one another throughout their development.

Both of the developments above were academic projects. LOTOS, on the other hand, was developed in response to a practical need. The telecommunications industry has long used precise notations, with varying degrees of formality, for describing protocols. The International Standardisation Organisation (ISO) recognised a need to develop a new Formal Description Technique (FDT) in the late 1970's [Logrippo et al. 1992]. The name "LOTOS" was coined in 1983 to describe a language based on early versions of CCS and CSP. It was subsequently refined and became an international standard in 1989 [ISO 1989].

In comparing the languages it is important to understand the different motivations that underly them. CSP strives to have a simple semantic model in which it is easy to define new operators as they are needed [Hoare 1985, p.247]. CCS was designed to have a minimal set of operators with communication as the central concept [Milner 1989, p.1–2]. Both languages are intended to be flexible and to encourage experimentation. LOTOS was created as a standardised FDT for open distributed systems. Unlike CSP and CCS it did not aim to be extensible [Bolognesi & Brinksma 1987, p.25].

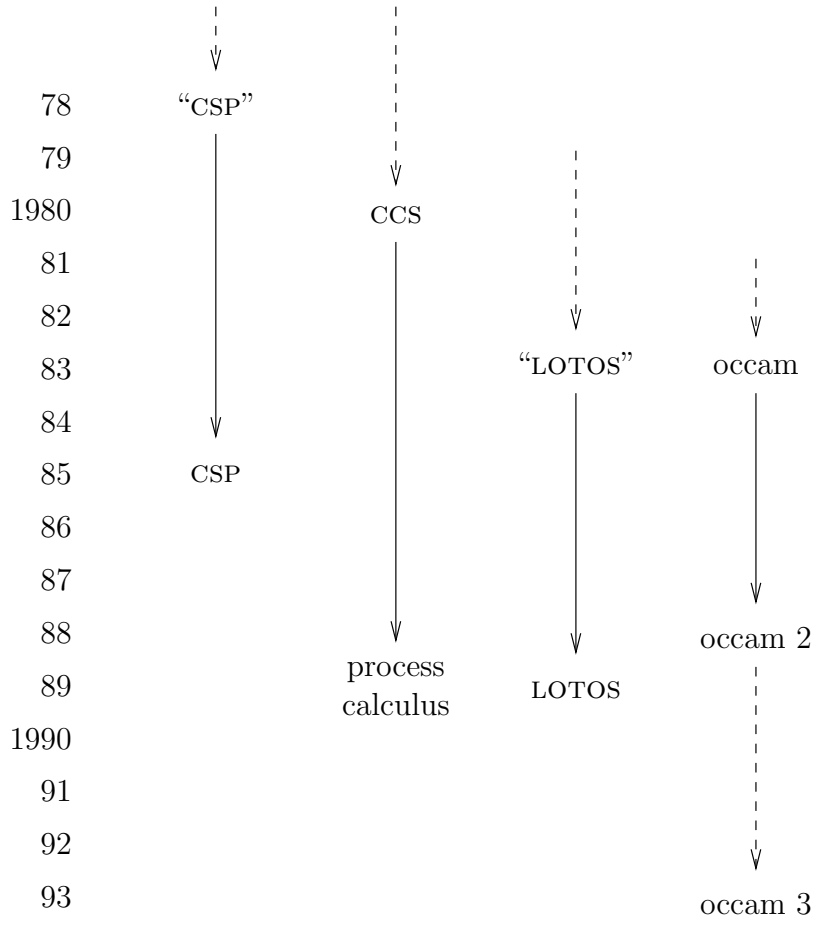

Figure 1: History.

The programming language occam also features in figure 1. Although not a process algebra *per se* it is significant because it is closely based on (early versions of) CSP. It embodies many of the concepts found in CSP and thus potentially benefits from the theory surrounding CSP [Roscoe & Hoare 1986].

2.2 Applicability

To date the process algebras have proven valuable in the specification and design of distributed systems [Logrippo et al. 1992, §6, p.341], for formal reasoning [Milner 1989], and for rapid prototyping [Loureiro et al. 1992].

There are, of course, many rivals to the process algebras. Below we indicate some of the advantages of process algebras in their particular domain.

- Specification languages such as Z and VDM allow specifications to be expressed non-constructively, often at a much higher level of abstraction than is possible with the process algebras. However they have no notations for expressing

concurrency or communication. In those situations where a problem is naturally, and unavoidably, described as a set of communicating processes (e.g., in a geographically-separated network) the process algebras provide a much more direct model.

- Petri nets can model concurrent behaviour. Moreover they have true concurrency semantics and can model causality, concepts lacking in the process algebras. Nevertheless there is no satisfactory algebraic theory for them, and they offer a more primitive notation than the process algebras [Milner 1989, pp.3-4].

- Regular expressions and finite-state automata are simple and familiar concepts; at first glance process algebras have much in common with them. However equivalence models for the process algebras include extensive consideration of nondeterministic choices [Milner 1989, §2.3, p.42, §4.1], a feature not normally found in, e.g., regular expressions.

3 The languages

This section describes the process algebras from their user's viewpoint. Section 4 describes the underlying formal models.

Language features have been grouped by function. To assist with a direct comparison of the languages a tabular format has been adopted; semantically equivalent concepts from each language are grouped together in the same table. Different concepts appear in separate tables (even when they are syntactically identical).

3.1 Terminology

All of the process algebras have the same fundamental basis. They start from a set of atomic actions from which processes may be constructed. Processes may, in turn, be composed using special operators to create more complex behaviours. The operators themselves obey algebraic laws which can be used for formal reasoning.

Despite these shared beginnings, an immediate source of confusion is the inconsistent use of terminology among the algebras. Table 1 summarises some of the more important synonyms.

All of the languages begin with a notion of atomic actions. These are known as "events" in CSP, "actions" in CCS, and "actions" or "gates" in LOTOS. (Strictly, actions and gates are equivalent in the basic form of LOTOS only but we will consistently use the latter for distinctness from CCS.) Nothing is known of these actions other than their identity. Their behaviour cannot be examined and they cannot be broken up into smaller actions. No assumptions are made about their granularity. They are chosen freely by the specifier to embody significant indivisible actions that

CSP	CCS	LOTOS
events $(e, e_1, \ldots)$	actions $(a, a_1, \ldots)$	actions or gates $(g, g_1, \ldots)$
processes $(P, P_1, \ldots)$	agents $(A, A_1, \ldots)$	behaviour expressions $(B, B_1, \ldots)$
[Hoare 1985, chs.1-3]	basic calculus	basic LOTOS
[Hoare 1985, chs.4-6]	value-passing calculus	full LOTOS
alphabet (αP)	sort $(\mathcal{L}(A))$	"events" or "actions"
channels $(c, c_1, \ldots)$	ports $(p, p_1, \ldots)$	gates $(g, g_1, \ldots)$
symbols	labels	"gates"
"domains"	value sets	sorts $(t, t_1, \ldots)$

Table 1: Terminology.

the system is expected to perform. The full set of actions that a system may perform is called the "alphabet" in CSP or the "sort" in CCS(to add to our confusion, LOTOS uses the word "sort" to denote data types).

These actions are then used as the basis for creating more complex, structured behaviours defined as CSP "processes", CCS "agents" or LOTOS "behaviour expressions". (We ignore a minor distinction between "agents" and "agent expressions" in CCS [Milner 1989, §2.4, pp.43–44].) All agents exist in the context of their "environment". The environment, which is itself an agent, shares in the performance of each visible action.

All three languages have a simple form, lacking data values, and then build the complete language on this. Both CCS and LOTOS call their fundamental language the "basic" form. This language is not given a specific name in CSP, but is embodied in chapters 1 to 3 of the CSP text. The "full" language in LOTOS is equivalent to the "value-passing" calculus in CCS and chapters 4 to 6 of the CSP book.

The borderline between these basic and full algebras is unclear, however. Several basic LOTOS constructs are not "basic" in CSP or CCS. Also, CSP and CCS are extensible and many operators are introduced in their textbooks merely to illustrate particular examples; it is unclear which operators should be treated as canonical.

Another prominent feature of the full algebras is the ability of interactions between agents to carry data values. Such interactions take place through "channels" in CSP, "ports" in CCS, and "gates" in LOTOS.

In this presentation we use the terms defined for the particular language when discussing specific language features. In general discussion we favour CCS terminol-

CSP: prefix [Hoare 1985, §1.1.1]	$e \rightarrow P$
CCS: prefix [Milner 1989, §1.3, p.27]	$a.A$
LOTOS: action prefix [Bolognesi & Brinksma 1987, §2.2]	g; B

Table 2: Prefixing actions.

ogy because it avoids overloaded or unnecessarily evocative words. For instance, it uses "agent" instead of "process"; the word "process" suggests concurrency even though the concept has other interpretations, as explained in section 3.5.

3.2 Basic concepts

In this section we introduce the language features found in the basic process algebras.

3.2.1 Prefixing

All of the algebras have a prefixing operator that is used to construct agents from atomic actions. This is the only way in which a new action can be introduced.

The CSP expression in table 2, for instance, defines a process that first performs some event named e and subsequently behaves like process P. Similarly, the CCS agent performs action a and then behaves like agent A, and the LOTOS behaviour expression participates in gate action g and then behaves like behaviour expression B. Despite the differences in syntax, all three operators express the same concept.

The actions in table 2 can proceed only in environments also willing to participate in the named action, otherwise the agents are blocked. Also see sections 3.2.4 and 3.2.5.

3.2.2 Choice

The operators available for expressing choices can be grouped according to the way they handle determinism.

Deterministic. The most fundamental form of choice operator allows the environment to choose between distinct events. Only CSP has a special operator to express this concept.

CSP: choice [Hoare 1985, §1.1.3]	$(e_1 \rightarrow P_1 \mid e_2 \rightarrow P_2)$

Table 3: Deterministic choice.

CSP: nondeterministic or [Hoare 1985, §3.2]	$P_1 \sqcap P_2$

Table 4: Nondeterministic choice.

In the process in table 3 the two initial events e_1 and e_2 must be uniquely identifiable by the environment of the process. Which of the two is actually chosen is under the control of the environment. (There is, however, no concept of causality [Hoare 1985, p.24]. We cannot say whether the process or its environment "caused" the event.)

Although this operator is not available in CCS or LOTOS the same effect can be achieved by using the general choice operator (see below) with distinct initial actions. For example, in CCS it can be expressed as $(\mathtt{a}.A_1 + \mathtt{b}.A_2)$ because `a` and `b` are distinct actions. The "choice" operator in CSP is therefore not fundamental. Non-essential operators are often introduced in CSP to aid clarity; CCS studiously avoids any unnecessary operators.

Nondeterministic. A purely nondeterministic choice behaves like one of its arguments, selected at random. Again, only CSP has a distinct operator for expressing this concept (table 4).

Such a choice cannot be influenced, or even observed, by the environment. It is impossible to tell when the choice is made; the decision may have been fixed in advance, or made only when the process is executed. There is no fairness assumption; in table 4 P_1 may *always* be chosen, no matter how many times the process is executed.

This form of choice is sometimes called "demonic" because it represents the least controllable form of decision. It is included in CSP mainly to give an explicit model of a situation we want to avoid! It does have a legitimate use, however, when specifying two alternative, equally valid, implementations of a particular behaviour.

The concept can be indirectly modelled in CCS or LOTOS by making a general choice between internal actions (see below). For example, in CCS a random choice between two agents can be expressed as $(\tau.A_1 + \tau.A_2)$ [Hoare 1985, §7.4.1]. Again, this operator is not fundamental.

CSP: general choice [Hoare 1985, §3.3]	$P_1 \,\Box\, P_2$
CCS: summation [Milner 1989, §1.3, p.29]	$A_1 + A_2$
LOTOS: choice [Bolognesi & Brinksma 1987, §2.2]	$B_1 \,[]\, B_2$

Table 5: General choice.

CSP: choice of [Hoare 1985, §1.1.3, p.32]	$(x{:}\, B \to P(x))$
LOTOS: generalised choice [Bolognesi & Brinksma 1987, §5.3]	**choice** x **in** $[g_1, \ldots, g_n]$ $\;[]\; N[x](\ldots)$

Table 6: Choice from a set of actions.

General. The generalised choice operator allows the environment to select between distinct alternatives, but if there is more than one available instance of the chosen action the choice becomes nondeterministic.

For instance, in table 5, if LOTOS behaviour expressions B_1 and B_2 offer disjoint sets of gates as their possible initial events then the environment can choose between the two behaviour expressions. However if there is a gate g which is a possible initial event of both B_1 and B_2 then it is impossible to predict which of these two behaviours will be performed if g is selected.

Indexed. When we want to specify a choice among a large number of possible initial actions it is often inconvenient to enumerate them all. Choices may therefore be expressed over indexing sets.

In CSP, a *deterministic* choice can be made of an event from set B, and the subsequent behaviour is determined by the chosen event. (In table 6 P is a process-valued function.)

CCS provides a more general form of indexed choice via its summation operator. In table 7 I is an indexing set whose values (which need not be limited to actions) can be used to distinguish instances of a parameterised agent A. Infinite summations are allowed when the size of the indexing set I is not bounded [Milner 1989, §2.4, p.44].

LOTOS offers two distinct forms of indexed choice. In the first case (table 6)

CCS: summation [Milner 1989, §2.4]	$\sum_{x \in I} A_x$

Table 7: Indexed general choice.

LOTOS: generalised choice [Bolognesi & Brinksma 1987, §5.3]	**choice** $x{:}t\ [\,]\ B(x)$

Table 8: Choice based on data value.

a gate x is selected from a list of possible gates $g_1, \ldots, g_n$ and used to choose the subsequent behaviour of a parameterised behaviour expression named N (see section 3.2.8). In the second form (table 8) the index consists of a data value x of type t. Note that **choice** is a full, not basic, LOTOS operator.

For convenience other operators in the algebras may be similarly indexed, e.g., nondeterminism in CSP [Hoare 1985, §3.2.1, p.104] and composition in CCS [Milner 1989, §5.5, p.118]. We will not describe these shorthands any further here.

3.2.3 Composition

In this section we describe the most prominent feature of the process algebras, their support for composition of agents. Although these operators are usually thought of as defining "concurrency", we use the CCS terminology instead to emphasise the fact that these operators may used for other purposes (see section 3.5).

Independent. In the simplest case the composed agents are entirely independent. They do not interact with each other, even when performing the same action. For instance, in table 9, CSP processes P_1 and P_2 perform their events separately. If they can both perform the same event e then two separate such events take place.

CSP: interleaving [Hoare 1985, §3.6]	$P_1 \mid\mid\mid P_2$
LOTOS: pure interleaving [Bolognesi & Brinksma 1987, §2.4, p.34]	$B_1 \mid\mid\mid B_2$

Table 9: Composition of independent agents.

CSP: interaction [Hoare 1985, §2.2]	$P_1 \mid\mid P_2$
LOTOS: full synchronisation [Bolognesi & Brinksma 1987, §2.4, p.34]	$B_1 \mid\mid B_2$

Table 10: Fully dependent agent composition.

"Interleaving" is an unfortunate misnomer; it reflects the operational semantics usually used to define the operators, not their purpose. These operators are intended to model concurrent processes that do not communicate.

In CCS, composed agents may interact whenever one agent is prepared to perform some action a and the other is prepared to perform a complementary action, denoted $\overline{a}$. Therefore the effect of independent composition can be achieved in CCS by using the general form of agent composition, i.e., $(A_1 \mid A_2)$ (see below), as long as the sorts of the two agents have no "co-names" in common, i.e., $\mathcal{L}(A_1) \cap \overline{\mathcal{L}}(A_2) = \{\}$ (see section 3.2.4).

Fully dependent. At the other extreme, agents composed with full dependency must interact on the occurrence of every action (table 10). Actions can be successfully performed only if they simultaneously occur in all argument agents.

The CSP operator in table 10 is, in fact, the same as the general CSP concurrency operator described below. When intended to denote fully dependent composition however we are obliged to ensure that the argument processes share the same set of possible events, i.e., their alphabets must be identical, $\alpha P_1 = \alpha P_2$ (see section 3.2.4).

The effect of these CSP and LOTOS operators is difficult to achieve in CCS due to the way CCS conceals interactions (see below). For instance, when LOTOS behaviour expressions B_1 and B_2 in table 10 participate in some shared gate action g, the interaction is visible to the environment. Thus several behaviour expressions can participate in an occurrence of g. In CCS, however, interactions between two agents are immediately hidden from the environment. (Although LOTOS is primarily based on CCS, it takes its communication model from CSP.)

General case. The general form of composition allows agents to interact on only a subset of their actions. Other actions are performed independently. The way in which the subset of interacting actions is specified is different in all three algebras.

In CSP, two processes P_1 and P_2 must synchronise on any events that both are potentially capable of performing, i.e., those in the intersection of their two alphabets, $\alpha P_1 \cap \alpha P_2$. If, for instance, P_1 in table 11 is prepared to perform some event e that P_2 is *capable* of performing, but not yet ready to undertake, then e cannot occur. Events that appear in the alphabet of only one process can be

CSP: concurrency [Hoare 1985, §2.3]	$P_1 \mid\mid P_2$

Table 11: General composition: shared actions interact.

CCS: composition [Milner 1989, §2.5, p.46]	$A_1 \mid A_2$

Table 12: General composition: complementary actions interact.

performed by that process alone. Whenever any event e is performed by such a composition of processes (whether shared or not) it is also visible to, and requires the participation of, the environment.

In CCS, agents may interact on two complementary actions a and $\overline{a}$. Only two agents may participate in each interaction. If interaction takes place it is hidden (see section 3.2.5) and cannot be seen by the environment. However, the agents may nondeterministically choose not to interact, even when interaction is possible, and perform a and $\overline{a}$ separately (table 12).

This model is difficult to replicate in CSP or LOTOS because they have no concept of such optional interaction. On the other hand, the CSP approach is not available in CCS, but can be defined [Milner 1989, §9.2, p.194].

The LOTOS operator (table 13) is similar to that of CSP except that an explicit list of gates $g_1, \ldots, g_n$ on which interaction must take place is given. These events may occur only when both B_1 and B_2 can perform them. All other gates may be performed by the argument behaviour expressions separately, even when both B_1 and B_2 are capable of performing the same gate event.

Note that "parallelism" is a misnomer; none of the process algebras support "true" concurrency, only interleaving.

3.2.4 Interaction

The previous section described the different forms of agent composition available and their influence on interaction. This section summarises the main differences.

LOTOS: general parallelism [Bolognesi & Brinksma 1987, §2.4]	$B_1 \mid[g_1, \ldots, g_n]\mid B_2$

Table 13: General composition: listed actions interact.

Interaction points:

- CSP and LOTOS processes can synchronise on identically named actions [Hoare 1985, §2.3.1] [Bolognesi & Brinksma 1987, §2.4, p.33].
- CCS agents synchronise only on actions with complementary names, e.g., `a` and $\overline{\texttt{a}}$ [Milner 1989, §2.2, p.39].

Ways of specifying interaction points:

- In CSP and CCS the actions on which processes may interact are implicit in the process alphabets [Hoare 1985, §2.3].
- In LOTOS the gates on which behaviour expressions may interact are explicitly listed [Bolognesi & Brinksma 1987, §2.4].
- In all three cases interaction takes place through named "channels". Process names are not used [Logrippo et al. 1992, p.328].

Number of participants:

- CSP and LOTOS support multi-party synchronisation. Any number of processes may interact on a single, shared action [Hoare 1985, §2.3.1] [Bolognesi & Brinksma 1987, §2.4, p.33].
- CCS supports bi-party interaction only [Milner 1989, §2.2, p.39].

Visibility of interactions:

- In CSP and LOTOS each interaction is visible to the environment [Hoare 1985, §2.3.1] [Bolognesi & Brinksma 1987, §2.4, p.33]. (This is necessary to support multi-way interaction.)
- In CCS interactions are always concealed from the environment [Milner 1989, §2.2, p.39]. If two agents interact this shared action can never be seen by any other agents. (Multi-way interactions are thus awkward to express in CCS.)

3.2.5 Concealment

Concealment operators are intended to prevent the environment from participating in, or observing, actions. Two different models are used.

The concealment operators in CSP and LOTOS allow the user to explicitly list those events that may occur invisibly, without the participation of the environment. If P in table 14 is ready to perform some event e in the list of concealed events then it may do so immediately, without the need to interact any other process. Thus,

CSP: concealment [Hoare 1985, §3.5]	$P\backslash\{e_1, \ldots, e_n\}$
LOTOS: hiding [Bolognesi & Brinksma 1987, §2.5]	**hide** $g_1, \ldots, g_n$ **in** B

Table 14: Concealing actions.

CCS: restriction [Milner 1989, p.40]	$A\backslash\{a_1, \ldots, a_n\}$

Table 15: Restricting possible actions.

in CSP the concealment operator is used to *prevent* interaction between process P and its environment.

The corresponding operator in CCS is used to restrict the actions that may occur. If A in table 15 is ready to perform some restricted action a then that action cannot occur at all (the complementary action $\overline{a}$ is also restricted).

For example, if A was prepared to perform an action $\mathtt{a}$ and the environment is ready to perform the complementary action $\overline{\mathtt{a}}$ then two things may normally occur, either

1. the two agents interact, without the knowledge of the environment, or
2. they visibly perform $\mathtt{a}$ and $\overline{\mathtt{a}}$ independently.

If, however, $\mathtt{a}$ (or $\overline{\mathtt{a}}$) is restricted, then that action cannot visibly occur at all; only the interaction between the two agents can take place. Thus, although syntactically identical to the CSP concealment operator, the CCS restriction operator is instead used to *force* interaction between agent A and its environment.

3.2.6 Internal action

We have already seen that internal actions, i.e., actions that occur without the knowledge of the environment, can result from

- agent interaction in CCS [Milner 1989, p.40], and
- explicit concealment in CSP and LOTOS [Hoare 1985, §3.5, p.112] [Bolognesi & Brinksma 1987, §2.5].

CCS and LOTOS have special notations for such anonymous actions and these may be explicitly used in prefixes to denote some "local" behaviour (table 16).

CCS: silent action [Milner 1989, p.39]	$\tau.A$
LOTOS: unobservable action prefix [Bolognesi & Brinksma 1987, §2.2]	$\mathbf{i}; B$

Table 16: Internal actions.

CSP: change of symbol [Hoare 1985, §2.6]	$f(P)$
CCS: relabelling [Milner 1989, p.32]	$A[f]$

Table 17: Renaming of actions.

Special actions τ and $\mathbf{i}$ are never seen by the environment and cannot be used as interaction points (even when appearing in an argument to the LOTOS full synchronisation operator).

3.2.7 Renaming

Having specified an agent that defines some useful behaviour we often want to reuse that specification but replace one or more of the actions with different action names. In table 17 f is a one-to-one function on action names. Wherever CCS agent A can perform some action a, agent $A[f]$ can perform action $f(a)$.

For convenience CSP also offers a notation for adding a prefix to all events performed by a process. Wherever process P in table 18 could perform some event e, process $l{:}\,P$ can perform an event with compound name $l.e$.

In LOTOS the effect of instantiating a given behaviour expression with different gate names is achieved by using parameterised process definitions (see section 3.2.8).

CSP: process labelling [Hoare 1985, §2.6.2]	$l{:}\,P$

Table 18: Systematically renaming actions.

CSP: equations [Hoare 1985, §1.1.2]	$N = P$
CCS: defining equations [Milner 1989, §2.4, p.44]	$N \stackrel{\text{def}}{=} A$
LOTOS: process definitions [Bolognesi & Brinksma 1987, §2.1]	**process** $N\ [g_1, \ldots, g_n]\ :=$ B **endproc**

Table 19: Agent definitions.

CSP: recursive expressions [Hoare 1985, §1.1.2]	$\mu X.P$
CCS: recursive expressions [Milner 1989, §2.9]	$\mathbf{fix}(X = A)$

Table 20: Recursive expressions.

3.2.8 Definitions

All of the algebras have a way of associating a name N with a specified behaviour (table 19). These definitions may be recursive.

The basic LOTOS method of defining a "process" N from a behaviour expression B optionally allows gates as parameters. CSP processes can be similarly parameterised, as already seen in section 3.2.2, via process-valued expressions $P(e)$, with parameter e consisting of an event name [Hoare 1985, p.32]. All three full algebras allow agents to be parameterised with data values (see section 3.3.1).

3.2.9 Recursive expressions

As well as allowing recursive definitions, CSP and CCS allow recursive agents to be formally defined as fixpoint expressions. For instance, in table 20, X may appear where a process is expected in CSP process P. To be meaningful such recursion must be properly guarded [Hoare 1985, p.28] [Milner 1989, p.65].

Note. At this point we have covered all of the basic CCS operators. The CCS operators in sections 3.2.10 through to 3.2.13 can be defined in terms of the operators above.

CSP: stop [Hoare 1985, p.25]	*STOP*
CCS: inactive agent [Milner 1989, §2.4]	**0**
LOTOS: inaction [Bolognesi & Brinksma 1987, §2.2]	**stop**

Table 21: Inaction.

CSP: successful termination [Hoare 1985, §5.1]	*SKIP*
LOTOS: successful termination [Bolognesi & Brinksma 1987, p.36]	**exit**

Table 22: Successful termination.

3.2.10 Inaction

All algebras have a notation for an agent that cannot do anything (table 21). This is often used to denote an agent that is deadlocked or in some other way "broken". An inactive agent never terminates.

Although considered to be fundamental in CSP and LOTOS, the inactive agent is not in CCS. It is defined as a choice from an empty set of alternatives, i.e.,

$$\mathbf{0} \stackrel{\text{def}}{=} \sum_{i \in \{\}} A_i \ .$$

3.2.11 Successful termination

The inaction operators indicate a failure to terminate. By constrast the operators in table 22 terminate immediately and denote successful completion of some task. An agent consisting of several composed agents can itself successfully terminate only when all of its component agents have successfully terminated.

The concept of successful termination is not normally found in CCS but can be modelled [Milner 1989, pp.172–173].

LOTOS allows its successful termination operator to be used as an alternative in a choice. For instance, $(\mathbf{exit}\,[\,]\,B)$ is a behaviour expression that can choose to behave like B or successfully terminate [Logrippo et al. 1992, p.335]. CSP, however, insists on "well-termination"; successful termination may not be offerred as an alternative and $(SKIP \,\Box\, P)$ is therefore invalid [Hoare 1985, §5.1, p.171] [Milner 1989, p.173].

CSP: sequential composition [Hoare 1985, §5.1]	$P_1;P_2$
LOTOS: sequential composition [Bolognesi & Brinksma 1987, p.36]	$B_1 \gg B_2$

Table 23: Sequential composition.

Note. At this point we have also covered all of the basic CSP operators.

3.2.12 Sequential composition

The ability of an agent to successfully terminate makes it possible to sequentially compose agents (table 23). The second agent begins execution only when, and if, the first terminates.

The CSP operator in table 23 and the LOTOS prefixing operator (table 2) should not be confused; prefixing introduces new actions, sequential composition orders entire agents.

Again this concept is not normally found in CCS but can be modelled as a special case of general composition [Milner 1989, p.173].

3.2.13 Pre-emption

In some applications we often wish to model the situation where an agent is prevented from completing its specified behaviour due to some external influence. The operators in table 24 represent a special case of sequential composition in which the second agent may begin before the first terminates. For instance, if an initial event e of process P_2 becomes possible while P_1 is executing, the environment can select e and the system subsequently continues to behave like P_2. Execution of P_1 is never continued; "interrupt" is thus a misleading name.

LOTOS behaviour expression $B_1 [> B_2$ terminates, without performing B_2, if B_1 successfully terminates. This is not possible in CSP because the first argument to the interrupt operator is not allowed to be one which may successfully terminate [Hoare 1985, p.180].

Again the concept can be defined in CCS [Milner 1989, §9.2, p.192].

Note. At this point we have also completed the basic LOTOS operators.

3.3 Value-passing

So far the algebras have consisted of atomic actions and agents composed of actions. Data values have been conspicuously absent. They appear only in the "full" algebras

CSP: interrupt [Hoare 1985, §5.4]	$P_1 \wedge P_2$
LOTOS: disabling [Bolognesi & Brinksma 1987, §2.8]	$B_1 [> B_2$

Table 24: Pre-emption of agents.

CSP: equations [Hoare 1985]	$N(v_1, \ldots, v_n) = P$
CCS: parameterised constant [Milner 1989, §2.8]	$N(v_1, \ldots, v_n) \stackrel{\text{def}}{=} A$
LOTOS: parametric processes [Bolognesi & Brinksma 1987, §5.4]	**process** N [...] $(v_1{:}\, t_1, \ldots, v_n{:}\, t_n){:}\, f$:= B **endproc**

Table 25: Value-parameterised agent definitions.

because, as explained in section 4.6, the algebras with value-passing can be defined entirely in terms of the basic algebras above.

3.3.1 Value-parameterised agent definitions

All three algebras allow the definition of an agent named N to accept data values $v_1, \ldots, v_n$ as parameters (table 25). Data types are treated informally in CSP and CCS, typically being restricted to simple types such as integers or characters. Types are defined more rigorously in LOTOS using the ISO standardised data description language ACT ONE [Bolognesi & Brinksma 1987, §4].

(The CSP textbook [Hoare 1985] is inconsistent in its use of this mechanism and frequently uses the subscripted form $N_{v_1,\ldots,v_n}$ where $N(v_1, \ldots, v_n)$ is appropriate.)

The LOTOS parametric process definition includes a functionality parameter f. This may take one of three forms:

- **noexit** indicates that the process never successfully terminates,
- **exit** indicates that the process is capable of successfully terminating, and
- **exit**$(t_1, \ldots, t_x)$ indicates that the process is capable of terminating and "offering" data values of type $t_1, \ldots, t_x$ (also see section 3.3.3).

CSP: – [Hoare 1985]	$N(e_1, \ldots, e_n)$
CCS: – [Milner 1989, pp.17–18]	$N(e_1, \ldots, e_n)$
LOTOS: process instantiation [Bolognesi & Brinksma 1987, §5.4]	$N[\ldots](e_1, \ldots, e_n)$

Table 26: Instantiating value-parameterised agents.

LOTOS: let [Bolognesi & Brinksma 1987, §5.4]	**let** $x_1{:}\, t_1 = e_1, \ldots, x_n{:}\, t_n = e_n$ **in** $N(x_1, \ldots, x_n)$

Table 27: Let.

3.3.2 Instantiating value-parameterised agents

Having declared an agent N, parameterised by data values, all three algebras allow familar notations for instantiating N with the values of expressions $e_1, \ldots, e_n$ as arguments (table 26).

LOTOS also has a more verbose alternative syntax (table 27).

3.3.3 Value-passing during sequential composition

In section 3.2.12 we saw how agents can be sequentially composed. Data values, however, have a lifetime equal to that of the agent in which they appear and hence are normally lost when the first agent successfully terminates. To overcome this LOTOS allows values to be explicitly passed from a behaviour expression B_1 to its sequential successor B_2.

To achieve this B_1 must end with a form of **exit** expression parameterised by one or more data expressions (table 28).

Behaviour expressions B_1 and B_2 are then composed using a special form of sequential composition operator with the number and types of parameters matching

LOTOS: value offers [Bolognesi & Brinksma 1987, §5.5.1]	**exit**$(e_1, \ldots, e_n)$

Table 28: Value offers at termination.

LOTOS: accepting values [Bolognesi & Brinksma 1987, §5.5.2]	$B_1 \gg \textbf{accept}\ x_1{:}\,t_1, \ldots, x_n{:}\,t_n\ \textbf{in}\ B_2$

Table 29: Accepting offered values.

CSP: channel output [Hoare 1985, §4.2]	$c!e \rightarrow P$
CCS: port output [Milner 1989, p.17]	$\overline{p}(e).A$
LOTOS: output gate [Bolognesi & Brinksma 1987, §5.1.3]	$g!e\textbf{;}\ B$

Table 30: Output actions.

the **exit** in B_1 (table 29). Values $x_1, \ldots, x_n$ can then be used in B_2.

Similar operators can be defined in CCS [Milner 1989, §8.2, p.174].

3.3.4 Output

The basic algebras allowed agents to synchronise by interacting on shared actions. In the full algebras data values can also be transferred during interaction. The value of an expression e can be sent as shown in table 30.

3.3.5 Input

A data value can be received during an interaction, into variable v, as shown in table 31. In each case the (optional) argument to the agents following the input represents the ability of input to change the local state.

CSP [Hoare 1985, §4.2, p.134] and CCS [Milner 1989, p.15] assume that value-passing interactions take place between only two agents. LOTOS allows any number of behaviour expressions to participate in a value-passing interaction [Logrippo et al. 1992, §3.2], as long as all participants agree on the type of the data value. All the participating input gate actions will receive the value sent. All the participating output gate actions must be attempting to send the same value, otherwise interaction cannot take place.

Thus, where CSP and CCS allow a single output to interact with a single input only, LOTOS allows three forms of communication.

CSP: channel input [Hoare 1985, §4.2]	$c?v \rightarrow P(v)$
CCS: input port [Milner 1989, p.17]	$p(v).A(v)$
LOTOS: input gate [Bolognesi & Brinksma 1987, §5.1.3]	$g?v{:}\,t\mathbf{;}\ B(v)$

Table 31: Input actions.

LOTOS: actions [Logrippo et al. 1992, §3.1]	$g\ c_1 \ldots c_n\mathbf{;}\ B$

Table 32: Multi-value message-passing.

- Value passing: an output expression $!e$ (of type t) is matched with an input variable $?v{:}\,t$. Variable v is assigned the value of expression e.
- Value matching: one output expression $!e_1$ matches with another $!e_2$. Both expressions must return the same value (and be of the same type), otherwise interaction is not possible.
- Value generation: an input variable $?v_1{:}\,t$ matches with another input variable $?v_2{:}\,t$. In this case both v_1 and v_2 will receive the same value. If there is no corresponding output event then the value is created nondeterministically.

3.3.6 Multi-value message-passing

In a full LOTOS action a gate name g may be followed by a list $c_1 \ldots c_n$ of offer ($!e$) and accept ($?v{:}\,t$) events. Thus, when matched with one or more other gate events, with lists of the right length and type, data values can be transferred in two or more directions during the one atomic action (table 32)

CSP and CCS allow only one data value to be transferred per interaction.

3.3.7 Conditional value-passing

In LOTOS a predicate p can be used to restrict the range of values that may be sent or received during communication (table 33).

Input variables may appear in the predicate. In other words, the predicate may examine the values of incoming data in order to decide whether to actually receive

LOTOS: selection predicate [Logrippo et al. 1992, §5.2.1]	$g \ldots [p]; B$

Table 33: Conditional value-passing.

it or not. This capability is not available in CSP or CCS.

3.3.8 Communication

Sections 3.3.4 to 3.3.7 described the operators available for value-passing communication in the algebras. Here we summarise the main features.

Message-passing:

- Value-passing is synchronous in all three process algebras, i.e., a receiver blocks until a compatible agent is ready to send. There are no message buffers linking communicating agents.
- Asynchronous, bufferred communication can be modelled by introducing explicit buffer agents between the communicating entities [Hoare 1985, p.138] [Milner 1989, §1.2].

Number of participants:

- CSP and CCS allow bi-party communication only [Hoare 1985, §4.2, p.134] [Milner 1989, p.15].
- LOTOS allows multi-party communication [Logrippo et al. 1992, §3.2].

Direction of data transfer:

- Value-passing is uni-directional in CSP and CCS [Hoare 1985, §4.2, p.134] [Milner 1989, §1.1].
- Value-passing may be multi-directional in LOTOS [Logrippo et al. 1992, §3.1] [Bolognesi & Brinksma 1987, §5.1.3].

Conditional communication:

- Communication may be conditional in LOTOS [Logrippo et al. 1992, §3.1] [Bolognesi & Brinksma 1987, §5.2.1].
- Communication is unconditional in CSP and CCS. (The 1978 CSP language included a conditional form of "input guard", as does the occam programming language, but this was removed from the 1985 version.)

CCS: condition [Milner 1989, §2.8, p.55]	**if** b **then** A
LOTOS: guarded expressions [Bolognesi & Brinksma 1987, §5.2.2]	$[b] \rightarrow B$

Table 34: Guarded commands.

LOTOS: process definition [Bolognesi & Brinksma 1987, §5, pp.47-48]	**process** ... := ... **where** ... **endproc**

Table 35: Structured process definition.

3.3.9 Guarded commands

Dijkstra's influential guarded command notation is approximated in the process algebras as shown in table 34. It allows an agent to occur only if a boolean expression b is true. In CCS it takes the form of a "one-armed" conditional statement (which is equivalent to inaction if the expression is false).

Guarded commands are not directly available in the 1985 definition of CSP, although the effect can be achieved indirectly using a conditional expression, e.g., $P \nleft b \nright STOP$ [Hoare 1985, §5.5, p.186]. (As noted above, the 1978 version of CSP included a guarded command notation, as does occam.)

Note. At this point we have covered all of the value-passing calculus operators of CCS.

3.3.10 Modularisation

A significant weakness of the process algebras is their failure to provide support for large-scale specifications. Only LOTOS offers any features for this and even this support is minimal.

The constructs in tables 35 and 36 allow us to group "processes" (i.e., named behaviour expressions) together in a hierarchical structure.

LOTOS: specification [Bolognesi & Brinksma 1987, §5, pp.47-48]	**specification** ... **behaviour** ... **where** ... **endspec**

Table 36: Structured system specification.

CSP: assignment [Hoare 1985, §5.5, p.185]	$v := e$
CCS: assignment [Milner 1989, ch.8, p.174,177]	v `:=` e

Table 37: Assignment.

CSP: if then [Hoare 1985, §5.5, p.186,188]	$P_1 \not< b \not> P_2$
CCS: condition [Milner 1989, §2.8, p.55]	**if** b **then** A_1 **else** A_2

Table 38: Conditional statements.

Note. At this point we have covered all of LOTOS.

3.4 Other derived operators

As extensible languages, both CSP and CCS offer the ability to define new operators. Here we list some of the more prominent ones appearing in the literature.

3.4.1 Programming language constructs

As demonstrations of their expressive power, CSP and CCS have been used to model conventional programming language constructs (tables 37, 38 and 39). However, it should be noted that these operators are not essential, and many authorities would argue that they have no place in abstract specifications.

Although the CSP text uses unfamiliar notations for conditional and iterative statements, the semantics conform with our usual notions.

CSP: while loop [Hoare 1985, §5.5, p.186]	$b * P$
CCS: iteration [Milner 1989, ch.8, p.175,177]	`while` b `do` A

Table 39: Iteration.

3.4.2 Further operators

Many other less familiar, but powerful, operators can be defined. Among those included in the CSP text are

- UNIX-like pipes [Hoare 1985, §4.4],
- subordinate processes [Hoare 1985, §4.5],
- restart after catastrophe [Hoare 1985, §5.4.2],
- checkpointing [Hoare 1985, §5.4.4],
- multiple checkpointing [Hoare 1985, §5.4.5], and
- alternating processes [Hoare 1985, §5.4.3].

The checkpointing operators use extraordinarily simple definitions for concepts that are very difficult to implement in distributed systems. (Possible primarily because the CSP semantic model allows *global* knowledge of a set of processes to be obtained easily.)

CCS examples include

- user-controlled iteration [Milner 1989, §9.1, p.187],
- for-loop style iteration [Milner 1989, §9.1, p.187],
- restart after catastrophe [Milner 1989, §9.2, p.193], and
- checkpointing [Milner 1989, §9.2, p.193].

These new operators are defined using the methods described in section 4.7 below.

3.5 Specification styles

So far we have presented the operators made available by the algebras but have said little about how they are used. (Detailed examples are beyond the scope of this paper—numerous examples can be found in the introductory literature [Hoare 1985, Milner 1989, Logrippo et al. 1992].)

LOTOS users have noted four distinct specification styles in practice [Faci et al. 1991]. They are described below in increasing order of abstraction.

Monolithic. Here all action sequences are listed. This is very verbose, but has applications when testing and debugging because it makes the sequences of actions performed explicit.

State-oriented. In this style all possible system states are modelled as distinct agents. Such a specification has the advantage of being directly implementable as a state automata, but the method is practical only for systems with few internal states.

Resource-oriented. Here each resource in the system being described is represented as a distinct agent. This can be seen as an advantage when implementing the system because the specification partitioning matches the implementation partitioning. This is also a disadvantage because the specification structure constrains possible implementations.

Constraint-oriented. This is the most abstract, implementation-independent style. Here behavioural constraints are each represented by an agent. The agent composition operators are then used to impose constraints on other agents via their interaction points. We can now see why CCS nomenclature has been favoured herein; "processes" may be thought of as merely defining behavioural restrictions, rather than the traditional notion of concurrent threads, and the "concurrency" operators may be thought of as ways of constraining behaviours, rather than the usual notion of simultaneous execution of such threads. LOTOS is the only process algebra that can take full advantage of this powerful specification style; the multi-party gate interaction mechanism was added to the emerging LOTOS standard specifically to support this [Logrippo et al. 1992, p.327]. Although constraint-oriented specifications can be written in CSP and CCS, their restriction (by convention) to bi-party interaction makes this clumsy.

4 Formal definitions

This section describes the formal definitions underlying the operators presented above. In section 3 we saw that LOTOS borrows features from both CSP and CCS. In its formal definition, however, LOTOS generally uses the same model as CCS. Therefore the remainder of this section will describe just the CSP and CCS models, with the latter applying to LOTOS unless otherwise noted.

4.1 Operational semantics

The algebras all have operational semantics; agents can be identified with system states [Milner 1989, §2.2, p.37] and "concurrent" actions are interleaved arbitrarily [Milner 1989, §3.3] [Hoare 1985, §2.3.3].

[Hoare 1985, §1.8.1] $\quad traces(STOP) = \{\langle\rangle\}$

[Hoare 1985, §1.8.1] $\quad traces(e \rightarrow P) = \{\langle\rangle\} \cup \{\langle e\rangle^\frown t | t \in traces(P)\}$

[Hoare 1985, §3.3.3] $\quad traces(P_1 \,\Box\, P_2) = traces(P_1) \cup traces(P_2)$

Figure 2: Some CSP trace semantics definitions.

4.1.1 Traces

The observable behaviour of a CSP process is defined by the set of all traces it may perform. A *trace* [Hoare 1985, §1.5] is a finite sequence of event symbols, recording the events performed up to some arbitrary moment in time. Only events visible to the environment of a process appear in its traces.

The (possibly infinite) set of all traces which are possible for a process are given by the *traces* function [Hoare 1985, §1.8]. (Just because a trace is *possible* for a process, does not mean that the trace will ever occur in a particular environment, however.) For instance, the rules in figure 2 define the set of traces possible for the stop process and the prefix and choice operators. The stop process can never undertake any event, so it can exhibit only the empty trace. The prefixing operator can additionally perform any trace that begins with the prefixed event e, followed by any trace of the subsequent process P. The choice operator can perform any of the traces possible for either of its arguments.

For example, these three rules are sufficient for us to determine the following set of traces.

$$traces(\mathtt{a} \rightarrow STOP \,\Box\, \mathtt{b} \rightarrow \mathtt{c} \rightarrow STOP) = \{\langle\rangle, \langle\mathtt{a}\rangle, \langle\mathtt{b}\rangle, \langle\mathtt{b}, \mathtt{c}\rangle\}$$

There are two principal behaviours: if the first choice is selected trace $\langle\mathtt{a}\rangle$ is possible; if the second choice is selected trace $\langle\mathtt{b}, \mathtt{c}\rangle$ is possible. All prefixes of these two traces are also potential behaviours of this process (because the environment may decline to offer sufficient events for the process to proceed).

4.1.2 Derivation trees

In the CCS approach the behaviour of an agent is expressed as a property of its derivation tree. Each agent has an associated set of *transitions* [Milner 1989, §2.3, p.38]. These represent the ability of the agent to perform an action; notation $A \xrightarrow{a} A'$ says that agent A can perform action a and subsequently become agent A'. Action a may be the special internal action τ.

As shown in figure 3, the meaning of operators is defined by transition rules [Milner 1989, p.46]. The first rule, for action prefix, tells us that agent $a.A$ can perform action a and then behaves like agent A. The first of the two rules for summation tells us that if agent A_1 can perform action a_1 and subsequently behave

$$\textbf{Act} \quad \frac{}{a.A \xrightarrow{a} A}$$

$$\textbf{Sum} \quad \frac{A_1 \xrightarrow{a_1} A_1'}{A_1 + A_2 \xrightarrow{a_1} A_1'} \qquad \frac{A_2 \xrightarrow{a_2} A_2'}{A_1 + A_2 \xrightarrow{a_2} A_2'}$$

Figure 3: Some CCS transition rule semantics definitions.

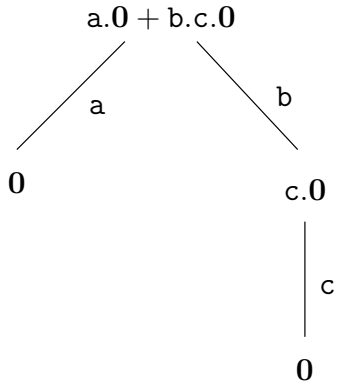

Figure 4: Derivation tree for summation.

like A_1', then agent $A_1 + A_2$ is capable of the same transition. Similarly for the second summation rule. (There is no explicit rule for the inactive agent because it cannot undertake any transitions.)

A *derivation tree* [Milner 1989, §2.2, p.40] collects together all *possible* agent derivatives. Such a tree may be infinite, with incomplete trees called "partial".

For example, figure 4 show the derivation tree defining all of the possible behaviours for the agent expression at the root. Each node is an agent (representing the system state at that point). Each arc is labelled by the action undertaken in changing from one agent to the next.

Comparing this with the equivalent example for CSP (section 4.1.1) we can see that although the CSP approach is intuitively simpler and easier to learn, the CCS model has the advantage of making the points at which choices were made explicit through branching. (Finding choice points in a set of linear traces involves searching for all traces with identical prefixes up to the point of interest.)

4.1.3 Choice vs. concurrency

It was noted above that the semantic models of the process algebras make it possible to determine where choices between alternative behaviours were made. However concurrency in the algebras is defined via an arbitrary interleaving of actions. The

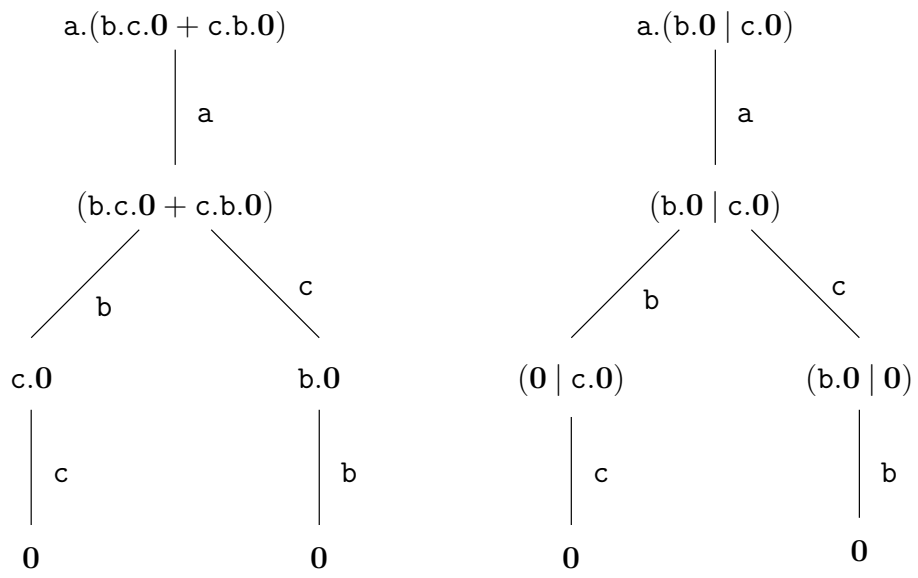

Figure 5: Equivalent derivation trees for choice and concurrency.

semantic models make no distinction between explicit choices made using one of the operators from section 3.2.2, or those due to "concurrent" behaviours.

In the CSP model, for instance, a process which first performs event a and then explicitly chooses between performing events b or c in either order defines the following set of traces.

$$\begin{aligned} & \mathit{traces}(\mathtt{a} \rightarrow (\mathtt{b} \rightarrow \mathtt{c} \rightarrow \mathit{STOP} \,\Box\, \mathtt{c} \rightarrow \mathtt{b} \rightarrow \mathit{STOP})) \\ = \; & \{\langle\rangle, \langle \mathtt{a} \rangle, \langle \mathtt{a}, \mathtt{b} \rangle, \langle \mathtt{a}, \mathtt{c} \rangle, \langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle, \langle \mathtt{a}, \mathtt{c}, \mathtt{b} \rangle\} \end{aligned}$$

However, the following process, defined using the interleaving operator, exhibits the *same* set of traces [Hoare 1985, p.120] and is therefore semantically equivalent.

$$\begin{aligned} & \mathit{traces}(\mathtt{a} \rightarrow (\mathtt{b} \rightarrow \mathit{STOP} \;|||\; \mathtt{c} \rightarrow \mathit{STOP})) \\ = \; & \{\langle\rangle, \langle \mathtt{a} \rangle, \langle \mathtt{a}, \mathtt{b} \rangle, \langle \mathtt{a}, \mathtt{c} \rangle, \langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle, \langle \mathtt{a}, \mathtt{c}, \mathtt{b} \rangle\} \end{aligned}$$

Merely examining the set of traces gives no indication as to the origin of alternative behaviours. Indeed, the CSP laws specifically equate concurrency with choice [Hoare 1985, p.71,p.120].

Similarly, CCS derivation trees do not distinguish choices due to summation from composition [Milner 1989, p.69]. Figure 5 shows two identical trees derived from different agent expressions, using the summation and composition operators, respectively. This equivalence is enshrined in CCS as the "expansion law" [Milner 1989, p.69], which allows any agent using the composition operator to be re-expressed without it.

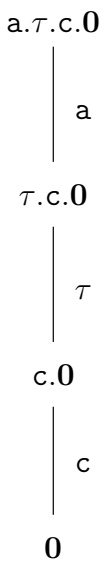

Figure 6: Derivation tree showing internal action.

In summary, interleaving semantics means that the concurrency operators are not fundamental in the process algebras; their effect can always be achieved using other operators.

4.2 Internal actions

In CSP hidden actions never appear in traces [Hoare 1985, §3.5.3]. For example, the following traces define the observable behaviour of a sequence of three actions in which the second is hidden.

$$traces((\mathtt{a} \rightarrow \mathtt{b} \rightarrow \mathtt{c} \rightarrow STOP)\backslash\{\mathtt{b}\}) = \{\langle\rangle, \langle\mathtt{a}\rangle, \langle\mathtt{a}, \mathtt{c}\rangle\}$$

By constrast, internal actions in CCS appear in derivation trees as τ [Milner 1989, p.40]. (Similarly for "**i**" actions in LOTOS [Bolognesi & Brinksma 1987, p.35].) For example, figure 6 shows the derivation tree for three consecutive CCS actions, where the second is internal. As explained in section 4.3.2 below, however, such actions are typically ignored when defining agent equivalence.

4.3 Nondeterminism

A major distinguishing feature of the models underlying the process algebras is their mechanism for handling nondeterminism.

4.3.1 Refusals

On their own CSP traces do not capture nondeterminism [Hoare 1985, §3.4]. For instance, the following two processes have identical sets of possible traces, but the first makes the choice arbitrarily, whereas the second allows the environment to decide which event to perform.

$$\begin{aligned} &\mathit{traces}((\mathtt{a} \to \mathit{STOP}) \sqcap (\mathtt{b} \to \mathit{STOP})) = \{\langle\rangle, \langle\mathtt{a}\rangle, \langle\mathtt{b}\rangle\} \\ &\mathit{traces}((\mathtt{a} \to \mathit{STOP}) \,\Box\, (\mathtt{b} \to \mathit{STOP})) = \{\langle\rangle, \langle\mathtt{a}\rangle, \langle\mathtt{b}\rangle\} \end{aligned}$$

To overcome this, CSP processes therefore also define sets of *refusals*. For each operator its refusals set records sets of events that *may* lead to deadlock when offered by the environment [Hoare 1985, §3.4]. The definitions for the nondeterministic and general choice operators are as follows.

$$\begin{aligned} &[\text{Hoare 1985}, \S 3.4.1] \qquad \mathit{refusals}(P_1 \sqcap P_2) = \mathit{refusals}(P_1) \cup \mathit{refusals}(P_2) \\ &[\text{Hoare 1985}, \S 3.4.1] \qquad \mathit{refusals}(P_1 \,\Box\, P_2) = \mathit{refusals}(P_1) \cap \mathit{refusals}(P_2) \end{aligned}$$

A nondeterministic choice may arbitrarily refuse to perform any of the events offered by either of its operands. By contrast, a general choice always accepts offerred events when it can, and therefore can refuse only those events that can be refused by both operands. For example, assuming $\alpha P_1 = \alpha P_2 = \{\mathtt{a}, \mathtt{b}\}$,

$$\begin{aligned} &\mathit{refusals}((\mathtt{a} \to \mathit{STOP}) \sqcap (\mathtt{b} \to \mathit{STOP})) = \{\{\}, \{\mathtt{a}\}, \{\mathtt{b}\}\} \\ &\mathit{refusals}((\mathtt{a} \to \mathit{STOP}) \,\Box\, (\mathtt{b} \to \mathit{STOP})) = \{\{\}\} \,. \end{aligned}$$

The first process may refuse to perform a, if the random choice has already selected the second alternative. The second process will never refuse any event in its alphabet.

Interestingly, the refusals function allows a precise definition of what we mean by "determinism". A deterministic process is one that never refuses any event it is capable of performing at the next step [Hoare 1985, §3.4].

An extension of the refusals concept is that of *failures*. This is a relation between traces and sets of events that cause a refusal after that trace has been performed [Hoare 1985, §3.9]. For instance,

$$\begin{aligned} &\mathit{failures}(\mathtt{a} \to \mathtt{b} \to \mathit{STOP}) = \\ &\{(\langle\rangle, \{\}), (\langle\rangle, \{\mathtt{b}\}), (\langle\mathtt{a}\rangle, \{\}), (\langle\mathtt{a}\rangle, \{\mathtt{a}\}), (\langle\mathtt{a}, \mathtt{b}\rangle, \{\}), \\ &(\langle\mathtt{a}, \mathtt{b}\rangle, \{\mathtt{a}\}), (\langle\mathtt{a}, \mathtt{b}\rangle, \{\mathtt{b}\}), (\langle\mathtt{a}, \mathtt{b}\rangle, \{\mathtt{a}, \mathtt{b}\})\} \end{aligned}$$

tells us that, for instance, after performing trace $\langle\mathtt{a}\rangle$ this process will refuse to perform event a again. The *failures* function thus encompasses both the *traces* and *refusals* functions.

The extreme form of nondeterminism is the process that may perform any event or refuse to perform any event. Such chaotic behaviour is rarely useful in a specification other than as a description of undesirable behaviour! It is defined as the result of unguarded recursion in CSP (because *any* process satisfies such a specification) [Hoare 1985, §3.8].

$$CHAOS = \mu X.X$$

To help with reasoning about such undesirable situations the *divergences* function returns the set of traces (if any) that a process may perform after which it behaves like *CHAOS* [Hoare 1985, §3.8.2]

4.3.2 Bisimulations

We noted above that CCS retains internal actions in derivation trees, with the intention of ignoring them where possible. However, in certain contexts, internal actions cannot be ignored because their presence influences nondeterminism and thus affects observable behaviour [Milner 1989, §2.3]. An internal action as the first action offerred by a summation has an observable influence on behaviour because it may spontaneously be performed, without the participation of the environment, and thus unexpectedly "decide" the choice, i.e.,

$$A_1 + \tau.A_2 \neq A_1 + A_2 \ .$$

As in CSP, CCS includes the effects of nondeterminism in its semantic model. Whereas standard automata theory would equate the two agents below, because they define the same "strings" of actions, CCS does not.

$$a.(A_1 + A_2) \neq a.A_1 + a.A_2$$

The distributive law is invalid here because the right-hand agent makes an initial nondeterministic choice whereas the left-hand one does not [Milner 1989, §4.1].

In keeping with its aim to be a framework for exploring different semantic models, CCS offers three quite distinct ways of assessing the equivalence of agents from their derivation trees.

Strong equivalence In this case the internal action τ is not treated as special [Milner 1989, p.84]. Thus, for two agents to be strongly equivalent, they must be capable of performing exactly the same sequences of actions, including internal ones.

Formally, this is defined through the concept of *bisimilarity*. A strong bisimulation is a binary relation $\mathcal{S}$ over agents. Two agents are strongly bisimilar, i.e., $(A_1, A_2) \in \mathcal{S}$, if we can find a relation such that

- whenever $A_1 \xrightarrow{a} A_1'$ then, for some A_2', $A_2 \xrightarrow{a} A_2'$ and $(A_1', A_2') \in \mathcal{S}$, and

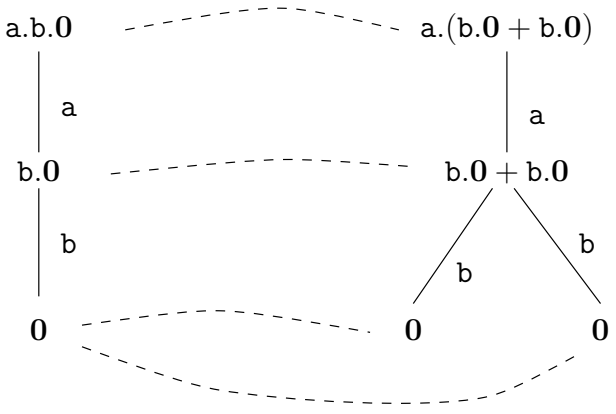

Figure 7: A strong bisimulation.

- vice versa [Milner 1989, §4.2, p.88].

In other words, whenever A_1 can perform some action a (which may be τ), then A_2 can perform the same action, and the two resulting agents are also strongly bisimilar; and similarly for any actions A_2 can perform.

For example, figure 7 represents a strong bisimulation of two agents $\mathtt{a}.\mathtt{b}.\mathbf{0}$ and $\mathtt{a}.(\mathtt{b}.\mathbf{0} + \mathtt{b}.\mathbf{0})$. The dotted lines link strongly equivalent agent expressions. The existence of this relation, representable as the following set of pairs,

$$\{(\mathtt{a}.\mathtt{b}.\mathbf{0}, \mathtt{a}.(\mathtt{b}.\mathbf{0} + \mathtt{b}.\mathbf{0})), (\mathtt{b}.\mathbf{0}, \mathtt{b}.\mathbf{0} + \mathtt{b}.\mathbf{0}), (\mathbf{0}, \mathbf{0})\}$$

allows us to conclude that these two agents are strongly bisimilar, written as $A_1 \sim A_2$ [Milner 1989, §4.2, p.90].

Observation equivalence The form of equivalence defined above does not conform with our intuitive notion that internal actions should not be significant in defining equivalence from the environment's viewpoint. *Observation equivalence*, also called weak equivalence or just bisimilarity, therefore seeks to ignore τ actions [Milner 1989, p.106].

To achieve this it defines a new transition system, with transitions denoted $A \stackrel{\hat{t}}{\Rightarrow} A'$, where $\hat{t}$ is a sequence of actions with all occurrences of τ removed ($\hat{t}$ may thus be the empty sequence) [Milner 1989, §5.1].

A (weak) bisimulation, $(A_1, A_2) \in \mathcal{S}$, thus exists if

- whenever $A_1 \stackrel{a}{\rightarrow} A_1'$ then, for some A_2', $A_2 \stackrel{\hat{a}}{\Rightarrow} A_2'$ and $(A_1', A_2') \in \mathcal{S}$, and
- vice versa [Milner 1989, §5.1, p.108].

In other words, whenever A_1 can perform some action a, then A_2 can perform the same action, possibly preceded and succeeded by some internal actions which we

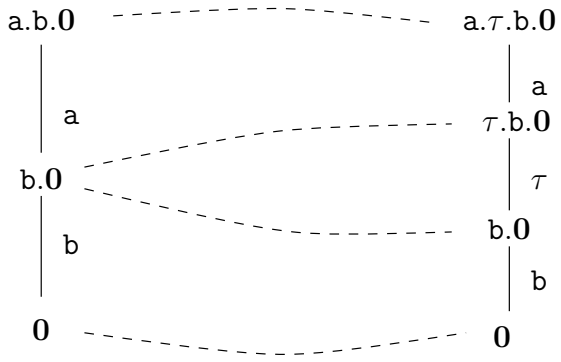

Figure 8: A weak bisimulation.

ignore, and the two resulting agents are also bisimilar; and similarly for any actions A_2 can perform. If action a is τ then the other agent must be able to make a transition without performing any *visible* actions.

For example, figure 8 represents a weak bisimulation of two agents, $\mathtt{a}.\mathtt{b}.\mathbf{0}$ and $\mathtt{a}.\tau.\mathtt{b}.\mathbf{0}$. The existence of this relation, representable as

$$\{(\mathtt{a}.\mathtt{b}.\mathbf{0}, \mathtt{a}.\tau.\mathtt{b}.\mathbf{0}), (\mathtt{b}.\mathbf{0}, \tau.\mathtt{b}.\mathbf{0}), (\mathtt{b}.\mathbf{0}, \mathtt{b}.\mathbf{0}), (\mathbf{0}, \mathbf{0})\}$$

allows us to conclude that these two agents are observationally equivalent, denoted $A_1 \approx A_2$ [Milner 1989, §4.2, p.109].

Equality In many situations strong bisimilarity is a satisfactory relation for reasoning about the equivalence of agents. Unfortunately, however, there is a special case that prevents us from adopting strong bisimilarity as a definition of agent *equality*. Although some agent A is strongly equivalent to agent $\tau.A$, these two agents are not equal because they cannot be substituted for one another as the argument to a summation [Milner 1989, §5.3]. Agent $\tau.A$ may nondeterministically make the choice "internally", whereas A may not.

Therefore, to complete the notion of agent equality, also called observation congruence, we define yet another transition system, using transitions denoted $A \stackrel{t}{\Rightarrow} A'$. For this to be valid transition $A(\stackrel{\tau}{\rightarrow})^* \stackrel{t}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* A'$ must be possible, i.e., A must perform *at least* the actions in t, possibly with additional internal actions [Milner 1989, p.107].

Given this we can say that two agents are equal in all contexts, i.e., $A_1 = A_2$, if

- whenever $A_1 \stackrel{a}{\rightarrow} A_1'$ then, for some A_2', $A_2 \stackrel{a}{\Rightarrow} A_2'$ and $A_1' \approx A_2'$, and
- vice versa [Milner 1989, §7.2].

Thus if a is τ then the other agent must be able to perform at least one τ as well.

Others relations Other equivalence relations can be defined, of course, although those above are the most common. It is worth noting, however, the notion of *testing* equivalence in LOTOS which equates behaviour expressions that cannot be distinguished through experimentation by the environment [Bolognesi & Brinksma 1987, p.43].

4.4 Equivalence

We have seen that the way nondeterminism is modelled has a profound influence on the way the process algebras define equivalence of agents. We can now summarise the two major approaches.

In CSP a process P is uniquely defined by the triple consisting of its alphabet, those traces leading to particular set of refusals, and those traces leading to chaotic behaviour [Hoare 1985, p.130], i.e.,

$$(\alpha P, \mathit{failures}(P), \mathit{divergences}(P)) \ .$$

The use of explicit *refusals* sets in CSP has the advantage of avoiding the need to distinguish observational equivalence from congruence [Hoare 1985, p.248-9].

In CCS (and LOTOS) there are many possible ways of defining agent equivalence, depending on whether internal actions are seen as significant or not. There are three principal views of agent equivalence, strong, weak and equality, which have the following relationship for two agents A_1 and A_2 [Milner 1989, p.154]:

$$(A_1 \sim A_2) \Rightarrow (A_1 = A_2) \Rightarrow (A_1 \approx A_2) \ .$$

For instance, two agents that are equal are also weakly equivalent, but not necessarily strongly so. Although seemingly complex, the CCS model has the advantage of being able to distinguish between nondeterministic agents that are equivalent in CSP [Hoare 1985, §7.4.1].

4.5 Algebraic laws

As their name suggests, the process algebras obey algebraic laws which define agent equivalence [Hoare 1985, §1.3] [Milner 1989, ch.3] and allow us to undertake transformations for verification and program derivation [Hoare 1985, p.37]. Based on the above concepts of agent equality, these laws can be stated straightforwardly. We make use of the example laws given below in section 4.8.1.

4.5.1 Algebraic laws

Figure 9 shows some typical laws for CSP processes. For instance, law **L2** tells us that hiding a set of events B and then hiding another set C is equivalent to hiding the union of the two sets. Law **L5** tells us that a concealed action occurs invisibly,

L1 [Hoare 1985, §3.3.1] $P \,\Box\, P = P$

L2 [Hoare 1985, §3.5.1] $(P \backslash B) \backslash C = P \backslash (B \cup C)$

L3 [Hoare 1985, §3.3.1] $P_1 \,\Box\, (P_2 \,\Box\, P_3) = (P_1 \,\Box\, P_2) \,\Box\, P_3$

L4 [Hoare 1985, §3.3.1] $P \,\Box\, STOP = P$

L5 [Hoare 1985, §3.5.1]
$$(e \rightarrow P) \backslash C = \begin{cases} e \rightarrow (P \backslash C) & \text{if } e \notin C \\ P \backslash C & \text{if } e \in C \end{cases}$$

L10 [Hoare 1985, §3.5.1]
$$(x{:}\, B \rightarrow P(x)) \backslash C = Q \sqcap (Q \,\Box\, (x{:}\, (B - C) \rightarrow P(x)))$$
(if $C \cap B \neq \{\}$ and $Q = \bigsqcap_{x:B \cap C} P(x) \backslash C$)

Figure 9: Some CSP algebraic laws

as if it was never present, but other actions are unaffected by the concealment operator. Law **L10** is one of the most complex algebraic laws in CSP. It states the interaction between the concealment and deterministic choice operators. Concealed behaviours, embodied by Q, may or may not happen spontaneously. Thus there is the possibility of nondeterministically choosing to behave like Q, or to offer the environment the ability to choose one of the observable events (from set $B - C$). In the latter case, however, it is not possible for the environment to distinguish between an observable event from the original set B, or a different occurrence of the same event that is offerred by Q *after* it has performed a concealed event, hence the general choice on the right-hand side.

An exciting possibility suggested by the existence of these laws is to apply them to the programming language occam, itself closely modelled on CSP. Some work has been undertaken in this area but with restrictions to exclude iteration and the requirement that programs are expressed in a normal form [Roscoe & Hoare 1986]. This work merits further research.

4.5.2 Equational laws

The "equational" laws are divided into three groups in CCS [Milner 1989, p.61].

- The *dynamic* laws, for the prefix and summation operators, are so-named because after their application the operator does not appear in the subsequent agent expression.
- The *static* laws, for composition, restriction, and relabelling, are so-named because the operators persist after each transition.

1(3)	[Milner 1989, *p*.62]	$A + A = A$
9(2)	[Milner 1989, *p*.80]	$A\backslash K\backslash L = A\backslash(K \cup L)$
1(2)	[Milner 1989, *p*.62]	$A_1 + (A_2 + A_3) = (A_1 + A_2) + A_3$
1(4)	[Milner 1989, *p*.62]	$A + \mathbf{0} = A$
7(1)	[Milner 1989, *p*.70]	$(a.A)\backslash L = \begin{cases} a.(A\backslash L) & \text{if } a \notin L \cup \overline{L} \\ \mathbf{0} & \text{otherwise} \end{cases}$
2(2)	[Milner 1989, *p*.62]	$A + \tau.A = \tau.A$

Figure 10: Some CCS equational laws.

- The *expansion* law links the two groups above. As mentioned earlier, it is of particular significance because it allows the behaviour of agent composition to be expressed in terms of dynamic laws only.

Figure 10 shows some representative examples of CCS laws. Law **7(1)** tells us that restricted actions cannot occur at all, hence an agent is equivalent to the inactive agent if prefixed by such an action. Law **2(2)** is valid because the internal action has no observable effect in the situation where either choice leads to the same agent, even though it is the first action in a summation.

Milner has shown completeness for finite agent axioms [Milner 1989, §7.4-5].

4.6 Value-passing

So far we have discussed the semantics of the basic algebras only. Extension to the value-passing algebras is straightforward, however, and is based on representing parameterised entities by (possibly infinite) sets of basic ones [Milner 1989, §2.8] [Hoare 1985, §4.1]. Indeed, CCS has an elegant and concise set of rules for translating between the basic and value-passing calculi [Milner 1989, p.56].

4.6.1 Message passing

To model sending the value of some expression e, the process algebras simply make the corresponding value part of the channel or port name. In CSP an output action on channel c is equivalent to performing an action with a unique, compound name $c.e$ [Hoare 1985, §4.2], i.e.,

$$(c!e \rightarrow P) = (c.e \rightarrow P) \ .$$

The approach is the same in CCS except that a subscripted notation is used [Milner 1989, p.54], i.e.,

$$(\overline{p}(e).A) = (\overline{p_e}.A) \ .$$

To model reception of some value $v \in V$, the interaction is again modelled using a compound action name. (Thus, to interact correctly, the sender and receiver must agree on the channel name *and* the data value to be "transmitted".) Since a received value can affect subsequent behaviour, the next agent is parameterised by the value. Thus, in CCS, there is a choice between possible behaviours decided by the value "received" [Milner 1989, p.54],

$$p(v).A(v) = \sum\nolimits_{v \in V} p_v.A_v \ .$$

Similarly, in CSP, the same effect is achieved using two functions to extract the *channel* name and *message* component from a compound action name [Hoare 1985, §4.2],

$$(c?v \rightarrow P(v)) = (y{:}\ \{y | \mathit{channel}(y) = c\} \rightarrow P(\mathit{message}(y))) \ .$$

LOTOS uses the same approach but, due to its ability to perform multi-value, multi-way communications, it associates a *tuple* of data values $\langle v_1, \ldots, v_n \rangle$ with the gate name. The gate identifier and the entire tuple form a structured name [Logrippo et al. 1992, §3.2]. Thus, letting x_i represent those values used in B, and v_i a particular instantiation of them,

$$(g\ c_1 \ldots c_n;\ B(x_i)) = (g\langle v_1, \ldots, v_n \rangle;\ B(v_i))\ [\,]\ (g\langle \ldots \rangle;\ B(\ldots))\ [\,] \ldots \ .$$

4.6.2 Parameter passing

Parameterised agent names are represented in much the same way, modelling an agent parameterised by a data value by a family of agents for each possible value. In CCS, agent $A(v)$ is equivalent to a family of agents A_v for each v [Milner 1989, p.54]. In CSP, notation $P(v)$ represents a process-valued function but we can also use unique indexed names P_v [Hoare 1985, pp.32-3].

4.6.3 Conditional behaviour

Decisions based on the current value of data are represented by rules that return agents depending on the (static) data values in each agent instance from the family of agents (one for *every* possible state!).

The one-armed conditional statement in CCS is defined as [Milner 1989, p.56],

$$\textbf{if } b \textbf{ then } A = \begin{cases} A & \text{if } b = \mathit{true} \\ \mathbf{0} & \text{otherwise} \ . \end{cases}$$

The unusual if-then-else notation in CSP is defined as [Hoare 1985, p.188],

$$\begin{aligned} P_1 \lhd \mathit{true} \rhd P_2 &= P_1 \\ P_1 \lhd \mathit{false} \rhd P_2 &= P_2 \ . \end{aligned}$$

$$Done \stackrel{\text{def}}{=} \overline{\texttt{done}}.\mathbf{0}$$

$$A_1 \; Before \; A_2 \stackrel{\text{def}}{=} (A_1[b/\texttt{done}] \mid b.A_2)\backslash\{b\}$$

Figure 11: Sequential composition defined using CCS equations.

4.7 Creating operators

The operators used in the process algebras are defined through simple formal definitions. CSP and CCS are extensible, and thus encourage their users to define new operators, either through further formal definitions or through equations using existing operators. In this section we look at both ways of defining operators, using sequential composition as a common example.

4.7.1 Equations

The simplest way of defining new operators is to make use of the standard mechanism for creating agents (see section 3.2.8) and to build them from existing ones.

For example, although sequential composition is not primitive in CCS, it can be defined easily using existing features [Milner 1989, §8.2].

As shown in figure 11 we firstly define an agent for denoting successful termination. The name `done` is reserved for denoting "success". Sequential composition itself is then defined elegantly as a special case of general agent composition. On the right a new name b is introduced; this action is used to co-ordinate the two "concurrent" agents. Through relabelling of `done`, b is the last action performed by A_1. Via prefixing it is also the first to be performed by the second argument to the composition operator. Thus A_2 cannnot begin execution until A_1 has performed agent *Done*. Lastly, the synchronising action b is restricted so that it is not seen by the environment.

The **accept** operator in LOTOS can be defined similarly with a value tuple associated with a special action δ used to denote successful termination [Bolognesi & Brinksma 1987, pp.54-5].

4.7.2 Formal definitions

We have seen in section 4.3 that the behaviour of operators can be defined in terms of traces in CSP and transitions in CCS and LOTOS. This mechanism is also available to CSP and CCS users who want a powerful way of defining their own operators. In this section we illustrate the two styles of definition, using the formal definitions of the standard CSP and LOTOS sequential composition features.

To define sequential composition in CSP we firstly define the possible traces [Hoare 1985, §5.3.1]. In figure 12 *SKIP* can perform only a special event "✓", used

$$
\begin{aligned}
traces(SKIP) &= \{\langle\rangle, \langle\checkmark\rangle\} \\
traces(P_1;P_2) &= \{s | s \in traces(P_1) \wedge \neg\langle\checkmark\rangle \textbf{ in } s\} \cup \\
&\qquad \{s^\wedge t | s^\wedge\langle\checkmark\rangle \in traces(P_1) \wedge t \in traces(P_2)\}
\end{aligned}
$$

Figure 12: Sequential composition (partly) defined using CSP traces.

$$
\frac{}{\textbf{exit} \xrightarrow{\delta} \textbf{stop}}
$$

$$
\frac{B_1 \xrightarrow{g} B_1'}{B_1 \gg B_2 \xrightarrow{g} B_1' \gg B_2} \qquad \frac{B_1 \xrightarrow{\delta} B_1'}{B_1 \gg B_2 \xrightarrow{\mathbf{i}} B_2}
$$

Figure 13: Sequential composition defined using LOTOS derivation rules.

to denote successful termination. The sequential composition operator can perform any trace possible for its first argument and any trace in which the first process successfully terminates, following by a trace of the second process. The ✓ event itself is not intended to be seen by the environment, so care is taken to exclude it from the traces of P_1. Definitions in a like style are then given for the *refusals* sets and *divergences* of the two operators to complete the description of sequential composition [Hoare 1985, p.179].

Like CCS, LOTOS defines operators through their derivation rules. The rules for successful termination and sequential composition are shown in figure 13 [Bolognesi & Brinksma 1987, §2.7].

The **exit** behaviour expression can perform only the special gate action δ, which serves the same purpose as ✓ and `done` above. The first of the two rules defining the sequential composition operator states that if the first expression is capable of performing g, then the entire composition is also capable of this action. The second rule says that if the first expression is capable of performing δ, i.e., it can successfully terminate, then the composition starts behaving like B_2. Notice, however, that the special δ action is represented in the composition as an *internal* action so that it is not seen by the environment. If an expression performs **exit** in another context then the δ action will be visible. This makes it possible to easily define the meaning of a group of parallel, successfully terminating processes. They must all synchronise on the δ gate before the entire parallel composition can itself successfully terminate (and visibly perform δ for use in its own environment).

$$
\begin{array}{lll}
 & A_1 + \tau.(A_1 + A_2) & \\
= & A_1 + (A_1 + A_2) + \tau.(A_1 + A_2) & \mathbf{2(2)} \\
= & (A_1 + A_2) + A_2 + \tau.(A_1 + A_2) & \mathbf{1(2)} \\
= & (A_1 + A_2) + \tau.(A_1 + A_2) & \mathbf{1(3)} \\
= & \tau.(A_1 + A_2) & \mathbf{2(2)}
\end{array}
$$

Figure 14: Proof in CCS using equational laws.

4.8 Verification

Apart from specification, the process algebras offer the ability to reason about systems. In general there are two approaches.

4.8.1 Verification using algebraic laws

Algebraic laws, such as those in sections 4.5.1 and 4.5.2, can be used to prove simple theorems [Hoare 1985, p.37].

In CCS, for instance, Milner asks whether the following (somewhat counter-intuitive) equality is true [Milner 1989, p.63].

$$A_1 + \tau.(A_1 + A_2) = \tau.(A_1 + A_2)$$

As shown in figure 14, we can confirm that it is indeed valid by rewriting the left-hand side to match the right-hand side using the laws given in section 4.5.2.

Similarly, in CSP, Hoare presents the counter-example in figure 15 to prove that concealment does not distribute backwards through general choice [Hoare 1985, p.114]. An attempt to show that

$$
\begin{array}{ll}
 & (e_1 \rightarrow STOP \,\Box\, e_2 \rightarrow STOP)\backslash\{e_1\} \\
= & ((e_1 \rightarrow STOP)\backslash\{e_1\}) \,\Box\, ((e_2 \rightarrow STOP)\backslash\{e_1\})
\end{array}
$$

using the laws in section 4.5.1 cannot succeed because the left-hand process can autonomously decide to become inactive, whereas the right-hand process always offers the environment event e_2.

Such proof methods have obvious advantages in terms of simplicity and their potential for automation [Roscoe & Hoare 1986, p.70].

4.8.2 General verification methods

Although the algebraic laws offer a straightforward means of verification through re-writing, many desirable features of a system are expressed more naturally as semantic properties. The algebras thus have related logics which form the basis for

$$
\begin{array}{lll}
 & (e_1 \rightarrow STOP \,\Box\, e_2 \rightarrow STOP)\backslash\{e_1\} & \\
= & STOP \sqcap (STOP \,\Box\, (e_2 \rightarrow STOP)) & \textbf{L10} \\
= & STOP \sqcap (e_2 \rightarrow STOP) & \textbf{L4} \\
\neq & e_2 \rightarrow STOP & \\
= & STOP \,\Box\, (e_2 \rightarrow STOP) & \textbf{L4} \\
= & ((e_1 \rightarrow STOP)\backslash\{e_1\}) \,\Box\, ((e_2 \rightarrow STOP)\backslash\{e_1\}) & \textbf{L5}
\end{array}
$$

Figure 15: Proof in CSP using algebraic laws.

L2B if P **sat** $S(tr)$
then $(e \rightarrow P)$ **sat** $((tr = \langle\rangle \wedge e \notin ref) \vee (tr_0 = e \wedge S(tr'))$

L5 if P_1 **sat** S_1
and P_2 **sat** S_2
then $(P_1 \,\Box\, P_2)$ **sat** (**if** $tr = \langle\rangle$ **then** $S_1 \wedge S_2$ **else** $S_1 \vee S_2$)

Figure 16: Some CSP proof rules.

more powerful techniques of formal verification [Hoare 1985, §1.10] [Milner 1989, ch.10].

In CSP, a specification S is a predicate on traces and refusals [Hoare 1985, §1.10]. The variable name *tr* is used to represent an arbitrary trace, and *ref* an arbitrary refusal set. Numerous operators on traces are also provided [Hoare 1985, §1.6, §1.9], for example, tr_0 is the first event in trace *tr* [Hoare 1985, §1.6.3], and tr' is the tail of *tr* [Hoare 1985, §1.6.3].

Formal proofs then aim to show that a CSP process P satisfies such a specification S, denoted P **sat** S [Hoare 1985, §1.10.1]. The trace definitions for CSP operators are used to determine which events are possible for P, and satisfaction rules are provided for each operator. Two such rules are shown in figure 16 [Hoare 1985, §3.7.1].

Rule **L2B** tells us that if process P satisfies some specification S on a trace *tr*, then process $e \rightarrow P$ has the property that when it has not yet performed any events it will never refuse to perform event e. However, if event e has already been performed, then the remainder of the trace will satisfy property S. Rule **L5** tells us that before a choice is made the properties of both operands are respected, e.g.,

Proof

L2B $(\texttt{tea} \rightarrow STOP)$ **sat** $(\textbf{if } tr = \langle\rangle \textbf{ then } \texttt{tea} \notin ref)$

L5 $(\texttt{tea} \rightarrow STOP \,\Box\, \texttt{coffee} \rightarrow STOP)$ **sat**
$(\textbf{if } tr = \langle\rangle \textbf{ then } \texttt{tea} \notin ref)$

L2B VM **sat** $NICE$

qed

Figure 17: A formal proof in CSP.

both sets of refusals are valid, but after a particular path has been taken only the properties of that alternative are guaranteed.

As an example we will use these laws to prove a desirable property of a beverage vending machine, namely that "I *can* get tea if I pay". The vending machine is defined by the following process:

$$VM = (\texttt{coin} \rightarrow (\texttt{tea} \rightarrow STOP \,\Box\, \texttt{coffee} \rightarrow STOP))$$

After accepting a coin it offers a choice between dispensing tea or coffee. The desirable property is specified as follows:

$$NICE = (\textbf{if } tr = \langle \texttt{coin} \rangle \textbf{ then } \texttt{tea} \notin ref)$$

After a coin has been accepted we have stated that the machine must not refuse to dispense tea.

The proof, shown in figure 17, proceeds by firstly showing that a process prefixed by `tea` cannot refuse this event, using the first disjunct in law **L2B**. This result is then used, via law **L5**, to show that a general choice in which the "tea process" is an alternative also has this property. Finally, law **L2B** is invoked again, using the second disjunct this time, to show that VM has the same property after the coin has been inserted.

CSP proofs are typically built up from components in this way [Hoare 1985, p.247].

The proof methods proposed for CCS are quite different. A modal logic is defined whose formulae F express properties of derivation trees [Milner 1989, §10.2]. Among the many operators available for constructing formulae, two commonly-used modal operators are

- $[a]F$, which states that if action a can be performed then F must hold afterwards, and
- $\langle a \rangle F$, which asserts that it is possible to do a and reach a state satisfying F.

i) $A \models \langle a \rangle F$ if, for some A', $A \xrightarrow{a} A'$ and $A' \models F$.

ii) $A \models [a]F$ if, for all A' such that $A \xrightarrow{a} A'$, $A' \models F$.

Figure 18: Some CCS proof rules.

Proof

$VM \models NICE$

ii) iff $(\texttt{tea}.\mathbf{0} + \texttt{coffee}.\mathbf{0}) \models \langle \texttt{tea} \rangle true$

(because $(\texttt{tea}.\mathbf{0} + \texttt{coffee}.\mathbf{0}) = \{A | VM \xrightarrow{\texttt{coin}} A\}$)

i) iff $\mathbf{0} \models true$

(because $\texttt{tea} \in \{a | (\texttt{tea}.\mathbf{0} + \texttt{coffee}.\mathbf{0}) \xrightarrow{a} X\}$)

qed

Figure 19: A formal proof in CCS.

Proofs then aim to show that some agent A satisfies some formula F, denoted $A \models F$ [Milner 1989, p.214]. The derivation rules for CCS operators are used to determine possible actions for A, and the satisfaction relation is defined by induction on the structure of formulae [Milner 1989, p.214]. Figure 18 shows the rules for the two operators above.

To illustrate the proof style we will undertake an example similar to that above. Here the aim is to prove "I *may* get tea if I pay". The vending machine and the desirable property are expressed as follows.

$$VM \stackrel{\text{def}}{=} \texttt{coin}.(\texttt{tea}.\mathbf{0} + \texttt{coffee}.\mathbf{0})$$
$$NICE = [\texttt{coin}]\langle \texttt{tea} \rangle true$$

The proof then proceeds as shown in figure 19. In the first step we immediately discharge the need to show that action `coin` can be performed because it is the only derivative of VM. In the second step we show that because `tea` is a possible derivative of the remaining agent expression we reach formula *true* and the proof is thus complete. (Proving "I *can* get tea" is harder in this instance, however. Liveness properties require an extension from modal to temporal logic [Milner 1989, p.251].)

5 Current research

The process algebras are by no means static. A considerable amount of research is currently being undertaken to enhance them. We outline some of the major research

areas below to highlight current limitations of the languages. (A good starting point for readers keen on pursuing these topics further is the proceedings of the annual "FORTE" conferences, published by North-Holland as the series "Formal Description Techniques".)

Timed algebras Although the process algebras can define the relative order in which actions occur, they say nothing about the *absolute time* at which events happen. Numerous proposals have been put forward for ways of adding real time to the process algebras. In some cases special "tick" actions have been used to explicitly represent the passage of time; another common approach has been to add a "delay" operator. In all attempts considerable difficulty has been encountered in dealing with the way choice operators become time-sensitive and the fact that interleaving semantics cannot model the temporal overlapping of actions.

True concurrency semantics Although conceptually simple the interleaving semantics used by the process algebras mean that the "concurrency" operators are not fundamental and hinder the ability to add time to the languages. True concurrency semantics, in which traces become only partially ordered, have been suggested as a more accurate model of concurrency. There are two principal methods: multi-set traces use linear traces with a set of actions listed at each step; causally ordered transitions maintain pointers denoting causal relationships between events in the traces.

"Biased" algebras The process algebras make no promises regarding the fairness of nondeterministic choices. In some contexts, however, it is felt to be necessary to make statements about the frequency of, for example, good vs. error behaviour. To this end methods of adding probabilities to alternatives, or priorities to actions, have been suggested, but good semantic definitions remain elusive.

Decomposable actions Actions in the process algebras are indivisible. Modern software derivation techniques, however, rely on "refining" high-level specifications into more concrete, lower-level ones. There has therefore been some interest in methods for decomposing "atomic" actions into their components, but much work remains.

Improved verification techniques More verification techniques for process algebras are under investigation.

Tools A major advantage of the process algebras, due to their operational semantics, is that they are very amenable to automated support. Many tools for rapid

prototyping, simulation and analysis of process algebra specifications have been developed. Research continues, although many tools are now sufficiently advanced to be available as commercial products.

6 Conclusion

This article has summarised the language features and underlying definitions of the three most widely known process algebraic specification language, CSP, CCS and LOTOS. We have seen that, despite their superficial similarities, there are many significant differences between them.

An obvious question at this point is: which is best? There is no simple answer to this. Each of the languages has advantages and disadvantages in particular application domains. CSP is easy to learn, offers simple semantics and verification techniques, and has the important practical advantage of being implemented as occam. Some questions about the consistency and completeness of its laws remain, however. CCS has the advantages of being minimal and elegant. Some users may see its paucity of operators and bi-party interaction model as disadvantages, however, and the subtle distinction between observation equivalence and congruence is an unfortunate complication. LOTOS is often criticised due to its verbose syntax. As an international standard, however, it has attracted a great deal of attention and has better tool support than CSP and CCS combined. Furthermore, its multi-party interaction mechanism provides exceptional expressive power.

Acknowledgements

This article is based on a tutorial presented to the Fifth International Conference on Formal Description Techniques. I wish to thank the tutorial attendees for their encouragement, advice and corrections, especially Luigi Logrippo and Tommaso Bolognesi. Thanks also to Ed Kazmierczak for his thorough review of this article.

References

[Bolognesi & Brinksma 1987] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[Bergstra & Klop 1984] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109–137, Jan–Mar 1984.

[Faci et al. 1991] M. Faci, L. Logrippo, and B. Stepien. Formal specification of telephone systems in LOTOS: the constraint-oriented style approach. *Computer Networks and ISDN Systems*, 21(1):53–67, 1991.

[Iso 1989] International Organisation for Standardisation. LOTOS: A formal description technique based on the temporal ordering of observational behaviour, 1989. IS 8807.

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

[Loureiro et al. 1992] A. Loureiro, S. Chanson, and S. Vuong. FDT tools for protocol development. In *Tutorial Proceedings: 5th International Conference on Formal Description Techniques*, Perros-Guirec, France, October 1992.

[Logrippo et al. 1992] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples. *Computer Networks and ISDN Systems*, 23(5):325–342, February 1992.

[Milner 1989] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Milner 1991] R. Milner. *The Polyadic π-Calculus: a Tutorial*. Computer Science Department, University of Edinburgh, October 1991.

[Roscoe & Hoare 1986] A.W. Roscoe and C.A.R. Hoare. The laws of occam programming. Technical Report PRG-53, Oxford University Computing Laboratory, February 1986.

[Raise 1992] The RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, 1992.