Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

R.J. van Glabbeek

Notes on the methodology of CCS and CSP

Computer Science/Department of Software Technology Report CS-R8624 August

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Notes on the Methodology of CCS and CSP

R.J. van Glabbeek
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In this paper the methodology of some theories of concurrency (mainly CCS and CSP) is analysed, focusing on the following topics: the representation of processes, the identification issue, and the treatment of nondeterminism, communication, recursion, abstraction, divergence and deadlock behaviour. Process algebra turns out to be a useful instrument for comparing the various theories.

1980 Mathematics Subject Classification: 68B10, 68C01, 68D25, 68F20.
1982 CR Categories: F.1.2, F.3.2, F.4.3, D.3.1.
Key words & phrases: concurrency, CCS, CSP, process algebra.
Note: Sponsored in part by Esprit project no. 432, Meteor (An Integrated Formal Approach to Industrial Software Development).

1. Introduction: Theories of Concurrency

This is an investigation into the methodology of some theories of concurrency. In general a concurrency theory offers a framework for the specification (or even the design) of parallel processes and the verification of statements about them. The features of concurrency, expressible within such a framework, include communication between parallel processes, deadlock behaviour, abstraction from internal steps, fairness, nondeterminism, priorities in the choice of actions, tight regions, etc.

Some interesting theories of concurrency are:

- The theory of Petri Nets (see for instance Reisig [16])
- Trace theory (see for instance Rem [17])
- Milner's Calculus of Communicating Systems (CCS) ([11])
- Hoare's theory of Communicating Sequential Processes (CSP) ([9])
- The topological process theory of De Bakker & Zucker ([3,4])
- The Algebra of Communicating Processes (ACP) of Bergstra & Klop ([5]).

This paper will be mainly devoted to CCS and CSP.

Report CS-R8624
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Table of contents

2. Models and Calculi

A framework for studying concurrency often has the shape of a mathematical model. Parameters in the classification of these models are the features captured by the model, the identifications made on processes and the particular way of representing them. These criteria (in reverse order) will be explained and applied in the next three sections.
Apart from being a mathematical model, the framework in question can also be a calculus for the verification of statements about processes, formulated in an algebraical language. For practical applications this means that instead of checking that a process fits into a selected model, one has to check that it operates in an environment where the rules and axioms of the calculus are satisfied.
Some theories of concurrency use both models and calculi, but with different emphasis on one of those. This provides an important criterion for method decomposition.
The theory of Petri nets establishes a model of concurrency, without a calculus, and so does the topological process theory.
Trace theory also presents a model, but a number of calculi, axiomatising this model, have been developed, starting with Kleene [10] and Salomaa [18].
CSP, as presented in Brookes, Hoare and Roscoe [7] and in Hoare [9], provides a model, illustrated with some algebraical laws. Systematic axiomatisations of CSP can be found in Brookes [8] and De Nicola [13].
CCS is essentially a calculus, but the rules and axioms in this calculus are presented as laws, valid in a given model.
ACP is a calculus that is not bound to a particular model. It is the core of a family of axioms systems, each describing some features of concurrency.
The systematic exploration of (families of) algebraical calculi is called process algebra. In process algebra models are merely used as illustration and for constructing consistency proofs. This model -independence makes process algebra, apart from a tool for studying concurrency directly, also suitable for analysing the different models: the presentation of axiomatisations illuminates their differences and similarities. Axioms for CCS and CSP and for the identification criteria discussed in §4, will be presented in §6. Most of them are taken from [8], [9], [11] and [13].

3. How to represent a process

3.1 Models of concurrency

As can be extracted from the previous section, five of the six concurrency theories mentioned in §1, work with an explicit model. In all these models processes are represented differently. In Net theory one of the ways to represent a process is as a *labeled Petri net* with a given initial configuration. De Bakker and Zucker use a topological construction to represent processes. In trace

theory a process is represented by a *trace set* and in CSP by a *failure set*; both these concepts will be explained below. Milner represents a process by a *synchronisation tree*. This is the same (though slightly less general) as what is known as a *state transition diagram* or *process graph*, and will be explained in section 3.5. In [5] three models of ACP are mentioned: its initial algebra, a *projective limit* model (resembling the topological construction of De Bakker and Zucker) and a process graph model.

3.2 Atomic actions

In all concurrency theories mentioned in this paper, the most elementary components of a process are the so-called *atomic actions*. They are indivisible and not subject to further investigations. Now a process just performs atomic actions $a,b,c,...$ out of a given alphabet A.

3.3 Trace sets

In trace theory a process is considered to be fully determined by the possible sequences of atomic actions it can perform (its *traces*). Therefore a model is created in which a process is represented by the set of its traces. Usually trace sets are required to be prefix closed and to contain only finite traces of infinite processes. In this setting any non-empty prefix closed set of finite words over A represents a process.

3.4 Failure sets

In CSP a process is considered to run in an environment which can veto the performance of certain atomic actions. Moreover the environment can decide to do so during the execution of a process. If, at some moment in the execution, no action in which the process is prepared to engage is allowed by the environment, then deadlock occurs, which is considered to be observable. Now, a finite experiment with a process yields either a trace, or a trace followed by deadlock. In the last case the trace $\sigma \in A^*$ may be recorded, as well as the set $X \subseteq A$ of actions allowed by the environment at the time of stagnation. An element of X is said to be refused by the process and X is called a *refusal set* of the process after performance of σ. Now the pair $\langle \sigma, X \rangle$ is a *failure pair* of the process and the set of all failure pairs of a process is called its *failure set*. Since in CSP a process is considered to be fully determined by the observations obtainable from all possible finite experiments (as described above) with the process, a failure model of CSP is created in which a process is represented by its failure set. In this model any set $F \subseteq A^* \times Pow(A)$ satisfying

I. $\langle \varepsilon, \emptyset \rangle \in F$

II. $\langle \sigma * \rho, \emptyset \rangle \in F \Rightarrow \langle \sigma, \emptyset \rangle \in F$

III. $\langle \sigma, Y \rangle \in F \wedge X \subseteq Y \Rightarrow \langle \sigma, X \rangle \in F$

IV. $\langle \sigma, X \rangle \in F \wedge \forall a \in Y (\langle \sigma * a, \emptyset \rangle \notin F) \Rightarrow \langle \sigma, X \cup Y \rangle \in F,$

represents a process. Here ε denotes the empty trace, $\emptyset$ the empty set and $\sigma * \rho$ the concatenation of the traces σ and ρ.

3.5 State transition diagrams

In CCS a process is considered to go through a number of states. The states are determined by the possible courses of action the process is ready to engage in. In a state transition diagram the states of a process are pictured as open dots (∘): the *nodes* of a process graph. Any action $a \in A$ the process can perform is regarded as a state transition from the state of the process before performance, to the state after. Such a state transition is pictured as an arrow between these two states, labeled by a: an *edge* of the process graph. If a process passes to another state, without performing an (observable) action, the corresponding state transition in the diagram is labeled by $\tau \notin A$ (the *invisible action*, or *τ-step*). If a process can remain in a state without terminating, then there is a τ-step from this state to itself (a *τ-loop* or *delay*). Finally the initial state in the diagram is denoted by a short arrow (→∘): the *root* of the process graph. Now in the graph model of CCS a process is represented by its state transition diagram, and any state transition diagram over $A \cup \{\tau\}$ represents a process. However, different state transition diagrams may represent the same process: two processes are identified if there exists a *bisimulation* between their state transition diagrams g and h. This is a binary relation R between the states of g and h, containing the pair of roots, such that if $(s,t) \in R$ and $s \xrightarrow{a} s'$ is an edge in g then there is an edge $t \xrightarrow{a} t'$ in h, with the same label $a \in A \cup \{\tau\}$, such that $(s',t') \in R$, and, vice versa, if $(s,t) \in R$ and $t \xrightarrow{a} t'$ is an edge in h, then there is an edge $s \xrightarrow{a} s'$ in g with $(s',t') \in R$.
This identification criterion is what Milner calls **strong congruence** (although his first definition of strong congruence (in [11]) was slightly different). In [11] Milner expresses the wish to identify also processes which are not strongly congruent. Then a process is modeled as an equivalence class of state transition diagrams, under an equivalence relation containing strong congruence. The appropriate equivalences are discussed later.

3.6 Operational semantics

In a calculus processes are represented by process expressions, built from the constants and operators in the language. This representation differs from the model representations in two ways: different expressions may represent the same process, and some processes may have no process expression representation.
The *initial algebra* of a theory is the set of closed process expressions modulo provable equality (if the theory is an algebraical calculus, then provable equality is always a congruence). If the language used is sufficiently expressive and the calculus complete for closed terms (with respect to an intended interpretation), then the initial algebra models the finite processes, i.e. any finite process is represented by exactly one equivalence class of process expressions. In CCS and CSP also a recursion operator μ is present, enableing the construction of process expressions representing infinite processes. In the presense of such an operator the idea of the initial algebra can be generalised, and the set of closed process expressions modulo provable equality again constitutes a model of concurrency. However, in the absence of a complete calculus (with respect to an intended interpretation), this model does not make enough identifications.
This asks for a coarser equivalence on process expressions. Such an equivalence can be obtained by the general method of endowing languages with operational semantics: For any label

$a \in A \cup \{\tau\}$ define a binary relation $\xrightarrow{a}$ on the set of process expressions, in such a way that $E \xrightarrow{a} F$ means that the process represented by the expression E may perform an a-step, thereby changing into a process that can be represented by the expression F. This makes the domain of process expressions into a state transition diagram (however without a root). From this universal state transition diagram the diagram of a particular expression can be obtained by appointing this expression as root of the diagram.
Since this approach identifies process expressions, processes and states, a bisimulation can be defined as a relation on process expressions, and strong congruence is just the union of all bisimulations. Now a model of closed CCS-expressions modulo strong congruence can be constructed, which is more satisfying than the generalized initial algebra approach of CCS-expressions modulo provable equality. In §6 an operational semantics for both CCS and CSP will be presented.

4. When to identify processes

4.1 Why identify processes?

As remarked in §1, one of the purposes of a concurrency theory is to verify statements about processes. Such a statement can be that a certain system correctly simulates a specified process. In that case the theory has to determine whether the two processes (i.e. the real and the intended behaviour of the system) are equal. This asks for a criterion for identifying processes. Such a criterion determines (partly) the semantics of the theory. The choice of a suitable semantics may depend on the tools an environment has, to distinguish between certain processes. It is conceivable that a concurrency theory is equipped with different semantics, and has the capacity to express equality on different levels.

4.2 How to identify processes

In the various concurrency theories different identification strategies have been pursued. In particular CCS identifies much less then CSP. An advantage of identifying more is that it becomes easier to verify statements in which processes are equated. All true statements x=y remain true after identifying more. However, one might identify too much, depending on the discriminating capacity of an environment. In particular the identification of two processes that cannot be distinguished with a given set of tools, disables the development of a new tool to distinguish them. Algebraically this means that some operators in a language of concurrency (which correspond to tools that distinguish between processes) are incompatible with some identifications. Moreover some useful conditionals axioms might get lost, because after making certain identifications the premisses of the axioms are true too often.

4.3 Bisimulation semantics

In CCS processes are identified only if there is no environment conceivable in which they can be distinguished. In each case processes should be identified if they are strongly congruent, in the sense of section 3.5. However, Milner identifies some more processes, only differing in their invisible steps. In [11] he proposes the notion of **observation equivalence**. Later he uses a slightly different version of observation equivalence (see [12]), adapted to the notion of bisimulation, as proposed by Park in [15]. In §6 the syntax of CCS is presented, together with an operational semantics, including the definition of observation equivalence. The basic operators of CCS are the constant 0, the unary operators a (for $a \in A$) and the binary operator +. 0 represents the process, unable of doing anything at all; aP represents the process, which will first perform an a-step and then proceeds with P; and P + Q represents the process, which first makes a choice between P and Q, and then proceeds with the chosen process. This is illustrated by the following state transition diagrams:

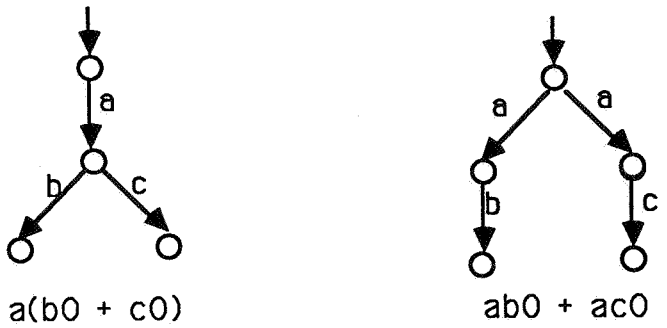

fig. 1

In bisimulation semantics the processes a(b0 + c0) and ab0 + ac0 are considered to be different. A motivation for this can be found in the different timing of the choice between b and c. Moreover, if they are placed in an environment that will not allow the execution of c, then they can be distinguished by observation: ab0 + ac0 has the possibility to deadlock after execution of a, while a(b0 + c0) has not: here a will always be followed by b.
Algebraically, such an environment is represented by the restriction operator \c (see §6; on process graphs \c removes all c-edges (as well as the disconnected parts that originate)). Now (ab0 + ac0)\c = ab0 + a0, while (a(b0 + c0))\c = ab0.
In [11], Milner remarks about observation equivalence that "two behaviour expressions should have the same interpretation in the model iff in all contexts they are indistinguishable by observation". However, he gives no clue, how one goes about distinguishing between abc0 + abd0 and a(bc0 + bd0) by observation (or it must be that the states of a process are considered to be directly observable). In any case, it cannot be done by any of the CCS operators.

4.4 Trace semantics

In trace theory much more processes are identified than in CCS. By defining trace equivalence on process graphs, it is possible to compare trace semantics with bisimulation semantics. Definition: $\sigma \in A^*$ is a *trace* of a process graph g, if there is a finite path in g, starting at the root, with label σ. Here the label of a path is the sequence of labels of the composing transitions, where all τ-labels are dropped. Now the *trace set* of g is the set of its traces, and two process graphs are **trace equivalent** iff they have the same trace sets. Remark that the model of trace sets of section 3.3 is isomorphic to the model of process graphs modulo trace equivalence. Since any two observation equivalent processes are also trace equivalent, trace equivalence is called a coarser (= less discriminating) equivalence than observation equivalence. This is pictured in section 4.7.
In the setting of trace theory, presented in section 3.3 (or above), no deadlock behavior is displayed. Not only the processes $ab0 + ac0$ and $a(b0 + c0)$ are identified (both have trace set $\{\epsilon, a, ab, ac\}$) but also the processes $ab0 + a0$ and $ab0$ (both have trace set $\{\epsilon, a, ab\}$). However, $ab0 + a0$ can deadlock after performing an a-step, while $ab0$ cannot. If deadlock is considered to be observable, a modification of trace theory can be made, in which also traces ending on 0 are allowed. In that setting $ab0 + a0$ has trace set $\{\epsilon, a, a0, ab, ab0\}$ while $ab0$ has trace set $\{\epsilon, a, ab, ab0\}$. Call the corresponding equivalence on graphs **∂-trace equivalence.**

4.5 Failure semantics

In ∂-trace semantics, where deadlock is observable, the processes $ab0 + ac0$ and $a(b0+c0)$ are equal (both have ∂-trace set $\{\epsilon, a, ab, ab0, ac, ac0\}$). However, the processes $(ab0 + ac0)\backslash c = ab0 + a0$ and $(a(b0 + c0))\backslash c = ab0$ are different (as remarked above). So Milner's restriction operators $\backslash c$ $(c \in A)$ are incompatible with ∂-trace semantics. If an environment is equipped with restriction as a tool for analysing processes, then a finer equivalence is needed to model the results of this analysis. As suggested previously, in section 3.4, failure semantics is adequate for restriction and deadlock behavior.
A tuple $\langle \sigma, X \rangle$ with $\sigma \in A^*$ and $X \subseteq A$ is a *failure pair* of a process graph g if there is a path from the root of g to a node p with label σ, such that the set $I(p)$ of labels of the outgoing edges of p, is contained in $A-X$, i.e. if the process can deadlock after execution of σ, in case the environment allows only actions from X. Two processes, not containing divergence (= infinite τ-paths) are **failure equivalent** iff they have the same set of failure pairs. Now the model of process graphs (not containing divergence) modulo failure equivalence is isomorphic to the model of failure sets of section 3.4.
A node p of a process graph is said to be *unstable* if it has an outgoing τ-edge. Remark that because $\tau \notin A$, a path ending in an unstable node cannot contribute to the failure set of a process. This is on purpose, since deadlock can never occur if a τ-step is possible. A consequence of this is that in the presence of divergence some information on the trace set of a process might get lost (in the construction of a failure set). A process containing a τ-loop at every node, for instance, has no failure pairs! This is the reason for excluding diverging processes. They will be added however in

section 5.5.
A variant of failure semantics is *readiness semantics*, as presented in Olderog & Hoare [14]. $\langle\sigma,X\rangle \in A^* \times Pow(A)$ is a *ready pair* of a process graph g, if there is a path from the root of g to a node p with $I(p) = X$. **Ready equivalence** must be a finer equivalence than failure equivalence since the failure set of a process is derivable from its ready set. The reverse however is not true: $ab0 + ac0$ and $ab0 + a(b0 + c0) + ac0$ are failure equivalent, but not ready equivalent.

4.6 Ready trace semantics

By now one might think that failure equivalence constitutes a preferable semantics for models of concurrency, since two processes are failure equivalent iff they are distinguishable by observation. However this depends to a great extent on the tools an environment has, to analyse processes. If these tools are unknown, then bisimulation semantics is in each case a safe choice. Therefore also in ACP and the topological process theory bisimulation semantics is used. In [2], Baeten, Bergstra & Klop show that a priority operator (as they introduced in the context of bisimulation semantics in [1]) is incompatible with failure semantics. Such an operator models an environment, which imposes a priority to the execution of certain atomic actions over others, and can be used for the specification of an interrupt mechanism. Moreover they present a semantics intermediate between readiness and bisimulation semantics (but without τ-steps) that is compatible with priorities. In this **ready trace** semantics the role of a ready pair is replaced by an alternating sequence of subsets and elements of A, representing a trace of a process, with for each node on the trace the set of possibilities to continue.

4.7 Survey

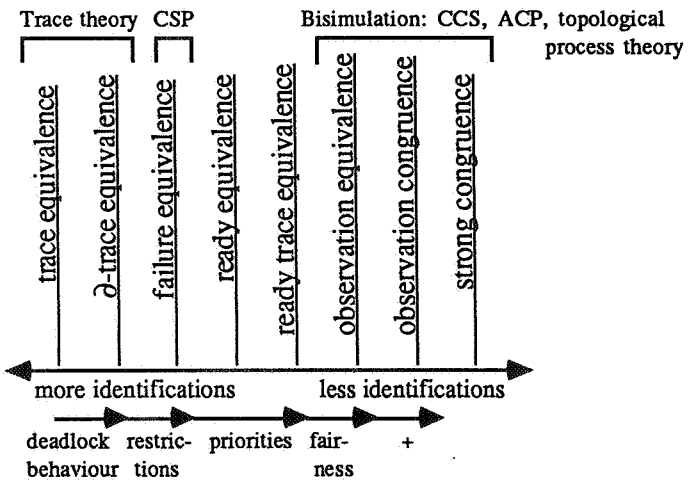

fig. 2

In figure 2, the equivalences mentioned above are classified. In the bottom line, the reasons are displayed to move into the direction of less identifications. Observation congruence will be discussed in section 5.1. If the schema suggests that all interesting equivalences can be linearly ordered by inclusion, then this is misleading; in order to keep the picture simple all equivalences disturbing the linearity are omitted. Furthermore bisimulation semantics identifies strictly less then failure semantics only in the absence of divergence. The differences between the various equivalences are further clarified by the axiomatisations in section 6.6. In figure 3, six process graphs are displayed, in such a way that in order to distinguish a process from the previous one, each time a finer equivalence is needed. This illustrates that in bisimulation semantics all information about the timing of choices is preserved, in trace semantics none, and in the other semantics some.

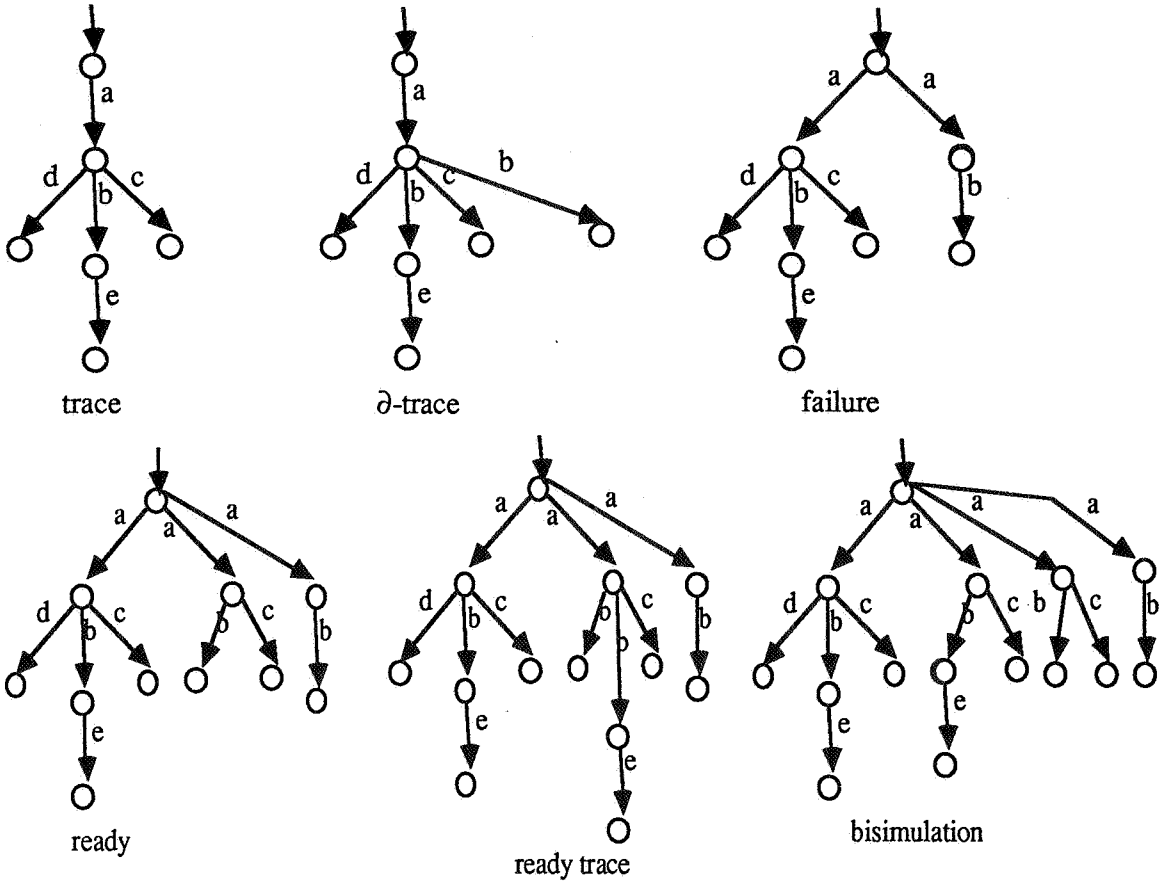

fig. 3

5. Features of concurrency

Both CCS and CSP capture nondeterminism, communication, recursion, abstraction, divergence and deadlock behaviour, but differently. A discussion per feature will follow below. In §6 the operators (the most important ones anyway) of CCS and CSP are presented, and provided with an operational semantics (as explained in section 3.6). For CSP this is not the usual method, but the failure semantics of both CCS and CSP are derivable by translating graphs to failure sets.

5.1 Nondeterminism

Both of the languages of CCS and CSP are equipped with a constant 0 for deadlock (called NIL in CCS and STOP in CSP) and with prefix multiplication aP (in CSP denoted as $a \rightarrow P$) for representing the sequential composition of a and P. However they have different operators for choice. Hoare uses two operators for choice: *external choice* $\Box$ and *internal choice* $\sqcap$. The first kind of choice is deterministic: it depends on the environment; the second is nondeterministic: it cannot be influenced by the environment. A nondeterministic choice appears after abstraction from the actions of the environment that cause the choice for one of the alternatives. Both $\Box$ and $\sqcap$ are commutative, associative and idempotent (see the table of CSP axioms in section 6.4), i.e. the alternatives can be regarded to form a set. The difference appears in combination with deadlock: $P \Box 0 = P$ but $P \sqcap 0 \neq P$! Now the influence of the environment can be modeled by Milner's restriction operator: $a0 \Box b0 \backslash b = a0$, while $a0 \sqcap b0 \backslash b = a0 \sqcap 0$ (for $a \neq b$). So the environment cannot force the process $a0 \sqcap b0$ to choose its left summand.

Milner makes no distinction between external and internal choice; there is only one choice operator, +, and apart from being commutative, associative and idempotent, it satisfies $P + 0 = P$. On synchronisation trees, + composes two processes by identifying their roots. In CCS, nondeterminism is not a property of the operator, making an alternative composition of two processes P and Q, but of the alternatives P and Q together. A choice can be regarded as fully nondeterministic if the environment does not participate in the selection of the alternatives. This can be modeled with the unary operator τ. A nondeterministic choice between P and Q can now be represented by $\tau P + \tau Q$ (so $P \sqcap Q = \tau P + \tau Q$) and a deterministic choice between, say, a0 and b0 is represented by $a0 + b0$. On the other hand, the process $\tau a0 + b0$ can be represented in CSP by $a0 \sqcap (a0 \Box b0)$.

If one tries to translate the CSP operator $\Box$ into CCS (as is done for $\sqcap$ above), one might think that it is just +. However, this is not the case. If P and Q are starting with a τ-step, then their +-composition yields a nondeterministic choice, while the operator $\Box$ intends to remove this nondeterminism: $\tau a0 \Box \tau b0 = \tau(a0 + b0) \neq \tau a0 + \tau b0$! Therefore it is not possible to translate $\Box$ into CCS directly. However it can be axiomatised over +, $\cdot$ and 0, as was shown by Brookes in [8], see section 6.5.

In CCS the processes a0 and $\tau a0$ are observation equivalent. However the processes $a0 + b0$ and $\tau a0 + b0$ are not; they are not even failure equivalent: $(a0 + b0) \backslash a = b0$ and $(\tau a0 + b0) \backslash a = \tau 0 + b0$ have a different deadlock behavior. So in the presence of the +-operator, observation equivalence cannot be a criterion for identification; once a0 and $\tau a0$ are identified, $a0 + b0$ and $\tau a0 + b0$ cannot be distinguished. Summation is incompatible with observational equivalence in the same way as restriction is incompatible with ∂-trace semantics. For that reason Milner introduced (in [11]) the notion of **observation congruence**: two processes are observation congruent if they are observation equivalent in every context. This does give a suitable identification criterion (see also figure 2). Now any observation equivalence class contains exactly two observation congruence classes (P and τP). In the same way **failure congruence** can be defined (congruence with respect to +), but in CSP this is not necessary, since + is not a

CSP operator and failure equivalence is already a congruence for the CSP operators (as is observation equivalence, see Brookes [8]).

5.2 Communication

In their treatment of communication, there are three differences between CSP and CCS:
- CSP has different operators for communication and interleaving ($\|$ and $|||$), while CCS has one operator ($|$) doing both.
- In CSP communication between two processes occurs if both of them offer the same action $a \in A$. In CCS this happens if one of them offers an atomic action $a \in A$, and the other its complementary action $\bar{a}$.
- In CSP the communication between a and a results in the same step a. In CCS the communication between a and $\bar{a}$ results in a τ-step, i.e. the communication serves only as synchronisation, the result is not visible.

These differences are illustrated by the following examples:

$(a0 \Box b0) ||| a0 = aa0 \Box ba0 \Box a(a0 \Box b0)$

$(a0 \Box b0) \| a0 = a0$

$(a0 + b0) | \bar{a}0 = a\bar{a}0 + b\bar{a}0 + \bar{a}(a0 + b0) + \tau 0.$

In CCS there is a restriction operator $\backslash a$, to remove the results of unsuccessful communication, i.e. to remove some of the interleaving component of paralel composition:

$\{(a0 + b0) | \bar{a}0\}\backslash a \backslash b = 0+0+0+\tau 0 = \tau 0.$

In CSP such an operator is not present, but it is expressible using $\|$, if the alphabet A is finite. Suppose that $A = \{a,b,c\}$ then $x\backslash a = x \| \mu X.(bX \Box cX)$. In this translation of $\backslash a$, the actions $A-\{a\}$, allowed by the environment are used, instead of the disallowed action a. $\mu X.(bX \Box cX)$ is the unique solution of the equation $X = bX \Box cX$, i.e. the infinite sequence of choices between b and c.

The exact meaning of these operators is given by the operational semantics of CCS and CSP in the sections 6.1 and 6.2. The algebraic laws governing them are listed in the sections 6.4 and 6.5. In the listing of CCS axioms also the axioms of CSP operators in CCS context are presented, as in Brookes [8].

5.3 Recursion

Both in CSP and CCS it is possible to specify a process by means of a *fixed point equation.* Such an equation has the form $X = P$ with $P \in E$ a process expression and X a variable. The process $aaaa...$, performing an infinite sequence of a-steps, for instance, is specified by the fixed point equation $X = aX$. Some fixed point equations, like $X = aX$, have unique solutions (in the

mentioned failure and graph models) but others have more solutions (any process satisfies $X = X$); however there are no fixed point equations without solutions. Both CSP and CCS use the expression $\mu X.P$ to denote the unique solution of $X=P$, if there is one. If $X = P$ has no unique solution, then $\mu X.P$ should denote some default element from the solution set. The question which one is answered differently in CSP and CCS.

In the failure model of CSP the inclusion ordering on failure sets makes this model into a complete lattice. On process expressions this ordering is characterised by the condition $X \sqsubseteq Y$ iff $X \sqcap Y = Y$ (Y is *less deterministic* than X). Now all CSP operators turn out to be monotonic for this ordering (i.e. $X \sqsubseteq X'$ implies $f(X,Y) \sqsubseteq f(X',Y)$), and using general fixed point theory this implies that any fixed point equation has a minimal and a maximal solution. Hoare chooses the maximal fixed point to be the interpretation of $\mu X.P$. His reason for doing so is that underspecification expresses uncertainty about the specified process. Therefore the default solution of the equation should be the least deterministic one (the least predictable). In the most extreme case (of the fixed point equation $X=X$) there is complete underspecification and no certainty at all. Therefore $\mu X.X$ is chosen to be the least deterministic of all processes: the process CHAOS. The failure set of CHAOS is $A^* \times Pow(A)$. CHAOS can be regarded as the internal sum ($\sqcap$) of all processes. In the calculus of CSP, CHAOS can be added as a new constant χ, satisfying the law $\chi \sqcap x = \chi$.

In CCS this method cannot be applied, since prefixing is not monotonic for the CSP-ordering (due to the absence of an axiom $aX \sqcap aY = a(X \sqcap Y)$), and no complete partial order can be found for which the CCS operators are continuous. However a fixed point is found in the graph, generated by the action rules for the operational semantics of section 6.1. That this graph really satisfies its fixed point equation (the recursion axiom in section 6.5) follows trivially from the action rule for recursion in section 6.1. Remark that Milner's fixed point is another one than Hoare's: in CCS $\mu X.X = 0$, while in CSP $\mu X.X = \chi$. Milner's fixed point is failure equivalent to the least fixed point existing in CSP, while in CSP the largest fixed point is chosen.

Also sets of fixed point equations can be used to specify processes. A *recursive specification* E is a set $\{X = P_X \mid X \in \Xi\}$ with Ξ a set of variables and P_X a process expression (for $X \in \Xi$). Example: if $E = \{X = aY, Y = bX\}$, then $X = ababab...$ and $Y = bababa....$ The X-component of the solution vector of E is denoted by $\langle X \mid E \rangle$. Thus, $\langle X \mid E \rangle$ means: 'the X, as specified by E'. This is a safer expression than just X, since the variable X can also occur in other specifications. However, in most contexts the names of the variables in all mentioned specifications are chosen to be distinct, so that $\langle X \mid E \rangle$ can safely be abbreviated by X.

If E is finite then the expression $\langle X \mid E \rangle$ can be translated into a CCS or CSP expression, involving the nested use of the recursion operator μ. Example: $\langle X \mid X = aX + bY, Y = cX + dY \rangle = \mu X.(aX + b\mu Y.(cX + dY))$.

5.4 Abstraction

In CSP there is a concealment operator $/a$ (for $a \in A$) for hiding those actions we are not interested in. As in Brookes [8] and De Nicola [13] the notation $/a$ is used instead of $\backslash a$, in order

to distinguish abstraction from restriction. Its operational behaviour and the axioms governing it can be found in the sections 6.2 and 6.4. The application of such an operator is called 'abstraction from internal steps'. There is a big difference between abstraction and restriction: $abc0/b = ac0$, while $abc0\backslash b = a0$.
In CCS there is no separate concealment operator, since there abstraction and communication are integrated. However, hiding can be expressed by the operators for parallel composition and restriction: $x/a = (x|\mu Y.aY+\bar{a}Y)\backslash a$, where $\mu Y.aY+\bar{a}Y$ is the process, only generating a- and $\bar{a}$-steps. The translation of concealment into CCS can be axiomatised by the axioms in section 6.5. Using the CSP axioms for concealment, one finds (if $a \neq b$): $(ac0 \Box bd0)/a/b = c0 \sqcap d0 \sqcap (c0 \Box d0)$, and using the CCS axioms: $(ac0 \Box bd0)/a/b = (ac0 + bd0)/a/b = \tau c0 + \tau d0 = c0 \sqcap d0$. This is indeed the same result, since in failure semantics $x \sqcap y = x \sqcap y \sqcap (x \Box y)$, as can be verified by either using the distributive laws of section 6.4, or the failure axioms of section 6.6.

5.5 Divergence

On process graphs, abstraction from an atomic action a consists of replacing all a-edges by τ-edges. This might result in divergence (infinite τ-paths) as is the case in $(\mu X.aX)/a$. Here an infinite a-path changes into an infinite τ-path. Contrary to the equation $X = aX$, that has the infinite a-path as unique solution, the CCS equation $X=\tau X$ is satisfied by many processes, of which $\tau 0$ is the simplest. However, the process that is selected to be the default solution of $X=\tau X$ (by Milner's operational semantics of CCS) is just the infinite τ-path. Hence $(\mu X.aX)/a = \mu X.\tau X$, and in general $(\mu X.P)/a = \mu X.(P/a)$, so abstraction and recursion commute.
In CSP the situation is different in two respects: first the expression $\mu X.P$ has another meaning than in CCS and second, by the absence of τ, it is not possible to define a divergent process directly (in section 4.5, divergent processes were even excluded from the failure model). In general X/a is the process X from which the a-steps are removed ($aab0/a = b0$ in CSP). Thus, the actions 'behind' the a-steps are moved forwards. But since it is not clear, what can be thought to be behind an infinite sequence of a-steps, Hoare has some freedom in giving a meaning to $(\mu X.aX)/a$ in the failure model. He chooses to treat 'overabstraction' like underspecification, and the result is that also in CSP recursion and abstraction do commute, so $(\mu X.aX)/a = \mu X.((aX)/a) = \mu X.X = \chi$.
In combination with the interpretation of abstraction on process graphs, this implies that any divergent process (= a process containing divergence at the root) is failure equivalent with the process χ. This removes the restriction on failure equivalence, that it is only defined on processes not containing divergence. However, by doing so, a lot of interesting information about divergent processes gets lost: even the processes

are identified! This is the reason a different form of failure semantics is presented in Bergstra, Klop

& Olderog [6], in which divergence is treated more subtly.

5.6 Deadlock behaviour

Deadlock is the state of a process where no further action is possible. It can occur in a merge of two processes if both of them are waiting for the other to provide a suitable communication. Example (in CCS) :

$$[\{\mu X.a(acX+bdX)\}|(\mu Y.\bar{a}\bar{b}\bar{a}Y)]\backslash a\backslash b = \tau\tau d\tau\tau c0.$$

As explained in §4, deadlock behaviour is preserved by ∂-trace, failure, ready and ready trace equivalence, but not by trace equivalence. Furthermore ∂-trace equivalence is disqualified since it is disturbed in the presence of communication and restriction operators, modeling the influence of the environment. In the absence of divergence deadlock behaviour is preserved in bisimulation semantics too, but in the presence of divergence it is preserved only in combination with livelock behaviour.
Livelock is the state of a process where only an infinite sequence of hidden moves is possible, as in $\mu X.\tau X$. In CSP livelock (being a special case of divergence) is equated with the fully unpredictable process CHAOS. In CCS it is equated with deadlock: $\mu X.\tau X = \tau 0$.
Deadlock can be visualised if processes are supposed to make noise. The noise starts at the beginning of a process and ends if the process reaches a state of deadlock. If a component in a merge has to wait for a suitable communication it becomes silent untill the communication is enabled, but as long as at least one component is making progress, noise is being made. Only if all components are waiting, the process becomes silent. This guarantees that no further action is possible.
In this interpretation deadlock can be distinguished from livelock. Of course it is also possible to define a version of bisimulation where deadlock and livelock behaviour are distinguished.

6. Survey of CCS and CSP

6.1 An operational semantics for CCS

Let Δ be a given set of names. $\Delta^- = \{\bar{a} \mid a \in \Delta\}$ is the corresponding set of *conames*. $\Delta \cap \Delta^- = \emptyset$. Let $\tau \notin A = \Delta \cup \Delta^-$ be the *invisible step* and write $A_\tau = \Delta \cup \Delta^- \cup \{\tau\}$. Let a,b,c range over A_τ and put $\bar{\bar{a}} = a$ $(\bar{\tau} = \tau)$. A function $R: \Delta \to \Delta$ is called a *relabeling*. The domain of a relabeling R can be expanded to A_τ by putting $R(\tau) = \tau$ and $R(\bar{a}) = \overline{R(a)}$.
Let V be a given set of variables, then the set E of CCS expressions is defined inductively by:

VARIABLES:	$V \subseteq E$
ACTION:	If $P \in E$ and $a \in A_\tau$ then $aP \in E$
INACTION:	$0 \in E$
CHOICE:	If $P,Q \in E$ then $P+Q \in E$
COMPOSITION:	If $P,Q \in E$ then $P \mid Q \in E$
RESTRICTION:	If $P \in E$ and $a \in A$ then $P \backslash a \in E$
RELABELING:	If $P \in E$ and $R: \Delta \to \Delta$ then $P[R] \in E$
RECURSION:	If $X \in V$ and $P \in E$ then $\mu X.P \in E$

Now the *action relations* $\xrightarrow{a} \subseteq E \times E$ for $a \in A_\tau$ are generated by the following rules:

- $aP \xrightarrow{a} P$
- From $P \xrightarrow{a} Q$ infer:
  - $P+S \xrightarrow{a} Q$
  - $S+P \xrightarrow{a} Q$
  - $P \mid S \xrightarrow{a} Q \mid S$
  - $S \mid P \xrightarrow{a} S \mid Q$
  - $P[R] \xrightarrow{R(a)} Q[R]$
  - and if $\bar{a} \neq b \neq a$: $P \backslash b \xrightarrow{a} Q \backslash b$
- From $P \xrightarrow{a} Q$ and $S \xrightarrow{\bar{a}} T$ $(a \neq \tau)$ infer $P \mid S \xrightarrow{\tau} Q \mid T$
- From $P[X:=\mu X.P] \xrightarrow{a} Q$ infer $\mu X.P \xrightarrow{a} Q$.

(Here $P[X:=S]$ denotes the result of substituting S for each free occurrence of X in P, with usual avoidance of name clashes.)

6.2 An operational semantics for CSP

Let A be a given alphabet of atomic actions and let V be a given set of variables, then the set E of CSP expressions is defined inductively by:

VARIABLES:	$V \subseteq E$
ACTION:	If $P \in E$ and $a \in A$ then $aP \in E$
INACTION:	$0 \in E$
EXTERNAL CHOICE:	If $P,Q \in E$ then $P \Box Q \in E$
INTERNAL CHOICE:	If $P,Q \in E$ then $P \sqcap Q \in E$
COMMUNICATION:	If $P,Q \in E$ then $P \| Q \in E$
INTERLEAVING:	If $P,Q \in E$ then $P \|\| Q \in E$
CONCEALMENT:	If $P \in E$ and $a \in A$ then $P/a \in E$
RELABELING:	If $P \in E$ and $f: A \to A$ is injective then $f(P) \in E$
RECURSION:	If $X \in V$ and $P \in E$ then $\mu X.P \in E$.

Now the action relations $\xrightarrow{a} \subseteq E \times E$ for $a \in A_\tau$ are generated by the following rules:

- $aP \xrightarrow{a} P$
- $P \sqcap Q \xrightarrow{\tau} P$
- $P \sqcap Q \xrightarrow{\tau} Q$
- From $P \xrightarrow{a} Q$ $(a \neq \tau)$ infer: $P \Box S \xrightarrow{a} Q$
  $S \Box P \xrightarrow{a} Q$
  $f(P) \xrightarrow{f(a)} f(Q)$
- From $P \xrightarrow{\tau} Q$ infer: $P \Box S \xrightarrow{\tau} Q \Box S$
  $S \Box P \xrightarrow{\tau} S \Box Q$
  $f(P) \xrightarrow{\tau} f(Q)$
  $P \| S \xrightarrow{\tau} Q \| S$
  $S \| P \xrightarrow{\tau} S \| Q$
- From $P \xrightarrow{a} Q$ and $S \xrightarrow{a} T$ $(a \neq \tau)$ infer: $P \| S \xrightarrow{a} Q \| T$
- From $P \xrightarrow{a} Q$ infer: $P \|| S \xrightarrow{a} Q \|| S$
  $S \|| P \xrightarrow{a} S \|| Q$
  $P/a \xrightarrow{\tau} Q/a$
  and if $a \neq b$: $P/b \xrightarrow{a} Q/b$
- From $P[X:=\mu X.P] \xrightarrow{a} Q$ infer $\mu X.P \xrightarrow{a} Q$.

6.3 Equivalences on process expressions

Let $\xrightarrow{\sigma} \subseteq E \times E$ for $\sigma \in A^*$, the set of finite words over A, be the least relation satisfying:
- $P \xrightarrow{\varepsilon} P$
- If $P \xrightarrow{a} Q$ then $P \xrightarrow{a} Q$ (for $a \in A$)
- If $P \xrightarrow{\tau} Q$ then $P \xrightarrow{\varepsilon} Q$
- If $P \xrightarrow{\sigma} Q$ and $Q \xrightarrow{\rho} S$ then $P \xrightarrow{\sigma * \rho} S$.

A *τ-bisimulation* is a relation $R \subseteq E \times E$, satisfying (for all $\sigma \in A^*$)
- If $P R Q$ and $P \xrightarrow{\sigma} P'$ then $Q \xrightarrow{\sigma} Q'$ and $P' R Q'$ for some $Q' \in E$.
- If $P R Q$ and $Q \xrightarrow{\sigma} Q'$ then $P \xrightarrow{\sigma} P'$ and $P' R Q'$ for some $P' \in E$.

$\sigma \in A^*$ is a *trace* of P if $P \xrightarrow{\sigma} Q$ for some $Q \in E$.
P is *divergent* if there is an infinite τ-path $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} P_2 \xrightarrow{\tau} \ldots$.
$\langle \sigma, X \rangle \in A^* \times Pow(A)$ is a *failure pair* of P if $P \xrightarrow{\sigma} Q$ for some $Q \in E$ such that $Q \xrightarrow{a} S$ implies $a \in A - X$, or if $\sigma = \sigma' * \sigma''$ and $P \xrightarrow{\sigma'} Q$ for some divergent $Q \in E$.

P and Q are *observation equivalent* ($P \approx Q$) if $P R Q$ for some τ-bisimulation *R*.
P and Q are *observation congruent* if $P+S \approx Q+S$ for any $S \in E$.
P and Q are *failure equivalent* ($P \equiv Q$) if they have the same failure sets.
P and Q are *failure congruent* if $P+S \equiv Q+S$ for any $S \in E$.
P and Q are *trace equivalent* if they have the same trace set.

6.4 Axioms for CSP (and failure equivalence)

External choice:

$$x \Box y = y \Box x$$
$$x \Box (y \Box z) = (x \Box y) \Box z$$
$$x \Box x = x$$
$$x \Box 0 = x$$

Internal choice:

$$x \sqcap y = y \sqcap x$$
$$x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$$
$$x \sqcap x = x$$

Distributive laws:

$$x \Box (y \sqcap z) = (x \Box y) \sqcap (x \Box z)$$
$$x \sqcap (y \Box z) = (x \sqcap y) \Box (x \sqcap z)$$
$$ax \sqcap ay = a(x \sqcap y)$$
$$ax \Box ay = a(x \sqcap y)$$

Communication:*

$$x \| y = y \| x$$
$$(x \sqcap y) \| z = (x \| z) \sqcap (y \| z)$$
$$P \| Q = \Box_{a_i = b_j} a_i (P_i \| Q_j)$$

Interleaving:*

$$x ||| y = y ||| x$$
$$(x \sqcap y) ||| z = x ||| z \sqcap y ||| z$$
$$P ||| Q = \Box_i a_i (P_i ||| Q) \Box \Box_j b_j (P ||| Q_j)$$

Concealment:

$$(x \sqcap y)/a = x/a \sqcap y/a$$
$$(ax \Box y)/a = x/a \sqcap (x \Box y)/a$$
$$(\Box_i b_i P_i)/a = \Box_i b_i (P_i/a) \text{ if } \forall i\ b_i \neq a$$

Relabeling:

$$f(0) = 0$$
$$f(x \sqcap y) = f(x) \sqcap f(y)$$
$$f(x \Box y) = f(x) \Box f(y)$$
$$f(ax) = f(a) f(x)$$

Recursion:

$$\mu X.P = P[X := \mu X.P]$$

---

*: Here $P = a_1 P_1 \Box a_2 P_2 \Box \ldots \Box a_n P_n = \Box_{i=1}^{n} a_i P_i$ and $Q = \Box_{j=1}^{m} b_j Q_j$. Put $\Box_{i=1}^{0} a_i P_i = 0$.

6.5 Axioms for CCS (and strong congruence)

Choice:

$$x + y = y + x$$
$$(x + y) + z = x + (y + z)$$
$$x + x = x$$
$$x + 0 = x$$

Restriction:

$$0\backslash a = 0$$
$$(x+y)\backslash a = x\backslash a + y\backslash a$$
$$(ax)\backslash a = (\bar{a}x)\backslash a = 0$$
$$(bx)\backslash a = b(x\backslash a) \text{ if } \bar{a}\neq b\neq a$$

Relabeling:

$$0[R] = 0$$
$$(x+y)[R] = x[R] + y[R]$$
$$(ax)[R] = R(a)\cdot(x[R])$$

Recursion:

$$\mu X.P = P[X:=\mu X.P]$$

Composition:[1)]

$$P|Q = \sum a_i(P_i|Q) + \sum b_j(P|Q_j) + \sum_{b_j=\bar{a}_i\neq\tau} \tau(P_i|Q_j)$$

Interleaving:[1,2)]

$$P|||Q = \sum a_i(P_i|||Q) + \sum b_j(P|||Q_j)$$

Communication:[2,3)]

$$P\|Q = \sum \tau(P_i'\|Q) + \sum \tau(P\|Q_j') + \sum_{a_i=b_j} a_i(P_i'\|Q_j')$$

External choice:[2,3)]

$$P\,\square\,Q = \sum a_iP_i + \sum b_jQ_j + \sum \tau(P_i'\,\square\,Q) + \sum \tau(P\,\square\,Q_j')$$

Internal choice:[2)]

$$P\sqcap Q = \tau P + \tau Q$$

Concealment:[2)]

$$0/a = 0$$
$$(x+y)/a = x/a + y/a$$
$$(ax)/a = (\bar{a}x)/a = \tau(x/a)$$
$$(bx)/a = b(x/a) \text{ if } \bar{a}\neq b\neq a$$

---

1) Here $P = a_1P_1 + a_2P_2 + \ldots + a_nP_n = \sum_{i=1}^{n} a_iP_i$ and $Q = \sum_{j=1}^{m} b_jQ_j$. Put $\sum_{i=1}^{0} a_iP_i = 0$.

2) Imported from CSP.

3) Here $P = \sum_{i=1}^{n} a_iP_i + \sum_{i=1}^{N} \tau P_i'$ and $Q = \sum_{j=1}^{m} b_jQ_j + \sum_{j=1}^{M} \tau Q_j'$ with $a_i, b_j \neq \tau$.

6.6 Axioms for identification of CCS expressions

Observational congruence:

$$a\tau x = ax$$
$$\tau x + x = \tau x$$
$$a(\tau x + y) = a(\tau x + y) + ax$$

Ready congruence:

$$a(\tau x + \tau y) = ax + ay$$
$$\tau x + x = \tau x$$
$$\tau(\tau x + y) = \tau x + y$$
$$\tau(ax + ay + z) = \tau(ax + z) + ay$$

Failure congruence:

$$a(\tau x + \tau y) = ax + ay$$
$$\tau x + y = \tau(x+ y) + \tau x$$
$$\tau(ax + ay + z) = \tau(ax + z) + ay$$

Trace equivalence:

$$ax + ay = a(x + y)$$
$$\tau x = x$$

7. References

[1] Baeten, J.C.M., J.A. Bergstra & J.W. Klop, *Syntax and defining equations for an interrupt mechanism in process algebra,* Fundamenta Informaticae IX, pp. 127-168, 1986.

[2] Baeten, J.C.M., J.A. Bergstra & J.W. Klop, *Ready trace semantics for concrete process algebra with priority operator,* report CS-R8517, Centrum voor Wiskunde en Informatica, Amsterdam 1985.

[3] De Bakker, J.W. & J.I. Zucker, *Denotational semantics of concurrency,* Proc. 14th ACM Symp. on Theory of Computing, pp. 153-158, 1982.

[4] De Bakker, J.W. & J.I. Zucker, *Processes and the denotational semantics of concurrency,* Information & Control 54 (1/2), pp. 70-120, 1982.

[5] Bergstra, J.A. & J.W. Klop, *Algebra of communicating processes,* Proc. of the CWI Symp. Math. & Comp. Sci., eds. J.W. de Bakker, M. Hazewinkel & J.K. Lenstra, pp. 89-138, North-Holland, Amsterdam 1986.

[6] Bergstra, J.A., J.W. Klop & E.-R. Olderog, *Failures without chaos: a new process semantics for fair abstraction,* Proc. of the Working Conference on the Formal Description of Programming Concepts, Ed. M. Wirsing, (Gl. Avernaes, August 1986), North-Holland, to appear.

[7] Brookes, S.D., C.A.R. Hoare & W. Roscoe, *A theory of communicating sequential processes,* J.Assoc.Comput.Mach. 31 (3), pp. 560-599, 1984.

[8] Brookes, S.D., *On the relationship of CCS and CSP,* Proc. 10th ICALP, Barcelona, ed. J. Diaz, Springer LNCS 154, pp. 83-96, 1983.

[9] Hoare, C.A.R., *Communicating sequential processes,* Prentice Hall International, 1985.

[10] Kleene, S.C., *Representation of events in nerve nets and finite automata,* Automata studies, pp. 3-41, Princeton Univ. Press, Princeton, 1956.

[11] Milner, R., *A calculus for communicating systems,* Springer LNCS 92, 1980.

[12] Milner, R., *Lectures on a calculus for communicating systems,* Seminar on concurrency, Springer LNCS 197, pp 197-220, 1985.

[13] De Nicola, R., *A complete set of axioms for a theory of communicating sequential processes,* Found. of Comp. Theory, Springer LNCS 158, pp. 115-126, 1983.

[14] Olderog, E.-R. & C.A.R. Hoare, *Specification-oriented semantics for communicating processes,* Proc. 10th ICALP, Barcelona, ed. J. Diaz, Springer LNCS 154, pp. 561-572, 1983.

[15] Park, D.M.R., *Concurrency and automata on infinite sequences,* Proc. 5th GI Conference, Springer LNCS 104, 1981.

[16] Reisig, W., *Petrinetze,* Springer-Verlag 1982.

[17] Rem, M., *Partially ordered computations, with applications to VLSI design,* Proc. 4th Advanced Course on Found. Comp. Sci., part 2, eds. J.W. de Bakker & J. van Leeuwen, Tract 159, Mathematisch Centrum, Amsterdam 1983.

[18] Salomaa, A., *Two complete axiom systems for the algebra of regular events,* J.Assoc.Comput.Mach. 13 (1), pp. 158-169, 1966.