Process Algebra

Rance Cleaveland
Department of Computer Science
P.O. Box 7534
North Carolina State University
Raleigh, NC 27695-7534
USA

Scott A. Smolka
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
USA

April 7, 1999

Abstract

Process algebra represents a mathematically rigorous framework for modeling concurrent systems of interacting processes. The process-algebraic approach relies on equational and inequational reasoning as the basis for analyzing the behavior of such systems. This chapter surveys some of the key results obtained in the area within the setting of a particular process-algebraic notation, the Calculus of Communicating Systems (CCS) of Milner. In particular, the Structural Operational Semantics approach to defining operational behavior of languages is illustrated via CCS, and several operational equivalences and refinement orderings are discussed. Mechanisms are presented for deducing that systems are related by the equivalence relations and refinement orderings, and different process-algebraic modeling formalisms are briefly surveyed.

Keywords: process algebra, equational reasoning, verification, verification tools, bisimulation, failures/testing relations.

1 Introduction

The term *process algebra* encompasses a collection of theories that support mathematically rigorous *(in)equational* reasoning about systems consisting of concurrent, interacting processes. The field grew out of a seminal book due to Milner [34] in 1980 and has been an active area of research since then. In particular, researchers have developed a number of different process-algebraic theories in order to capture different aspects of system behavior; however, each such formalism generally includes the following characteristics.

1. A language, or *algebra*, is defined for describing systems.

2. A *behavioral equivalence* is introduced that is intended to relate systems whose behavior is indistinguishable to an external observer.

3. Equational rules, or *axioms*, are developed that permit proofs of equivalences between systems to be conducted in a syntax-driven manner.

Some formalisms include a *refinement ordering*, in lieu of an equivalence; in this case, the theories allow one to determine if a system is "greater than or equal to" (i.e. refines) another. The literature typically refers to each theory as a process algebra; so the field of process algebra contains many process algebras.

Process algebras derive their motivation from the fact that a system design often consists of several different descriptions of the system involving different levels of detail. The behavioral equivalence or refinement relation provided by a process algebra may be used to determine whether these different descriptions conform to one another. More specifically, higher-level descriptions of system behavior may be related to lower-level ones using the equivalence or refinement ordering supplied by the algebra. These relations are typically *substitutive*, meaning that related systems may be used interchangeably inside larger system descriptions; this facilitates compositional system verification, since low-level designs of system components may be checked in isolation against their high-level designs.

This chapter surveys some of the main features of process algebra, and it develops along the following lines. The next section introduces CCS, the process algebra that we use throughout the chapter to illustrate the principles we cover. Section 3 then introduces behavioral equivalences based on the notion of *bisimulation*, a fundamental concept due to Milner and Park. We then show how two of these equivalences may be given equational axiomatizations. The section following then introduces the failures/testing refinement relations and provides inequational axiomatizations for them for CCS. Section 6 then shows how these relations may be computed for finite-state systems. The penultimate section surveys related work, and the final one summarizes the contents of the chapter.

2 A Calculus of Communicating Systems

This section introduces the syntax and semantics of the process algebra A Calculus of Communicating Systems (CCS). CCS will serve as a vehicle for illustrating the different ingredients that make up a process algebra throughout the remainder of this chapter. Other process algebras are briefly discussed in Section 7.

2.1 The Syntactic Form of CCS Processes

CCS provides a small set of operators that may be used to construct system descriptions from definitions of subsystems. The basic building blocks of these descriptions, and indeed of system definitions in all existing process algebras, are *actions*. Intuitively, actions represent atomic, uninterruptible execution steps, with some actions denoting internal execution and others representing potential interactions with its environment that the system may engage in.

Actions in CCS. A binary, synchronous model of process communication underlies CCS, and the structure of the set of actions reflects this design decision. Actions represent either inputs/outputs on *ports* or internal computation steps. The former are sometimes called *external*, as they require interaction from the environment in order to take place.

To formalize these intuitions, let Λ represent a countably infinite set of labels, or ports, not containing the distinguished symbol τ. Then an action in CCS has one of the following three forms.

- α, where $\alpha \in \Lambda$, represents the act of receiving a signal on port α.
- $\overline{\alpha}$, where $\alpha \in \Lambda$, represents the act of emitting a signal on port α.
- τ represents an internal computation step.

In what follows we use A_{CCS} to stand for the set of all CCS actions; that is,

$$A_{CCS} = \Lambda \cup \{\, \overline{\alpha} \mid \alpha \in \Lambda \,\} \cup \{\tau\}.$$

We also abuse notation by defining $\overline{\overline{\alpha}} = \alpha$; note that $\overline{\tau}$ is not a valid action. We refer to the actions α and $\overline{\alpha}$, where $\alpha \in \Lambda$, as *complementary*, as they represent the input and output action on the same channel. The set $A_{CCS} - \{\tau\}$ then contains the set of external, or visible, actions; the only internal action is τ.

CCS operators. Having defined the set A_{CCS} of CCS actions we now introduce the operators the process algebra provides for assembling actions into systems. In what follows, we assume that p, p_1 and p_2 denote CCS system descriptions that have previously been constructed, and we also assume a countably infinite set $\mathcal{C}$ of *process variables*. CCS then includes seven different mechanisms for building systems.

- *nil* represent the terminated process that has finished execution.
- Given $a \in A_{CCS}$, the *prefixing operator* $a.$ allows an action to be "prepended" onto an existing system description. Intuitively, $a.p$ is capable first of an a and then behaves like p.
- $+$ represents a choice construct. The system $p_1 + p_2$ offers the potential of behaving like either p_1 or p_2, depending on the interactions offered by the environment.
- $|$ denotes parallel composition. The system $p_1|p_2$ interleaves the execution of p_1 and p_2 while also permitting complementary actions of p_1 and p_2 to synchronize; in this case case, the resulting composite action is a τ.

- If $L \subseteq A_{CCS} - \{\tau\}$ then the *restriction* operator $\backslash L$ permits actions to be localized within a system. Intuitively, $p\backslash L$ behaves like p except that it is disallowed from interacting with its environment using actions mentioned in L. Note that τ can never be restricted.

- The operator $[f]$ allows actions in a process to be *renamed*. Here f is a function from A_{CCS} to A_{CCS} that is required to satisfy the following two restrictions.

  - $f(\tau) = \tau$
  - $f(\overline{a}) = \overline{f(a)}$.

  When this is the case, f is called a *renaming*. The system $p[f]$ behaves exactly like p except that f is applied to each action that p wishes to engage in.

- If $C \in \mathcal{C}$ then C represents a valid system provided that a *defining equation* of the form $C \stackrel{\Delta}{=} p$ has been given. Intuitively, C represents an "invocation" that behaves like p. This construct allows systems to be defined recursively.

In process-algebraic parlance, system descriptions built using the above operators are often referred to as *terms* or *processes*. We use $\mathcal{P}_{CCS}$ to represent the set of all CCS processes. As examples, consider the following, where we assume that Λ contains send, recv, msg, ack, get, put, $\mathsf{get\_ack}$ and $\mathsf{put\_ack}$.

- The term $\mathsf{send}.\overline{\mathsf{recv}}.nil$ represents a system that engages in a sequence of two actions: an "input" on the *send* channel, followed by an "output" on the recv channel.

- Consider the definition

  $$\mathsf{M} \stackrel{\Delta}{=} \mathsf{put}.\overline{\mathsf{get}}.M + \mathsf{put\_ack}.\overline{\mathsf{get\_ack}}.\mathsf{M}$$

  This defines a system M that may be thought of as a one-place communication buffer: given a "message" on its put channel it delivers it on its get channel, and similarly for acknowledgments. This example illustrates how, although the version of CCS considered here does not explicitly support value-passing, a limited form of data exchange can be implemented by encoding values in port names. Here M can handle two kinds of "data": messages and acknowledgments.

- Now consider the following definitions, where M is as defined previously.

  $$\begin{aligned} \mathsf{S} &\stackrel{\Delta}{=} \mathsf{send}.\overline{\mathsf{msg}}.\mathsf{ack}.\mathsf{S} \\ \mathsf{R} &\stackrel{\Delta}{=} \mathsf{msg}.\overline{\mathsf{recv}}.\overline{\mathsf{ack}}.\mathsf{R} \\ \mathsf{P} &\stackrel{\Delta}{=} (\mathsf{S}[\mathsf{put}/\mathsf{msg}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\} \end{aligned}$$

  P represents the CCS term for a simple communications protocol consisting of a sender S, a receiver R, and a medium M, a graphical depiction of which may be found in Figure 1. The sender repeatedly accepts "messages" on its send channel, outputs them

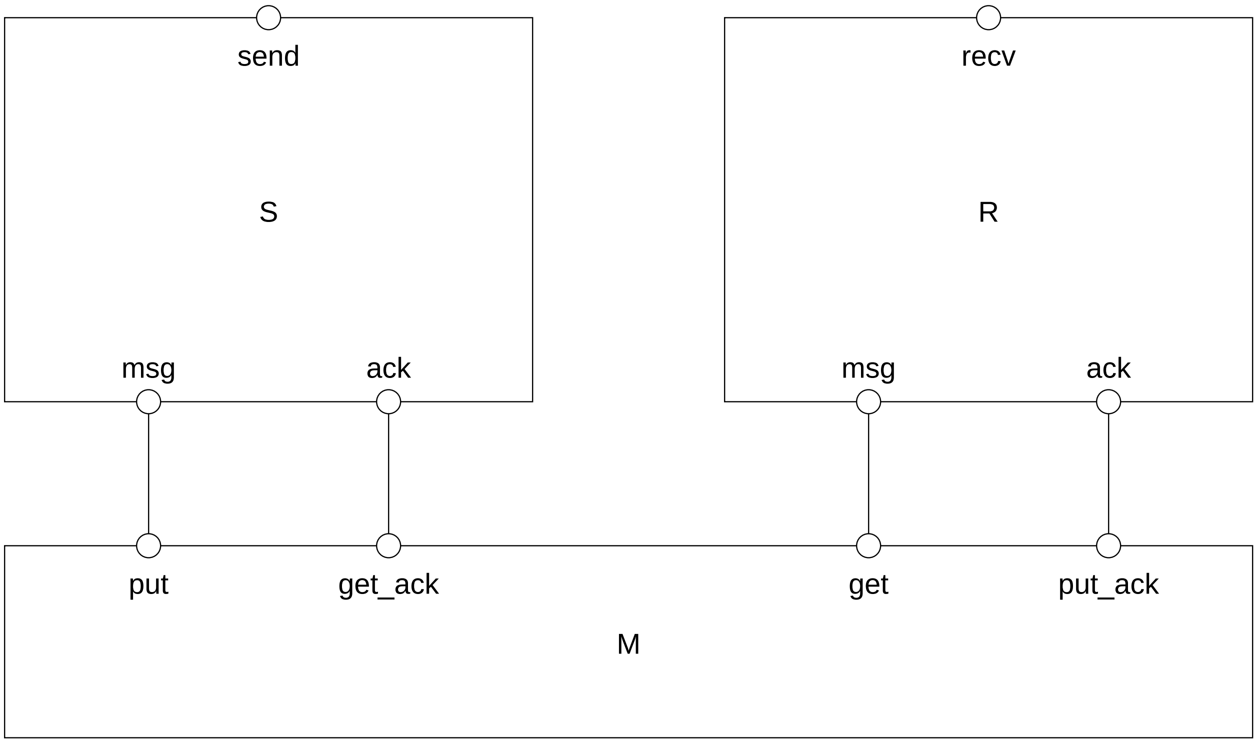

Figure 1: The architecture of a sample communications protocol.

on its msg channel, and then awaits an acknowledgment on its ack channel. The receiver behaves similarly: it awaits a message on its msg channel, delivers it on its recv channel, and then sends an acknowledgment via its ack channel. The relabeling operators are given in the form $a/b, c/d, \ldots$; intuitively, such a relabeling changes b (and its inverse) to a, d to c, etc. Actions not mentioned are unaffected. In this example the relabelings effect the "wiring" given in the figure. The restriction operator ensures that only the sender and receive may interact directly with the medium.

2.2 The Operational Semantics of CCS Terms

In the account so far we have relied on the reader's intuition to understand the meaning of the CCS operators. To make these meanings precise, CCS and other process algebras usually include an *operational semantics* that is intended precisely to define the "execution steps" that processes may engage in. This semantics is usually specified in the form of a ternary relation, $\longrightarrow$; intuitively, $p \stackrel{a}{\longrightarrow} p'$ holds if system p is capable of engaging in action a and then behaving like p'. Process algebras such as CCS typically define $\longrightarrow$ inductively using a collection of *inference rules* for each operator. These rules have the following form.

$$\boxed{\frac{\textit{premises}}{\textit{conclusion}}\ (\textit{side condition})}$$

A rule states that, if one has established the premises, and the side condition holds, then one may infer the conclusion. This presentation style for operational semantics is often called

SOS, for *Structural Operational Semantics*, and was devised by Plotkin [39].

The remainder of this section covers the SOS rules for CCS and shows how they may be used rigorously to characterize the behavior of CCS system descriptions. We group the rules on the basis of the CCS operators to which they apply.

***nil*.** The CCS process *nil* has no rules; consequently, it is incapable of any transitions.

Prefixing. The prefixing operator contains one rule.

$$\boxed{\dfrac{}{a.p \xrightarrow{a} p}}$$

This rule has no premises, and the conclusion states that processes of the form $a.p$ may engage in a and thereafter behave like p. Note that the side condition is omitted; in such cases it is assumed to be "true".

Choice. The choice operator has two symmetric rules.

$$\boxed{\dfrac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'}} \quad \boxed{\dfrac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'}}$$

These rules in essence state that a system of the form $p + q$ "inherits" the transitions of its subsystems p and q.

Parallel Composition. The parallel composition operator has three rules, the first two of which are symmetric.

$$\boxed{\dfrac{p \xrightarrow{a} p'}{p|q \xrightarrow{a} p'|q}} \quad \boxed{\dfrac{q \xrightarrow{a} q'}{p|q \xrightarrow{a} p|q'}}$$

These rules indicate that $|$ interleaves the transitions of its subsystems. The next rule allows processes connected by $|$ to interact.

$$\boxed{\dfrac{p \xrightarrow{a} p',\, q \xrightarrow{\overline{a}} q'}{p|q \xrightarrow{\tau} p'|q'}}$$

According to this rule, subsystems may *synchronize* on complementary actions (i.e. inputs and outputs on the same port). Note that the action produced as the result of the synchronization is a τ; since $\overline{\tau}$ is undefined, this ensures that synchronizations involve only two partners.

Restriction. The restriction operator has one rule.

$$\boxed{\dfrac{p \xrightarrow{a} p'}{p\backslash L \xrightarrow{a} p'\backslash L}\ (a, \overline{a} \notin L)}$$

This rule, which includes a side condition, only allows actions not mentioned in L (or whose complements are not in L) to be performed by $p\backslash L$. Restriction in effect "localizes" actions in L, since the operator forbids the system's environment from interacting with the system using them.

Relabeling. The relabeling operation has one rule.

$$\boxed{\dfrac{p \xrightarrow{a} p'}{p[f] \xrightarrow{f(a)} p'[f]}}$$

As the intuitive account above suggests, $p[f]$ engages in the same transitions as p, the difference being that the actions are relabeled via f.

Process Variables. The behavior of process variables is given by one rule.

$$\boxed{\dfrac{p \xrightarrow{a} p'}{C \xrightarrow{a} p'}\ (C \triangleq p)}$$

This rule states that a system C behaves like its "body", p, provided that C has been provided with a definition of the form $C \triangleq p$.

Examples. As stated above, the SOS rules for CCS define the single-step transitions that CCS processes may engage in. As one example, consider the medium process M defined above. Using the prefixing rule, one may infer the transition

$$\mathsf{put}.\overline{\mathsf{get}}.\mathsf{M} \xrightarrow{\mathsf{put}} \overline{\mathsf{get}}.\mathsf{M}$$

Using this fact and one of the rules for $+$, one may therefore infer that

$$\mathsf{put}.\overline{\mathsf{get}}.\mathsf{M} + \mathsf{put\_ack}.\overline{\mathsf{get\_ack}}.\mathsf{M} \xrightarrow{\mathsf{put}} \overline{\mathsf{get}}.\mathsf{M}$$

This observation and the rule for constants then permit the following transition to be inferred.

$$\mathsf{M} \xrightarrow{\mathsf{put}} \overline{\mathsf{get}}.\mathsf{M}$$

Using similar lines of reasoning, one may also deduce that

$$\mathsf{P} \xrightarrow{\mathsf{send}} ((\overline{\mathsf{msg}}.\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{msg}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{get\_ack}, \mathsf{put}, \mathsf{put\_ack}\}$$

Note that this is the only transition available to P, since the transitions of M and R all involve actions in the restriction set.

2.3 CCS, Processes and Labeled Transition Systems

The definition of $\longrightarrow$ just given allows CCS processes to be viewed as state machines of a certain type. To begin with, we show how CCS may be viewed as a structure called a *labeled transition system* consisting of a collection of possible system states and transitions.

Definition 2.1 *A labeled transition system (LTS) is a triple $\langle Q, A, \longrightarrow\rangle$, where Q is a set of states, A is a set of actions, and $\longrightarrow \subseteq Q \times A \times Q$ is a transition relation.*

Some definitions of LTS also designate a start state. We refer to labeled transitions of this form (i.e. quadruples of the form $\langle Q, A, \longrightarrow, q_S \rangle$ where $q_S \in Q$ is the start state) as *rooted* labeled transition systems.

Perhaps surprisingly, the definitions of this chapter show that CCS may be viewed as a single LTS. Recall that $\mathcal{P}_{CCS}$ represents the (infinite) set of syntactically valid CCS system definitions, and let $\longrightarrow_{CCS}$ be the transition relation defined in the previous subsection. Then $\langle \mathcal{P}_{CCS}, A_{CCS}, \longrightarrow_{CCS} \rangle$ satisfies the definition of LTS. This observation also holds for other process algebras and has two consequences. The first is that certain definitions, such as those for behavioral equivalences and refinement orderings, may be given in a language-independent manner by defining them with respect to LTS's. The second consequence is that that individual system descriptions may be "converted" into rooted LTS's. Mathematically, for any CCS system p the quadruple $\langle \mathcal{P}_{CCS}, A_{CCS}, \longrightarrow_{CCS}, p \rangle$ constitutes a rooted LTS. As $\mathcal{P}_{CCS}$ is infinite this observation is only of theoretical interest until one observes that not every state in $\mathcal{P}_{CCS}$ is "reachable" from p via $\longrightarrow_{CCS}$. Consequently, we may instead define another LTS, M_p, consisting only of CCS terms reachable from p via sequences of transitions. If M_p contains only finitely many states, then it may be analyzed using algorithms for manipulating finite-state machines. As an example, Figure 2 contains the finite-state rooted LTS corresponding to the communication protocol P described above.

3 Behavioral Congruences for CCS

Process algebras usually use a notion of behavioral congruence as a basis for system analysis. A *congruence* for an algebra is an equivalence relation (i.e. a relation that is reflexive, symmetric and transitive) that also has the substitution property: equivalent systems may be used interchangeably inside any larger system. Formally, define a *context* $C[\,]$ to be a system description with a "hole", $[\,]$; given a system description p, then, $C[p]$ represents the system obtained by "filling" the hole with p. Then an equivalence $\approx$ is a congruence for a language if, whenever $p \approx q$, then $C[p] \approx C[q]$ for any context $C[\,]$ built using operators in the language. It should be noted that relations that are congruences for some languages are not congruences for others.

In this section we study congruences for CCS with a view toward defining a relation that relates systems with respect to their "observable" behavior. In each case we first define an equivalence relation on states in an arbitrary LTS; since CCS may be viewed as an LTS, these relations may then be used to relate CCS system descriptions. We then consider the suitability of the equivalence from the standpoint of the observable behavior to which it is sensitive and study whether or not the relation is a congruence for CCS. In the first part of the section we make no special allowance for the "unobservability" of the action τ, deferring its treatment to later.

3.1 The Inadequacy of Trace Equivalence

State machines have a well-studied equivalence, *language equivalence*, that stipulates that two machines are equivalent if they accept the same sequences of symbols. Rooted labeled transition systems do not contain "accepting states" per se, and consequently the notion

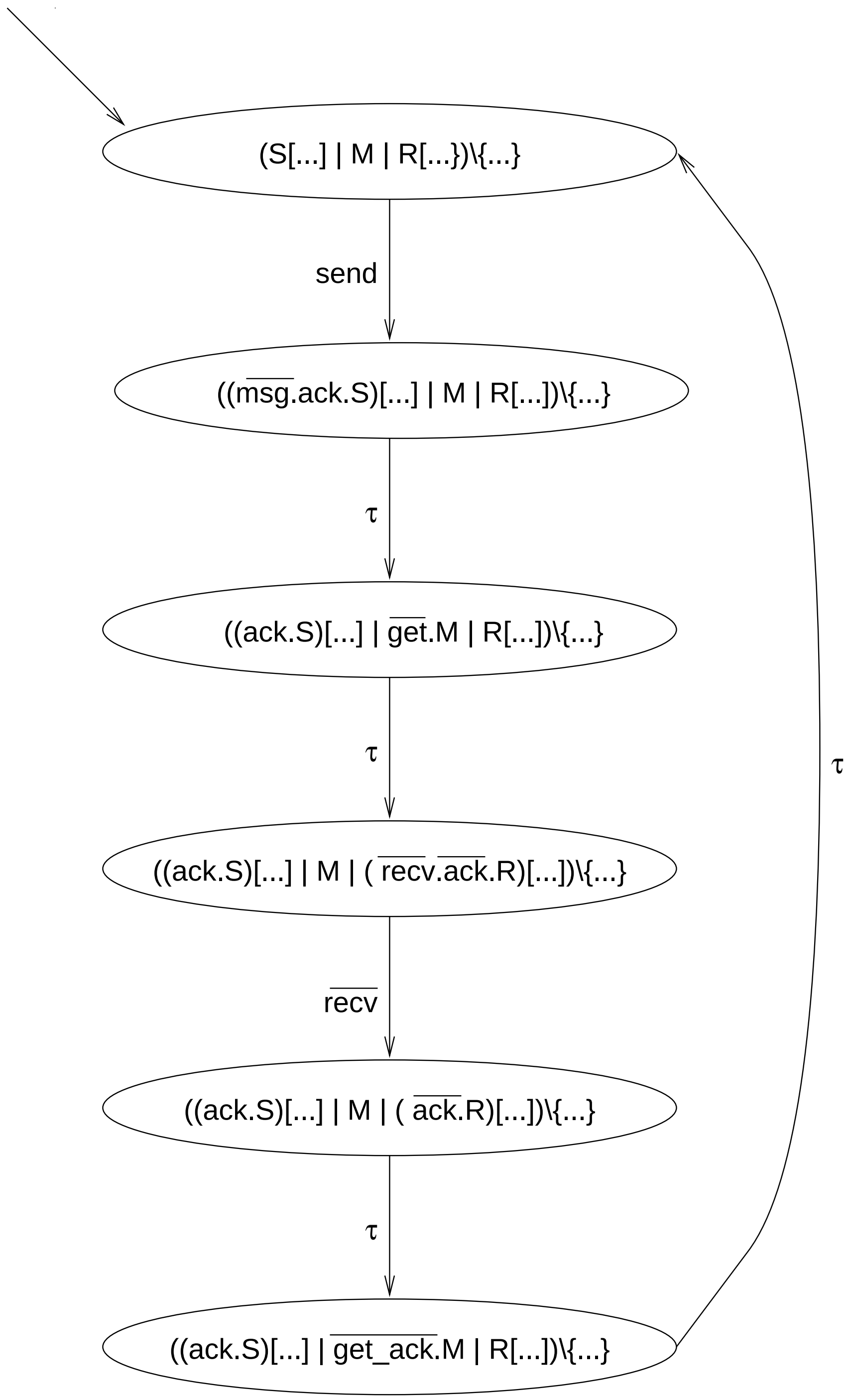

Figure 2: The state machine for P.

of language equivalence from finite-state machine theory cannot be directly applied. However, if we identify every state in a rooted LTS as being accepting, then the "language" of the machine contains the execution sequences, or "traces", that a machine may engage in. Consequently, a reasonable first attempt at defining a behavioral equivalence for CCS and other process algebras might be to relate two system descriptions (i.e. states in the LTS $\langle Q, A, \longrightarrow\rangle$ exactly when the machines for them have exactly the same traces.

Before formalizing these notions we first review some concepts from the theory of finite sequences. If A is a set, then A^* consists of the set of (possibly empty) finite sequences of elements of A. We use ϵ to represent the empty sequence. One may now define traces, and trace equivalence, as follows.

Definition 3.1 *Let $\langle Q, A, \longrightarrow\rangle$ be a labeled transition system.*

1. *Let $s = a_1 \ldots a_n \in A^*$ be a sequence of actions. Then $q \stackrel{s}{\longrightarrow} q'$ if there are states $q_0, \ldots, q_n$ such that $q = q_0$, $q_i \stackrel{a_i}{\longrightarrow} q_{i+1}$, and $q' = q_n$.*
2. *s is a* strong trace *of q if there exists q' such that $q \stackrel{s}{\longrightarrow} q'$. We use $S(q)$ to represent the set of all strong traces of q.*
3. *$p \approx_S q$ exactly when $S(p) = S(q)$.*

We use the term *strong traces* because the definition given above does not distinguish between internal and external actions; all may appear in a strong trace. In contrast, the traditional definition of traces treats τ actions in a special manner.

Since CCS is a labeled transition system whose states are system descriptions we may apply the definition of $\approx_S$ to CCS systems. Unfortunately, $\approx_S$ suffers from severe deficiencies for CCS and other languages that permit the definition of nondeterministic systems, as the following examples illustrate.

1. Let p be $a.b.nil + a.c.nil$ and q be $a.(b.nil + c.nil)$. Then $p \approx_S q$, as $S(p) = S(q) = \{\epsilon, a, ab, ac\}$. However, after an a-transition q can perform both a b and a c, whereas p must reject one or the other of these possibilities after each of its (two) a-transitions.
2. Let $C_1 \stackrel{\Delta}{=} a.C_1$ and $C_2 \stackrel{\Delta}{=} a.C_2 + a.nil$. Then $C_1 \approx_S C_2$, and yet C_2 can reach a "deadlocked" state after an a-transition (i.e. a state that is incapable of any transitions) while C_1 cannot.

The trouble with trace equivalence and nondeterministic systems is that even though two systems have the same traces, they may go through inequivalent states in performing them. (This situation cannot occur in deterministic systems.) In particular, trace equivalent systems can have different deadlocking behavior.

3.2 Bisimulation Equivalence

The last observation in the previous section suggests that an appropriate equivalence for CCS, and indeed for any language permitting the definition of nondeterministic systems, ought to have a recursive flavor: execution sequences for equivalent systems ought to "pass through" equivalent states. This intuition underlies the definition of *bisimulation*, or *strong* equivalence. The name of the equivalence stems from the fact that it is defined in terms of special relations called bisimulations.

Definition 3.2 *Let $\langle Q, A, \longrightarrow \rangle$ be an LTS. A relation $R \subseteq Q \times Q$ is a* bisimulation *if, whenever $\langle p, q \rangle \in R$, then the following conditions hold for any a, p' and q'.*

1. *if $p \xrightarrow{a} p'$ then $q \xrightarrow{a} q'$ for some q' such that $\langle p', q' \rangle \in R$.*
2. *if $q \xrightarrow{a} q'$ then $p \xrightarrow{a} p'$ for some p' such that $\langle p', q' \rangle \in R$.*

Intuitively, if two systems are related by a bisimulation, then it is possible for each to simulate, or "track", the other's behavior: hence the term *bi*simulation. More specifically, for a relation to be a bisimulation, related states must be able to "match" transitions of each other by moving to related states. Two states are then *bisimulation equivalent* exactly when a bisimulation may be found relating them.

Definition 3.3 *Systems p and q are* bisimulation equivalent*, or* bisimilar*, if there exists a bisimulation R containing $\langle p, q \rangle$. We write $p \sim q$ whenever p and q are bisimilar.*

Since CCS may be viewed as an LTS, one may use $\sim$ to relate CCS processes. As examples, we have the following.

1. $a.b.nil + a.b.nil \sim a.b.nil$
2. $a.b.nil + a.c.nil \not\sim a.(b.nil + c.nil)$
3. $C_1 \not\sim C_2$.

Bisimulation equivalence has a number of pleasing properties. Firstly, for any labeled transition system it is indeed an equivalence; that is, the relation $\sim$ is reflexive, symmetric and transitive. Secondly it can be shown in a precise sense that two equivalent systems must have the same "deadlock potential"; this point is addressed in more detail below. Thirdly, $\sim$ implies $\approx_S$ and coincides with it if the LTS is *deterministic* in the sense that every state has at most one outgoing transition per action. Finally, $\sim$ is a congruence for CCS; if $p \sim q$ then p and q may be used interchangeably inside any larger system.

However, $\sim$ does suffer from a major flaw from the perspective of CCS and other process algebras allowing asynchronous execution: it is too sensitive to internal computation. In particular, the definition does not take account of the special status that τ has *vis à vis* other actions. For example, the systems $a.\tau.b.nil$ and $a.b.nil$ are not bisimulation equivalent, even though an external observable cannot detect the difference between them. Nevertheless, $\sim$ has been studied extensively in the literature, and for process algebras in which internal computation in one component can indeed affect the behavior of other components, it is a reasonable basis for verification.

Deadlock, Logical Characterizations and $\sim$

The preceding discussion states that $\sim$ relates systems on the basis of their relative "deadlock potentials". The remainder of this subsection makes this statement precise by defining a logic, called the Hennessy-Milner Logic (HML) [26], that permits the formulation of simple system properties, including potentials for deadlock. The logic also characterizes $\sim$ in the following sense: two systems are bisimilar if and only if they satisfy exactly the same formulas in the logic.

Syntax of HML. The definition of HML is parameterized with respect to a set A of actions. Given such a set, the syntax of HML formulas can be given via the following grammar.

$$\begin{array}{rcl} \phi & ::= & tt \\ & | & ff \\ & | & \phi \wedge \phi \\ & | & \phi \vee \phi \\ & | & \langle a \rangle \phi \\ & | & [a]\phi \end{array}$$

We use Φ for the set of all well-formed HML formulas.

The constructs in the logic may be understood as follows. First, it should be noted that formulas are intended to be interpreted with respect states in a labeled transition system. Then tt and ff represent the constants "true" and "false" that hold of any state and no state, respectively, while $\wedge$ and $\vee$ denote conjunction ("and") and disjunction ("or"), respectively. The final two operators are referred to as *modalities*, as they permit statements to be made about the transitions emanating from a state; thus HML is a *modal logic*. A state satisfies $\langle a \rangle \phi$ if a target state of one of its a-transitions satisfies ϕ, while $[a]\phi$ holds of a state if the target states of all of its a-transitions satisfy ϕ.

Semantics of HML. In order to formalize the previous informal discussion, we first fix a labeled transition system $\mathcal{L} = \langle Q, A, \longrightarrow \rangle$ having the same action set as HML. We then define a relation $\models_{\mathcal{L}} \subseteq Q \times \Phi$; intuitively, $q \models_{\mathcal{L}} \phi$ should hold if state q "satisfies" ϕ. The formal definition is given inductively as follows.

- $q \models_{\mathcal{L}} tt$ for any $q \in Q$.
- $q \models_{\mathcal{L}} ff$ for no $q \in Q$.
- $q \models_{\mathcal{L}} \phi_1 \wedge \phi_2$ if and only if $q \models_{\mathcal{L}} \phi_1$ and $q \models_{\mathcal{L}} \phi_2$.
- $q \models_{\mathcal{L}} \phi_1 \vee \phi_2$ if and only if $q \models_{\mathcal{L}} \phi_1$ or $q \models_{\mathcal{L}} \phi_2$.
- $q \models_{\mathcal{L}} \langle a \rangle \phi$ if and only if $q \xrightarrow{a} q'$ and $q' \models_{\mathcal{L}} \phi$ for some $q' \in Q$.
- $q \models_{\mathcal{L}} [a]\phi$ if and only if for every q' such that $q \xrightarrow{a} q'$, $q' \models_{\mathcal{L}} \phi$.

This definition includes some subtleties that deserve comment. To begin with, formula $[a]\mathit{ff}$ is satisfied by any state *not* having an a-transition; such states vacuously fulfill the requirement imposed by $[a]$. Indeed, a state with no a-transitions satisfies $[a]\phi$ for any ϕ. These facts also imply that a state incapable of any action in the set $\{a_1, \ldots, a_n\}$ will satisfy the formula $[a_1]\mathit{ff} \wedge \cdots \wedge [a_n]\mathit{ff}$. If such a state occurs in an environment that requires one of these actions, then a deadlock results. In a related vein, a state satisfies $\langle b \rangle \mathit{tt}$ if and only if it has an b-transition; more generally, given a (nonempty) sequence of actions $b_1 \ldots b_m$, a state includes $b_1 \ldots b_m$ as one of its strong traces if and only if the state satisfies the formula $\langle b_1 \rangle \cdots \langle b_m \rangle \mathit{tt}$. Finally, consider a state satisfying a formula of the form

$$\langle b_1 \rangle \cdots \langle b_m \rangle ([a_1]\mathit{ff} \wedge \cdots \wedge [a_n]\mathit{ff}).$$

Such a state satisfies this formula if it can engage in the sequence $b_1 \ldots b_m$ and arrive at a state that rejects offers for interaction involving any of $a_1, \ldots, a_n$. In an environment capable of exercising the sequence $b_1 \ldots b_m$ and then requiring an interaction involving one of $a_1, \ldots, a_n$, the given state could deadlock. It is in this sense that HML permits the formulation of properties expressing potentials for deadlock.

HML and $\sim$. The relationship between HML and $\sim$ is captured by the following theorem that states that HML characterizes $\sim$ for labeled transition systems that are *image-finite*. An LTS is image-finite if every state in the LTS has at most finitely many transitions sharing the same action label. In practice almost all labeled transition systems satisfy this requirement; in particular, CCS does provided the definitions of process variables obey a small restriction.

Theorem 3.4 *Let $\mathcal{L} = \langle Q, A, \longrightarrow \rangle$ be an image-finite LTS, and let $p, q \in Q$. Then $p \sim q$ if and only if for all HML formulas ϕ, either $p \models_{\mathcal{L}} \phi$ and $q \models_{\mathcal{L}} \phi$ or $p \not\models_{\mathcal{L}} \phi$ and $q \not\models_{\mathcal{L}} \phi$.*

On the one hand, this result and the previous discussion substantiates the claim that bisimulation equivalence requires equivalent systems to have the same "deadlock potentials". On the other hand, the theorem provides a useful mechanism for explaining why two systems fail to be equivalent; one need only present a formula satisfied by one system and not the other. The following provides examples illustrating this latter point in the context of CCS.

- Consider the system p given by $a.b.\mathit{nil}+a.c.\mathit{nil}$ and the system q given by $a.(b.\mathit{nil}+c.\mathit{nil})$. Since $p \not\sim q$ there must be a formula satisfied by one and not the other. One such formula is $\langle a \rangle [b]\mathit{ff}$, which is satisfied by p but not by q.
- Consider C_1 and C_2 given above. The formula $\langle a \rangle [a]\mathit{ff}$ distinguishes them, as C_2 satisfies it and C_1 does not.

3.3 Observational Equivalence and Congruence for CCS

This subsection presents a coarsening of bisimulation equivalence that is intended to relax the sensitivity of the former to internal computation. The definition of this relation relies on the introduction of so-called "weak" transitions.

Definition 3.5 *Let $\langle Q, A, \longrightarrow\rangle$ be an LTS with $\tau \in A$, and let $q \in Q$.*

1. *If $s \in A^*$ then $\hat{s} \in (A - \{\tau\})^*$ is the action sequence obtained by deleting all occurrences of τ from s.*

2. *Let $s \in (A - \{\tau\})^*$. Then $q \stackrel{s}{\Longrightarrow} q'$ if there exists s' such that $q \stackrel{s'}{\longrightarrow} q'$ and $s = \hat{s'}$.*

Intuitively, $\hat{s}$ returns the "visible content" (i.e. non-τ elements) of sequence s; in particular, if $a \in A$ then $\hat{a} = \epsilon$ if $a = \tau$, while $\hat{a} = a$ if $a \neq \tau$. In addition, $q \stackrel{s}{\Longrightarrow} q'$ if q can perform a sequence of transitions with the same visible content as s and evolve to q'. In this case note that the sequence of transitions that is performed is the same as s except that it potentially includes an arbitrary number of τ transitions in between the visible actions of s. In particular, $q \stackrel{\epsilon}{\Longrightarrow} q'$ if a sequence of τ-transitions leads from q to q', while for a single visible action a, $q \stackrel{a}{\Longrightarrow} q'$ if q can perform an a, possibly "surrounded" by some internal computation, in order to arrive at q'.

We may now define *weak bisimulations* as follows.

Definition 3.6 *Let $\langle Q, A, \longrightarrow\rangle$ be an LTS, with $\tau \in A$. Then a relation $R \subseteq Q \times Q$ is a* weak bisimulation *if, whenever $\langle p, q\rangle \in R$, then the following hold for all $a \in A$ and $p', q' \in Q$.*

1. *If $p \stackrel{a}{\longrightarrow} p'$ then $q \stackrel{\hat{a}}{\Longrightarrow} q'$ for some q' such that $\langle p', q'\rangle \in R$.*

2. *If $q \stackrel{a}{\longrightarrow} q'$ then $p \stackrel{\hat{a}}{\Longrightarrow} p'$ for some p' such that $\langle p', q'\rangle \in R$.*

States p and q are observationally equivalent, *or* weakly equivalent, *or* weakly bisimilar, *if there exists a weak bisimulation R containing $\langle p, q\rangle$. When this is the case we write $p \approx q$.*

A weak bisimulation closely resembles a regular bisimulation; the only difference lies in the fact that systems may use weak transitions to simulate normal transitions in the other system.

As CCS is a labeled transition system whose action set contains τ, the definition of $\approx$ may be used to relate CCS system descriptions. Doing so leads to the following observations.

- $a.\tau.b.nil \approx a.b.nil$.
- For any p, $\tau.p \approx p$.
- Let $\mathsf{Svc} \stackrel{\Delta}{=} \mathsf{send}.\overline{\mathsf{recv}}.\mathsf{Svc}$. Then $\mathsf{P} \approx \mathsf{Svc}$, where P is the simple communications protocol described in the previous section.

The last example illustrates the power of equivalences in relating system designs at different levels of abstraction, since Svc could be thought of as a "high-level" design that P is intended to conform to.

Even though it ignores internal computation observational equivalence still enjoys a similar degree of deadlock-sensitivity to bisimulation equivalence: a variant of HML can be defined that characterizes $\approx$ in the same way that HML characterizes $\sim$. (This logic replaces the $\langle a\rangle$ and $[a]$ modalities of HML by two new operators, $\langle\langle a\rangle\rangle$ and $[[a]]$; a state

$q \models_{\mathcal{L}} \langle\langle a \rangle\rangle \phi$ if there exists a q' such that $q \stackrel{a}{\Longrightarrow} q'$ and $q' \models_{\mathcal{L}} \phi$, and similarly for $[[a]]$.) Consequently it would appear to be a viable candidate for relating CCS system descriptions. Unfortunately, however, it is *not* a congruence for CCS. To see why, consider the context $C[\,]$ given by $[\,] + b.nil$. It is easy to establish that $p \approx q$, where p is given by $\tau.a.nil$ and q by $a.nil$. However, $C[p] \not\approx C[q]$. To see this, note that $C[p] \stackrel{\tau}{\longrightarrow} a.nil$. This transition must be matched by a weak ϵ-labeled transition from $C[q]$. The only such transition $C[q]$ has is $C[q] \stackrel{\epsilon}{\Longrightarrow} C[q]$. However, $a.nil \not\approx C[q]$, since the latter can engage in a b-labeled transition that cannot be matched by the former.

This defect of $\approx$ arises from the interplay between $+$ and the initial internal computation that a system might engage in; in particular, the only CCS operator that "breaks" the congruence-hood of $\approx$ is $+$. Some researchers reasonably suggest that this is an argument against including $+$ in the language. Milner [34, 36] adopts another point of view that we pursue in the remainder of this section, and that is to focus on finding the *largest* CCS congruence $\approx^C$ that implies $\approx$. Such a largest congruence is guaranteed to exist [26].

Definition 3.7 *Let $\langle Q, A, \longrightarrow \rangle$ be an LTS with $\tau \in A$, and let $p, q \in Q$. Then $p \approx^C q$ if the following hold for all $a \in A$ and $p', q' \in Q$.*

1. *If $p \stackrel{a}{\longrightarrow} p'$ then $q \stackrel{a}{\Longrightarrow} q'$ for some q' such that $p' \approx q'$.*
2. *If $q \stackrel{a}{\longrightarrow} q'$ then $p \stackrel{a}{\Longrightarrow} p'$ for some p' such that $p' \approx q'$.*

Some remarks about this relation are in order. Firstly, it should be noted that for $p \approx^C q$ to hold, any τ-transition of p must be matched by a $\stackrel{\tau}{\Longrightarrow}$-transition of q; in particular, this weak transition must consist of a *non-empty* sequence of τ-transitions. Secondly, the definition is not recursive: the targets of initial matching transitions need only be related by $\approx$. Finally, it indeed turns out that $\approx^C$ is a congruence for CCS and that it is the largest CCS congruence entailing $\approx$. That is, $p \approx^C q$ implies $p \approx q$, and for any other congruence R such that $p\,R\,q$ implies $p \approx q$, $p\,R\,q$ also implies $p \approx^C q$. As examples, we have the following.

1. $a.\tau.b.nil \approx^C a.b.nil$
2. $\tau.a.nil \not\approx^C a.nil$, since the $\stackrel{\tau}{\longrightarrow}$ transition of the former cannot be matched by a $\stackrel{\tau}{\Longrightarrow}$ transition of the latter.
3. For any p, q, if $p \approx q$ then $\tau.p \approx^C \tau.q$.
4. $\mathsf{Svc} \approx^C \mathsf{P}$, where Svc and P are as defined above.

4 Equational Reasoning in CCS

In addition to definitions of behavioral congruences, process algebras traditionally provide *equational axiomatizations* that permit equivalences to be established by means of simple syntactic manipulations. This section presents such axiomatizations for CCS for both $\sim$ and $\approx^C$.

Table 1: Axiomatizing $\sim$ for Basic CCS: Rule Set E_1.

$$\begin{array}{lrcl}
\text{(A1)} & x+y & = & y+x \\
\text{(A2)} & x+(y+z) & = & (x+y)+z \\
\text{(A3)} & x+\mathit{nil} & = & x \\
\text{(A4)} & x+x & = & x
\end{array}$$

4.1 Axiomatizing $\sim$

We present the axiomatization of $\sim$ for CCS in stages by considering successively larger fragments of CCS. The first, and most basic, subset of CCS we investigate we term "Basic CCS".

4.1.1 Axiomatizing Basic CCS

Basic CCS contains only the *nil*, prefixing and + operators of CCS, and hence it only allows the definition of "sequential" (i.e. no parallelism) terminating systems. The axiomatization of $\sim$ for Basic CCS consists of the four rules given in Table 1.

Some words of explanation about these axioms are in order. Firstly, and for convenience, each rule we present has a name; in this case, the rules are named (A1)–(A4). Secondly, each rule contains variables that are intended to be arbitrary terms in the language under consideration. In (A2), for example, x, y and z are variables, and the rule should be read as asserting that regardless of the Basic CCS terms substituted for these variables, the indicated equivalence holds. Finally, axioms are used to construct equational proofs as illustrated by the following example.

$$\begin{array}{rcll}
a.(b.\mathit{nil}+\mathit{nil})+(a.\mathit{nil}+a.b.\mathit{nil}) & = & a.b.\mathit{nil}+(a.\mathit{nil}+a.b.\mathit{nil}) & \text{by (A3)} \\
& = & a.b.\mathit{nil}+(a.b.\mathit{nil}+a.\mathit{nil}) & \text{by (A1)} \\
& = & (a.b.\mathit{nil}+a.b.\mathit{nil})+a.\mathit{nil} & \text{by (A2)} \\
& = & a.b.\mathit{nil}+a.\mathit{nil} & \text{by (A4)}
\end{array}$$

This proof establishes that $a.(b.\mathit{nil}+\mathit{nil})+(a.\mathit{nil}+a.b.\mathit{nil}) = a.b.\mathit{nil}+a.\mathit{nil}$ in four steps, where each step represents the "application" of a rule to a subterm, yielding a new term. The development of such equational proofs typically relies on four rules of inference reflecting the fact that = is reflexive, symmetric, and transitive and that equal terms may be used interchangeably; these rules implicitly support the construction of proofs such as the one above. We will not say more about this matter.

When a proof that terms t_1 and t_2 exists using axioms in set E, we write $E \vdash t_1 = t_2$. Thus,

$$E_1 \vdash a.(b.\mathit{nil}+\mathit{nil})+(a.\mathit{nil}+a.b.\mathit{nil}) = a.b.\mathit{nil}+a.\mathit{nil},$$

where E_1 contains the four rules in Table 1.

Returning to the rules in Table 1, Rules (A1) and (A2) assert that + is commutative and associative, respectively. Rule (A3) indicates that *nil* is an *identity element* for +; these first three rules are sometimes referred to as the *monoid laws*, a monoid being any mathematical

Table 2: Axiomatizing $\sim$ for Basic Parallel CCS: Rule Set E_2.

(A1)–(A4) from Table 1

(Exp) $(\sum_{i\in I} a_i.x_i) \mid (\sum_{j\in J} b_j.y_j) =$
$\sum_{i\in I} a_i.(x_i \mid \sum_{j\in J} b_j.y_j) \; + \; \sum_{j\in J} b_j.((\sum_{i\in I} a_i.x_i) \mid y_j) \; + \; \sum_{\{(i,j) \mid a_i=\overline{b_j}\}} \tau.(x_i \mid y_j)$

structure obeying these axioms. The final rule is often called the *absorption law*, as it allows multiple copies of the same summand to be "absorbed" into one.

Metatheory. Given a proposed axiomatization for an equivalence relation, one may ask two questions.

1. Is the axiomatization *sound*? That is, are all proved equalities true?

2. Is the axiomatization *complete*? That is, are all true equalities provable?

Soundness is an absolute necessity; an unsound proof system is worse than useless, since it allows the derivation of untrue information. Completeness is highly desirable, since once a proof system is shown complete, one knows that there can be no "missing" axioms.

The following results establish the soundness and completeness of the axioms in Table 1 for $\sim$ over Basic CCS.

Theorem 4.1 (Soundness)
Let t_1 and t_2 be terms in Basic CCS, and suppose that $E_1 \vdash t_1 = t_2$. Then $t_1 \sim t_2$.

Theorem 4.2 (Completeness)
Let t_1 and t_2 be terms in Basic CCS such that $t_1 \sim t_2$. Then $E_1 \vdash t_1 = t_2$.

4.1.2 Axiomatizing Basic Parallel CCS

The next fragment of CCS we present an axiomatization for extends Basic CCS with the inclusion of the parallel composition operator, $|$. We call this fragment Basic Parallel CCS.

As it turns out Rules (A1)–(A4) remain sound for Basic Parallel CCS, but they are obviously not complete, since none of the rules mentions $|$. In order to devise a complete axiomatization for this subset of CCS we therefore must add axioms for $|$. The new axiomatization is presented in Table 2.

The single new axiom, (Exp), is often referred to as the *expansion law*, as it shows how terms involving $|$ at the top level may be "expanded" into ones involving prefixing and summation. This axiom is the most complicated rule for CCS, and it deserves further commentary. Firstly, the $\sum$ notation needs explanation. Rules (A1) and (A2) indicate that $+$ is commutative and associative. This means that expressions of the form $t_1 + \cdots + t_n$, while not strictly speaking expressions since they are not fully parenthesized, nevertheless have a precise meaning, since all parenthesizations of such expressions are equivalent. More generally, given a finite index set I and an I-indexed set of terms of the form t_i, we may define $\sum_{i \in i} t_i$ as *nil* if I is empty and as the summation of all the t_i's otherwise.

The second feature of (Exp) is that it may only be applied to a term $t_1|t_2$ if both t_1 and t_2 have a special form: namely, each must be a summation of terms whose outermost operator involves prefixing. Technically speaking, (Exp) is not a single axiom but an axiom schema, with each different value of I and J yielding a different axiom.

Finally, the right-hand side of (Exp) consists of three summands, each corresponding to a different SOS rule for $|$. The first summand allows the left subterm to "move" autonomously, and the second permits the same behavior from the right subterm. The third summand handles possible synchronizations.

To see how (Exp) is used in equation proofs, consider the following example showing that $E_2 \vdash nil \,|\, b.nil = b.nil$; recall that *nil* is the same as $\sum_{i \in \emptyset} t_i$.

$$\begin{array}{rcll} nil \,|\, b.nil & = & nil + b.(nil \,|\, nil) + nil & \text{by (Exp)} \\ & = & b.(nil \,|\, nil) & \text{by (A3) twice} \\ & = & b.(nil + nil + nil) & \text{by (Exp)} \\ & = & b.nil & \text{by (A3) twice} \end{array}$$

Indeed, for any term t in Basic Parallel CCS it follows that $E_2 \vdash nil \,|\, t = t$. It may also be shown that for any terms t_1, t_2 and t_3 $E_2 \vdash t_1|t_2 = t_2|t_1$ and $E_2 \vdash t_1|(t_2|t_3) = (t_1|t_2)|t_3$; consequently, $|$ is commutative and associative. Finally, as the strict application of (Exp) results in many occurrences of *nil* as a summand, these *nil*'s are suppressed in practice, since they may be removed by applying (A3) appropriately.

It may be shown that E_2 is a sound and complete axiomatization of $\sim$ for Basic Parallel CCS.

4.1.3 Axiomatizing $\sim$ for Finite CCS

The next fragment of CCS we axiomatize includes all operators except for process variables; the literature refers to this fragment as Finite CCS. Finite CCS extends Basic Parallel CCS with the restriction and relabeling operators; the axioms for this subset of CCS appear in Table 3.

The axioms for $\backslash L$ and $[f]$ only explain how these operators interact with *nil*, prefixing and $+$. That no rules are needed defining the interaction between $|$ and $\backslash L$, or $\backslash L$ and $[f]$, is a consequence of the fact that the innermost occurrences of these so-called *static* operators (with *nil*, prefixing and $+$ being the *dynamic* ones) can be eliminated by repeated use of the laws for the operator in conjunction with (A1)–(A4). This argument may be formalized and used to show that rule set E_3 constitutes a sound and complete axiomatization of $\sim$ for Finite CCS.

4.1.4 Rules for Recursive Processes

In order to axiomatize full CCS, we need rules for reasoning about terms that include process variables. Unfortunately, results from computability theory imply that no complete axiom-

Table 3: Axiomatizing $\sim$ for Finite CCS: Rule Set E_3.

(A1)–(A4) from Table 1; (Exp) from Table 2

$$
\begin{array}{lrcl}
\text{(Res1)} & nil\backslash L & = & nil \\
\text{(Res2)} & (a.x)\backslash L & = & \begin{cases} nil & \text{if } a, \overline{a} \in L \\ a.(x\backslash L) & \text{otherwise} \end{cases} \\
\text{(Res3)} & (x+y)\backslash L & = & x\backslash L + y\backslash L
\end{array}
$$

$$
\begin{array}{lrcl}
\text{(Rel1)} & nil[f] & = & nil \\
\text{(Rel2)} & (a.x)[f] & = & f(a).(x[f]) \\
\text{(Rel3)} & (x+y)[f] & = & x[f] + y[f]
\end{array}
$$

atization can exist for $\sim$ for full CCS.[1] However, two useful heuristics have been developed for handling process variables, and we review these here.

Both techniques take the form of inference rules and are therefore similar in form to the SOS rules used to define the operational semantics of CCS. The first rule, called the *unrolling rule*, states that a process invocation is equivalent to the body of the invocation.

$$\text{(Unr)} \boxed{\frac{C \triangleq p}{C = p}}$$

The second inference rule is often called the *unique fixpoint induction* principle, and stating it relies on introducing the notion of *equation* and *solution*. Given a variable X and a CCS term t potentially containing X, such that at most X appears "free" in t,[2] we call the expression $X = t$ an *equation*. A CCS process p is a *solution* to $X = t$ if and only if $p \sim t[p/X]$, where $t[p/X]$ is the CCS term obtained by replacing all occurrences of variable X by p. An equation has a *unique solution up to* $\sim$ if for any two solutions p and q to the equation, $p \sim q$. We may now formulate the unique fixpoint induction rule as follows.

$$\text{(UFI)} \boxed{\frac{p = t[p/X] \quad q = t[q/X]}{p = q} \ (X = t \text{ has a unique solution})}$$

This rule allows one to conclude that two terms are equal, provided one can prove that they are both solutions to the same equation and the equation has a unique solution.

A couple of comments about (UFI) are in order. Firstly, every equation $X = t$ has a solution: given definition $X \triangleq t$, it is easy to see that process X is a solution of $X = t$. Secondly, (UFI) is only useful insofar as one may readily identify when equations have a unique solution. One such class of equations, and a large one at that, can be defined as follows.

[1] The set of equalities one can prove using any axiomatization can only be recursively enumerable; however, $\sim$ for full CCS is known not to be recursively enumerable.

[2] For the present discussion, variables only occur free; i.e. they are not bounded by a fixed-point operator. See [36] for further details.

Table 4: Axiomatizing $\approx^C$ for Finite CCS: Rule Set E_4.

(A1)–(A4) from Table 1; (Exp) from Table 2; (Res1)–(Res3), (Rel1)–(Rel3) from Table 3

$$
\begin{array}{lrcl}
(\tau 1) & a.\tau.x & = & a.x \\
(\tau 2) & x + \tau.x & = & \tau.x \\
(\tau 3) & a.(x + \tau.y) & = & a.(x + \tau.y) + a.y
\end{array}
$$

Definition 4.3 *Let X be a variable, and t be a CCS term involving X. Then X is* guarded *in t if every occurrence of X in t falls within the scope of a prefix operator.*

For example, X is guarded in $a.X$ and $a.X|(b.(X + c.nil))$, but it is not guarded in $X + b.X$. We now have the following result.

Theorem 4.4 *Let X be guarded in t. Then equation $X = t$ has a unique solution up to $\sim$.*

As an application of (Unr) and (UFI), suppose we wish to prove that A and B are bisimilar, where $A \triangleq a.A$ and $B \triangleq a.a.B$. Consider the equation $X = a.a.X$. We can show that both A and B are solutions to this equation:

$$
\begin{array}{rcll}
A & = & a.A & \text{by (Unr)} \\
 & = & a.a.A & \text{by (Unr)} \\
 & & & \\
B & = & a.B & \text{by (Unr)}
\end{array}
$$

Since X is guarded in $a.a.X$, $X = a.a.X$ has a unique solution, and consequently using (UFI) one may conclude that $A = B$.

4.2 Axiomatizing $\approx^C$

This section presents an axiomatization for $\approx^C$ and CCS. Following the development in the previous subsection, we first consider the Finite CCS fragment and then full CCS.

4.2.1 Axiomatizing Finite CCS

To begin with, it should be noted that the axioms in rule set E_3 of Table 3 are also sound for $\approx^C$, since whenever $p \sim q$ it immediately follows that $p \approx^C q$. In order to obtain a full axiomatization for $\approx^C$, then, we need only add axioms reflecting the special status of the action τ in this congruence.

One tempting axiom to add would be $x = \tau.x$; however, this is not sound for $\approx^C$, since it would allow one to prove that $\tau.a.nil = a.nil$, which is not valid. The correct rules are listed in Table 4 and are often called the τ *laws.*

Rule ($\tau 1$) allows for the "absorption" of τ actions that immediately follows prefixing operations. Rule ($\tau 2$) is more subtle, and may be understood as follows. First, note that any strong transition of $\tau.x$ is also a strong transition of $x + \tau.x$. Secondly, any strong transition of $x + \tau.x$, including any τ-transition, may be matched by an appropriate weak transition in $\tau.x$. The final rule, ($\tau 3$) is perhaps the most difficult to interpret; note that the strong transition

$$a.(x + \tau.y) + a.y \xrightarrow{a} y$$

of the right-hand side may however be matched by the weak transition

$$a.(x + \tau.y) \overset{a}{\Longrightarrow} y$$

of the left-hand side.

Somewhat surprisingly, these rules suffice; the axiomatization E_4 is sound and complete for $\approx^C$ and Finite CCS.

4.2.2 Axiomatizing Full CCS

The same observations for $\sim$ also hold for $\approx^C$ *vis à vis* sound and complete axiomatizations: none can exist. The (Unr) and (UFI) rules nevertheless still hold, although the characterization of which equations have unique fixpoints becomes somewhat more complex; guardedness no longer suffices. To see this, consider the equation $X = \tau.X$. X is guarded in $\tau.X$, and yet any process capable of an initial τ action is a solution to this equation up to $\approx^C$. In particular, $\tau.a.nil \approx^C \tau.\tau.a.nil$ and $\tau.b.nil \approx^C \tau.\tau.b.nil$, and yet $\tau.a.nil \not\approx^C \tau.b.nil$.

One potential solution to this problem is to require a stronger condition than guardedness in equations.

Definition 4.5 *Let X be a variable and t a CCS term involving X. Then X is* strongly guarded *in t if every occurrence of X falls within the scope of a prefixing operator a where $a \neq \tau$.*

That is, X is strongly guarded in t if a prefix operator involving a visible action "guards" each occurrence of X in t. Note that X is not strongly guarded in $\tau.X$. However, even if X is strongly guarded in t it does not follow that $X = t$ has a unique solution up to $\approx^C$. To see this, consider the equation

$$X = (a.X \mid \overline{a}.nil)\backslash\{a\}.$$

X is strongly guarded in the right-hand side of the equation, and yet it can be shown that e.g. $\tau.b.nil$ and $\tau.c.nil$ are both solutions. We may nevertheless fix this problem by requiring the following.

Definition 4.6 *Let X be a variable and t a CCS term involving X. Then X is* sequential *in t if no occurrence of X in t falls within the scope of a parallel composition operator.*

As examples, X is sequential in $a.X$ and $\tau.X + (b.nil \mid c.nil)$ but not sequential in $a.X \mid b.nil$. The following can now be proved.

Theorem 4.7 *Let $X = t$ be an equation with X strongly guarded and sequential in t. Then $X = t$ has a unique solution up to $\approx^C$.*

We conclude this section with an extended example illustrating the use of the axioms. Recall the simple communications protocol P given in Section 2.1 and the specification Svc in Section 3.3. We may establish that $E_4 \cup \{(\text{Unr}), (\text{UFI})\} \vdash \mathsf{P} = \mathsf{Svc}$ as follows. First note that X is strongly guarded and sequential in $\mathsf{send}.\overline{\mathsf{recv}}.X$ and consequently has a unique solution up to $\approx^C$. Therefore, we need only show that both P and Svc are solutions to this equation; then, by (UFI), $\mathsf{P} = \mathsf{Svc}$. Now,

$$\mathsf{Svc} \quad = \quad \mathsf{send}.\overline{\mathsf{recv}}.\mathsf{Svc} \quad \text{by (Unr)}$$

so Svc is a solution. As for P, we can prove that

$$\mathsf{P} = (\mathsf{S}[\mathsf{put}/\mathsf{msg}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}]) \backslash \{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}$$

using (Unr), so it suffices to prove that the right-hand side is a solution to the given equation. The proof of this may be found in Figure 3.

5 Refinement Orderings for CCS

This chapter has so far concentrated on the role of behavioral equivalences in process algebra in general, and CCS in particular. We now shift our attention to refinement orderings, and to a particular class of refinement orderings that are often referred to as the failures/testing orderings. This section presents a definition of these orderings and gives axiomatizations for them for CCS.

5.1 The Failures/Testing Orderings

The motivation for the failures/testing orderings arises from two sources. On the one hand, equivalences sometimes impose overly severe restrictions on a designer defining a lower-level design that is intended to implement a higher-level one. In particular, equivalences require that the behaviors of the designs be identical; this precludes a higher-level design offering several possibilities for behavior or including "don't-care points". This suggests that an ordering in which a "more deterministic" system is larger, or "better", than a less deterministic one would be desirable. On the other hand, while $\approx$ and $\approx^C$ abstract from internal computation and are sensitive to deadlock, it can be argued that they are overly sensitive to unobservable differences in the branching structure of systems. As an example, consider the two CCS definitions $P \triangleq a.b.c.nil + a.b.d.nil$ and $Q \triangleq a.(b.c.nil + b.d.nil)$. These two systems are not related by $\approx$; the formula $[[a]]\langle\langle b\rangle\rangle\langle\langle c\rangle\rangle tt$ is satisfied by the latter and not the former. However, a user ought not to be able to distinguish them, since to a user it does not matter when the nondeterministic choice that ultimately eliminates the possibility of c or d is made.

The failures [12, 27] and testing [19, 25] orderings differ substantially in their approaches to addressing these issues, and yet the resulting orderings turn out to coincide. In this section we follow the failures presentation given in [32] because it requires the introduction of less notation given the machinery we have already developed. We need the following definitions.

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\mathsf{S}[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid R[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}} \\
= & \mathsf{send}. \\
& ((\overline{\mathsf{msg}}.\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by (Exp), (Rel1)–(Rel3), (Res1)–(Res3)} \\
= & \mathsf{send}.\tau. \\
& ((\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid (\overline{\mathsf{get}}.\mathsf{M}) \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by (Exp), (Rel1)–(Rel3), (Res1)–(Res3)} \\
= & \mathsf{send}. \\
& ((\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid (\overline{\mathsf{get}}.\mathsf{M}) \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by } (\tau 1) \\
= & \mathsf{send}.\tau. \\
& ((\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid (\overline{\mathsf{recv}}.\overline{\mathsf{ack}}.\mathsf{R})[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by (Exp), (Rel1)–(Rel3), (Res1)–(Res3)} \\
= & \mathsf{send}. \\
& ((\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid (\overline{\mathsf{recv}}.\overline{\mathsf{ack}}.\mathsf{R})[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by } (\tau 1) \\
= & \mathsf{send}.\overline{\mathsf{recv}}. \\
& ((\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid (\overline{\mathsf{ack}}.\mathsf{R})[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by (Exp), (Rel1)–(Rel3), (Res1)–(Res3)} \\
= & \mathsf{send}.\overline{\mathsf{recv}}.\tau. \\
& ((\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \overline{\mathsf{get\_ack}}.\mathsf{M} \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by (Exp), (Rel1)–(Rel3), (Res1)–(Res3)} \\
= & \mathsf{send}.\overline{\mathsf{recv}}. \\
& ((\mathsf{ack}.\mathsf{S})[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \overline{\mathsf{get\_ack}}.\mathsf{M} \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by } (\tau 1) \\
= & \mathsf{send}.\overline{\mathsf{recv}}.\tau \\
& (\mathsf{S}[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by (Exp), (Rel1)–(Rel3), (Res1)–(Res3)} \\
= & \mathsf{send}.\overline{\mathsf{recv}}. \\
& (\mathsf{S}[\mathsf{put}/\mathsf{ack}, \mathsf{get\_ack}/\mathsf{ack}] \mid \mathsf{M} \mid \mathsf{R}[\mathsf{get}/\mathsf{msg}, \mathsf{put\_ack}/\mathsf{ack}])\backslash\{\mathsf{get}, \mathsf{put}, \mathsf{get\_ack}, \mathsf{put\_ack}\}) \\
& \text{by } (\tau 1)
\end{array}
$$

Figure 3: Proving that $\mathsf{P} = \mathsf{Svc}$.

Definition 5.1 *Let $\langle Q, A, \longrightarrow \rangle$ be an LTS with $\tau \in A$, let $q \in Q$, and let $s \in (A - \{\tau\})^*$ be a sequence of visible actions.*

1. *$q \stackrel{s}{\Longrightarrow}$ holds if there exists q' such that $q \stackrel{s}{\Longrightarrow} q'$. In this case we say s is a* trace *of q. $L(q)$ denotes the set of all traces of q.*

2. *q* refuses *$B \subseteq A - \{\tau\}$ if $|B| < \infty$ and for all $b \in B$, there exists no q' such that $q \stackrel{b}{\Longrightarrow} q'$.*

3. *q is* divergent, *written $q \Uparrow$, if and only if there exists an infinite sequence $q_0, q_1, \ldots$ such that $q = q_0$ and $q_i \stackrel{\tau}{\longrightarrow} q_{i+1}$ for all $i \geq 0$. $q \Uparrow s$ if and only if there exists a (possibly empty) prefix s' of s and state q' such that $q \stackrel{s'}{\Longrightarrow} q'$ and $q' \Uparrow$. When this is the case we say q* diverges *on s. We write $q \Downarrow s$ if $q \Uparrow s$ is* not *true and say that q converges on s in this case.*

4. *A state q is* totally convergent *if $q \Downarrow s$ holds for all sequences s.*

5. *Let s be a sequence of visible actions and $B \subseteq A$ be finite. Then $\langle s, B \rangle$ is a* failure *for q if either $q \Uparrow s$ or there is a q' such that $q \stackrel{s}{\Longrightarrow} q'$ and q' refuses B. We use $F(q)$ to represent the set of all failures of q.*

The failures/testing ordering rely on the notions of *trace*, *refusal*, *divergence* and *failure*. Intuitively, a trace of a state consists of a sequence of visible actions the state can perform, with arbitrary amounts of internal computation allowed in between. A refusal consists of a finite set of visible actions that a state is incapable of engaging in, no matter how much internal computation is performed. A state is divergent if it can engage in an infinite sequence of internal transitions, thereby ignoring its environment; $q \Uparrow s$ holds if, in the course of "executing" s, q could enter a divergent state. Finally, a failure consists of a sequence of actions and a set of "offered actions" that a state can fail to complete, either by diverging in the course of performing the sequence or completing the sequence and arriving at a state that is incapable of responding to the offered actions. As examples, consider the following.

- The pair $\langle a, \{b\} \rangle$ is a failure of $a.b.nil + a.c.nil$ and of $a.(\tau.b.nil + \tau.c.nil)$ but not of $a.(b.nil{+}c.nil)$. Both of the former processes have $\stackrel{a}{\Longrightarrow}$ transitions to $c.nil$, which refuses $\{b\}$; the last process has no such transition.

- Consider $\mathsf{D} \stackrel{\Delta}{=} \tau.\mathsf{D}$; $\mathsf{D} \Uparrow s$ for any sequence s of visible actions, and consequently $\langle s, B \rangle$ is a failure for any D for any sequence s and finite set of actions B.

The sets $L(q)$ and $F(q)$ satisfy a number of properties. For example, the empty sequence ϵ is in $L(q)$ for any q. In addition, if $q \Downarrow s$ then $s \in L(q)$ if and only if there is a B such that $\langle s, B \rangle \in F(q)$. It should also be noted that if $\langle s, B \rangle \in F(q)$ and $B' \subseteq B$ then $\langle s, B' \rangle \in F(q)$. Readers are referred to [32] for other such properties.

We now introduce the following orderings and equivalences.

Definition 5.2 *Let $\langle Q, A, \longrightarrow \rangle$ be an LTS, with $p, q \in Q$.*

1. *$p \sqsubseteq_L q$ if $L(p) \subseteq L(q)$; $p \approx_L q$ if $p \sqsubseteq_L q$ and $q \sqsubseteq_L p$.*

2. *$p \sqsubseteq_F q$ if $F(p) \supseteq F(q)$; $p \approx_F q$ if $p \sqsubseteq_F q$ and $q \sqsubseteq_F p$.*

The orderings $\sqsubseteq_L$ and $\sqsubseteq_F$ capture different aspects of system behavior. The former relates systems on the basis of their execution sequences; a "lesser" system has fewer execution possibilities. The latter identifies failure as undesirable; consequently, a "lesser process" has *more* possibilities for failure than a "greater one." In this case failure can either be the result of nondeterminism or of divergence; the more nondeterministic or divergent a system is, the more failures it has.

Both orderings are *preorders* on Q; that is, they are reflexive and transitive relations. The relation $\sqsubseteq_L$ is also referred to as the *may preorder* in [19, 25], while $\sqsubseteq_F$ is called the *must preorder*. This terminology derives from connections with process testing: $p \sqsubseteq_L q$ holds if and only if every test that p may pass may also be passed by q, in a precisely defined sense, while $p \sqsubseteq_F q$ holds if and only if every test that p must pass must also be passed by q. In addition, if q is totally convergent, then $p \sqsubseteq_F q$ implies that $q \sqsubseteq_L p$. This follows because for any failure $\langle s, B\rangle$ of q, $s \in L(q)$.

Finally, it should be noted that in CCS, the system Div given by $\mathsf{Div} \triangleq \tau.\mathsf{Div}$ is a least element for both $\sqsubseteq_L$ and $\sqsubseteq_F$. That is, $\mathsf{Div} \sqsubseteq_L p$ and $\mathsf{Div} \sqsubseteq_F p$ for any p.

For many process algebras $\sqsubseteq_L$ and $\sqsubseteq_F$ are *precongruences*: "larger" systems may be substituted for "smaller" ones inside any context, with the resulting over-all system being larger after the substitution. For CCS, $\sqsubseteq_L$ is a precongruence, but $\sqsubseteq_F$ is not, owing to the impact that initial internal computation can have on the $+$ operator. As was the case with $\approx$, one may identify the largest precongruence $\sqsubseteq_F^C$ contained within $\sqsubseteq_F$ for CCS; it turns out that for CCS systems p and q, $p \sqsubseteq_F^C q$ if and only if the following hold: $p \sqsubseteq_F q$, and $p \stackrel{\tau}{\not\longrightarrow}$ implies $q \stackrel{\tau}{\not\longrightarrow}$.

The relations $\sqsubseteq_F$ and $\sqsubseteq_F^C$ have attracted much more attention in the literature than $\sqsubseteq_L$ because of certain *full-abstractness* results that have been established for the former. In particular, for a number of languages it turns out that $\sqsubseteq_F$ and $\sqsubseteq_F^C$ are the coarsest (i.e. most permissive) preorders that preserve deadlock information, in a precisely defined sense. Accordingly, the remainder of this section is devoted to a study of $\sqsubseteq_F^C$.

5.2 Axiomatizing $\sqsubseteq_F^C$ for CCS

As was the case for $\sim$ and $\approx^C$, $\sqsubseteq_F^C$ has been axiomatized for (fragments of) CCS. We present the axiomatization for Finite CCS below and talk briefly about mechanisms for handling recursive processes.

5.2.1 Finite CCS

The axiomatization for Finite CCS appears in Table 5. Unlike the other axiomatizations we have seen, it is an *inequational* axiomatization: it is used to prove statements of the form $p \leq q$ rather than $p = q$. The axioms therefore include inequalities; equalities such as Rule (F1) should be interpreted as short-hand for two inequalities, one in each direction.

Table 5: Axiomatizing $\sqsubseteq_F^C$ for Finite CCS: Rule Set E_5

(A1)–(A4) from Table 1; (Exp) from Table 2

$$\begin{array}{lrcl}
\text{(F1)} & a.x + a.y & = & a.(\tau.x + \tau.y) \\
\text{(F2)} & x + \tau.y & \leq & \tau.(x + y) \\
\text{(F3)} & a.x + \tau.(a.y + z) & = & \tau.(a.x + a.y + z) \\
\text{(F4)} & \tau.x & \leq & x \\
\text{(F5)} & \tau.x + \tau.y & \leq & x
\end{array}$$

To see how these rules may be used to derive results, we give a sample proof of $E_5 \vdash a.b.nil + a.c.nil \leq a.b.nil$.

$$\begin{array}{rcll}
a.b.nil + a.c.nil & = & a.(\tau.b.nil + \tau.c.nil) & \text{by (F1)} \\
& \leq & a.b.nil & \text{by (F5)}
\end{array}$$

The rules in Table 5 are sound and complete for $\sqsubseteq_F^C$ for Finite CCS.

5.2.2 Reasoning about Recursive Processes

To handle recursive processes, one may use Rules (Unr) and (UFI) as given in Section 4.1.4. A sufficient condition for the existence of unique solutions to equations includes a requirement of *divergence-freedom* in addition to the strong-guardedness and sequentiality requirements needed for $\approx^C$.

Interpreting systems as sets of failures also permits the use of reasoning techniques from fixed-point theory in denotational semantics [25]. This is because the collection of sets of failures can be turned into a *domain*. We do not pursue this topic further, however.

6 Computing Behavioral Relations for Finite-State Systems

The previous sections have developed several semantic equivalences and refinement orderings in the context of CCS, and (in)equational axiomatizations have been presented for determining when two systems are related. However, the equational reasoning supported by these axiomatizations is tedious to undertake by hand. When the systems in question are *finite-state*, meaning that the rooted labeled transitions systems for them contain only finitely many distinct states, these relations can be computed algorithmically. This section discusses some of the ideas underlying these decision procedures.

6.1 Computing Behavioral Equivalences

Most behavioral equivalences can be computed by combining appropriate LTS transformations with an algorithm for calculating bisimulation equivalence [17]. Accordingly, we first discuss techniques for deciding $\sim$ and then show how these methods may be used in the computation of other equivalences as well.

6.1.1 Calculating $\sim$

Algorithms for $\sim$ come in two basic varieties. *Global* algorithms require the *a priori* construction of the state spaces of the systems in question before any analysis can be undertaken. *On-the-fly* approaches, on the other hand, combine analysis with state-space construction. The latter algorithms offer obvious potential benefits: when systems are inequivalent, this may be determined by examining only a subset of their states. These approaches are relatively new, however, and have not proven themselves in practice. Global approaches also enjoy better asymptotic efficiency than existing on-the-fly methods. Consequently, we only discuss the former.

Global approaches to calculating $\sim$ over a finite-state LTS [20, 29, 38] compute the *equivalence classes* of $\sim$ using approximation-refinement techniques. Typically, these algorithms begin with a very coarse approximation to $\sim$: they assume that every state is related to every other state, meaning that there is one equivalence class. Existing classes that are found to contain inequivalent states are then split. The determination of inequivalence relies on examining the transitions emanating from states in a given equivalence class and the equivalence classes containing the targets of these transitions. When no more splitting is possible, the final equivalence classes indeed represent the equivalence classes of $\sim$ over the given LTS. These algorithms are sometimes called *partition-refinement* algorithms, as the collections of equivalence classes are maintained as partitions (i.e. lists of disjoint sets of states). The best algorithm has complexity $O(m \log n)$, where m represents the number of transitions in the LTS and n the number of states [20, 38].

In order to use a partition-refinement algorithm to determine whether two CCS expressions are bisimilar, one would first construct the labeled transition system whose states consist of all CCS expressions reachable from the two in question. A partition-refinement algorithm may then be applied to this LTS, and if the two expressions in question ever wind up in different equivalence classes, they are inequivalent. Otherwise, if the refinement procedure terminates with them in the same class, then they are equivalent.

Partition-refinement algorithms may also be used to *minimize* LTS's with respect to $\sim$. This is done by replacing states by equivalence classes; the resulting LTS contains exactly one state per equivalence class.

6.1.2 Computing Other Equivalences

As the introduction to this section indicates, a variety of other behavioral equivalences may be computed by first applying a transformation to the underlying LTS and then using an algorithm for $\sim$. Here we present two examples of this approach.

Calculating $\approx$. To calculate the $\approx$-equivalence classes of an LTS, one may alter the LTS by replacing the $\xrightarrow{a}$-transitions by $\xRightarrow{\hat{a}}$-transitions and then computing $\sim$ over the transformed LTS. A similar approach works for $\approx^C$, although one must first transform the LTS to ensure that the start state contains no incoming transitions and then replace $\xrightarrow{a}$-transitions from the start state by $\stackrel{a}{\Longrightarrow}$-transitions (and not $\stackrel{\hat{a}}{\Longrightarrow}$-transitions).

Computing $\approx_S$. To determine whether two states in a given finite-state LTS are strong trace equivalent, one may apply the well-known subset construction to *determinize* the LTS [28] and then compute the equivalence classes of $\sim$. The two states in question will have the same strong traces if and only if the subsets containing only these states are bisimilar in the transformed LTS.

6.2 Computing Refinement Orderings

The calculation of refinement orderings follows a similar pattern to that of equivalences: a given ordering can be computed by combining an LTS transformation with a procedure for a certain generic ordering [15, 17]. The generic ordering is somewhat less standard than its equivalence-relation counterpart, $\sim$, but in many cases the *simulation ordering* may be used. In the remainder of this section we define this ordering and indicate very briefly how it is used as a basis for computing other relations.

The simulation ordering. Given an LTS $\langle Q, A, \longrightarrow\rangle$, a *simulation* is a relation $R \subseteq Q \times Q$ with the property that when $\langle p, q\rangle \in R$, then the following holds for all $a \in A$.

$$p \xrightarrow{a} p' \text{ implies } q \xrightarrow{a} q' \text{ for some } q' \text{ with } \langle p', q'\rangle \in R.$$

So if p is related to q in a simulation, then q can "simulate" the behavior of p by "matching" its transitions. The simulation ordering then may be defined by: $p \sqsubseteq q$ if and only if there exists a simulation R with $\langle p, q\rangle \in R$.

Algorithms for computing $\sqsubseteq$ on finite-state LTS's follow a similar strategy to that for $\sim$ in that they use approximation refinement. Initially, every state is assumed to be related to every other state; then, as pairs of states are found not to be related because the first has a transition that can't be "simulated" by the second, they are removed. When no more pairs can be removed, the remaining pairs constitute $\sqsubseteq$ for this LTS.

Since $\sqsubseteq$ is not an equivalence, partitions cannot be used as data structures, and the resulting algorithms exhibit somewhat worse worst-case performance: the best algorithms use $O(mn)$ time, where m is the number of transitions and n the number of states [15].

Computing other orderings. As an example of how an algorithm for $\sqsubseteq$ may be used in the calculation of other relations consider the trace-containment relation: $p \sqsubseteq_L q$ if and only if $L(p) \subseteq L(q)$. This relation may be computed by first replacing $\xrightarrow{a}$ transitions by $\stackrel{\hat{a}}{\Longrightarrow}$ ones, determinizing the resulting LTS using the subset construction, and then applying a $\sqsubseteq$ algorithm to the result. Other relations, including the failures/testing ordering, may be computed similarly [17].

6.3 Tool Support

Several tools have been implemented that include implementations of algorithms for different behavioral relations. Noteworthy examples include Aldébaran [11], the Concurrency Workbench [18] and FDR [40].

7 Other Process Algebras

The presentation in this chapter has focused on a particular process algebra, CCS, and on semantic relations for CCS. In this section we discuss other process algebras and process-algebra-oriented results. Since 1980 over 1000 journal and conference papers have been published in the area; as a result, the discussion here will necessarily be incomplete. Interested readers are referred to the forthcoming *Handbook of Process Algebra*, to be published by Elsevier, for a more complete account of the state of the art.

Schools of process algebras. The discussion in this chapter has followed the approach advocated by the *Edinburgh school* of process algebra, so named because CCS was invented at the University of Edinburgh. The Edinburgh school places primacy on operational semantics, with equivalences and refinement relations then defined on labeled transition systems resulting from these operational definitions. The chief virtue of this approach lies in its insistence on understanding language constructs operationally; this emphasis accords well with intuitions about system behavior. The drawback of this approach arises from the fact that since operational equivalences and refinement orderings are defined on language-independent structures (i.e. labeled transition systems), determining which relations are congruences becomes nontrivial.

Two other schools of process algebra have also arisen. The *Amsterdam school* focuses on equational axioms as the basis for defining the semantics of languages [6]. In this approach one defines the syntax of an algebra and then provides a set of axioms that one uses to deduce equivalences. Traditional techniques from universal algebra may then be used to construct models of these equational theories. These constructions ensure that the model-theoretic notions of equivalence are congruences for the language in question; the drawback is that equations may, to a certain degree, obscure the operational intuitions underlying operators in the algebra.

The *Oxford school* focuses on *denotational semantics* as the basis for defining process algebras [27]. The Oxford approach relies on defining a mathematical space of *system meanings* and then interpreting algebraic operators as functions in this space. The space most studied by Oxford adherents consists of *failure sets* as presented in Section 5; process constructors then become functions mapping sets of failures to sets of failures. As with the Amsterdam approach, the virtue of this methodology is that the semantic equivalence inherited from the semantic space is guaranteed to be a congruence for the language; additionally, traditional techniques from denotation semantics may be used to define the semantics of recursive processes in a mathematically elegant fashion. The drawback arises from the paucity of operational insight the semantics provides for the operators.

Operators in traditional algebras. The different schools just mentioned have also traditionally focused on including somewhat different operators in their algebras. CCS includes a parallel composition operator that supports binary synchronous communication. The Algebra of Communicating Processes (ACP) algebra developed by the Amsterdam school on the other hand allows the specific communication mechanism to be parameterized; by including different axioms one obtains different synchronization behavior. ACP also includes a traditional sequential composition operator that generalizes the prefixing construct of CCS. Theoretical CSP (TCSP), the process algebra studied by the Oxford school, features multiway rendezvous as its model of interaction; a hiding operator allows actions to be converted into internal actions. Another novel feature of this language is its separation of choice (i.e. $+$) into two constructs, external and internal. The former can only be resolved by visible actions, while the latter is always resolved autonomously, without interaction from the environment.

These algebras have also inspired the development of LOTOS, a process algebra with explicit data passing that is an ISO standard protocol specification notation [9]. LOTOS combines CSP-like operators with a facility for user-defined data types; like CCS, actions may be categorized as inputs or outputs, with the former extended with a capability for binding incoming values to variables and the latter including specific values to be output.

Algebras for synchrony. Traditional process algebras, including those mentioned above, typically include a synchronous model of communication but an asynchronous model of execution. That is, processes interact by synchronizing, but not every process in a system need execute in order for a system transition to take place. This makes traditional algebras useful for modeling loosely coupled systems, but it renders them problematic as vehicles for describing synchronous, globally-clocked systems like traditional digital circuits. To overcome this difficulty, several researchers have proposed algebras whose parallel composition operator requires all subsystems to engage in transitions in order for the system to perform an execution step. The best-known of these is Synchronous CCS [35], whose action set forms a commutative group whose product operator is interpreted as "simultaneous execution." Other synchronous process algebras of note include Meije [2] and CIRCAL [33]; the latter was specifically developed for reasoning about circuits. All three algebras use equivalences based on strong bisimulation; weak equivalences such as observational equivalence will necessarily not be congruences for such languages, since the internal computation a subsystem may engage in will directly affect the transitions available to the surrounding system.

Meta-algebraic results. The algebras just described feature a variety of different operators; in each case, the Edinburgh approach (which has become dominant) requires the proof of congruence results for bisimulation. Some researchers have addressed this problem by proving that, provided the SOS rules defining a language's operators satisfy a certain format, bisimulation is guaranteed to be a congruence [7, 8, 23]. Other results show how equational axiomatizations for languages satisfying these requirements may be automatically derived [1].

Other semantic relations. Researchers have also investigated relations other than the ones presented here. Branching bisimulation [41] aims to remedy a perceived defect of observational congruence that allows transitions in one process to be matched by weak transitions in the other that permit inequivalent states to be "transitioned through". This equivalence is somewhat finer (i.e. relates fewer systems) than observational equivalence, and a sound and complete axiomatization for finite ACP terms, and algorithms for finite-state systems, have been developed. Ready simulation [7] represents a refinement ordering that is fully abstract for deadlock when the language considered includes all operators definable using SOS rules of a certain format. A number of other relations have also been proposed; the interested reader is referred to [42] for a thorough survey and taxonomy.

Capturing other features of system behavior. Traditional process algebras have focused on nondeterminism and synchronization as the essential behavioral features distinguishing concurrent systems from sequential ones. Inspired by the elegance of the resulting theories, researchers have attempted to develop operational theories that allow other aspects of system behavior to be captured (in)equationally. One strand of inquiry has focused on so-called *true concurrency*. One criticism of traditional process algebras is that they "reduce" concurrency to nondeterminism by interpreting parallelism as interleaving. Truly concurrent models instead attempt to capture "true" notions of simultaneity. A number of different theories have been developed, and a full account is beyond the scope of this chapter. A good starting point, however, may be found in [10], which introduces the notion of *location* of a transition explicitly into the operational semantics of CCS and develops a bisimulation-based theory of equivalence based on this.

Other work has focused on including notions of *priority* into the operational semantics of process algebras. The first such work [4] extends ACP action with priority and and operator for "enforcing" priorities. In [16] CCS actions are enriched with a two-level priority structure, with high-priority actions intuitively being thought of as "interrupts". Camilleri and Winskel [14] opt instead for a prioritized choice operator that gives precedence to one choice over another when both are enabled. Also worthy of note is the resource-oriented process algebra ACSR [21], which allows the modeling of resource contention in which different resource requests may be given different priorities. In all of these cases, semantic equivalences based on strong bisimulation are defined and axiomatizations developed.

Process algebras for real-time systems have also been developed. Generally speaking, these theories introduce special "time-passing" actions, with all other actions being viewed as instantaneous. The Algebra of Timed Processes [37] pioneered this approach, with useful variants being proposed in [30].

Another area of ongoing research involves the incorporation of probabilistic behavior into systems, with a view toward providing a theory in which quality-of-service statements can be made. One strand of this research augments traditional process algebra with notions of probabilistic choice in which nondeterminism is resolved probabilistically [5, 31, 43]. Other pieces of work incorporate notions of time and probability in order to model stochastic systems, in which the time needed to perform a given action is drawn from a continuous probability distribution. Noteworthy examples include [22, 24].

8 Conclusion

This chapter has surveyed results in the area of process algebra. It has presented several behavioral equivalences and refinement orderings, and it has shown how they may be axiomatized in the setting of CCS [36]. Decision procedures for finite-state systems have also been touched on. The treatment has necessarily been sketchy, and much interesting material has been omitted, including a variety of case studies illustrating different applications of process algebra. Interested readers may turn to [3, 13, 40] as a starting point for investigating this topic.

References

[1] L. Aceto, B. Bloom, and F. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, May 1994.

[2] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Science*, 30:91–131, 1984.

[3] J.C.M. Baeten, editor. *Applications of Process Algebra*, volume 17 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1990.

[4] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informatica*, 9:127–168, 1986.

[5] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. Axiomatizing probabilistic processes: ACP with generative probabilities. *Information and Computation*, 121(2):234–255, September 1995.

[6] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1990.

[7] . Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced. *Journal of the Association for Computing Machinery*, 42(1):232–268, January 1995.

[8] R.N. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. *Journal of the Association for Computing Machinery*, 43(5):863–914, September 1996.

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

[10] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing localities. *Theoretical Computer Science*, 114(1):31–61, June 1993.

[11] A. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the Aldébaran toolset. *Software Tools for Technology Transfer*, 1(1+2):166–183, December 1997.

[12] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[13] G. Bruns. *Distributed Systems Analysis with CCS.* Prentice-Hall, London, 1997.

[14] J. Camilleri and G. Winskel. CCS with priority choice. *Information and Computation*, 116(1):26–37, January 1995.

[15] U. Celikkan and R. Cleaveland. Generating diagnostic information for behavioral preorders. *Distributed Computing*, 9:61–75, 1995.

[16] R. Cleaveland and M.C.B. Hennessy. Priorities in process algebra. *Information and Computation*, 87(1/2):58–77, July/August 1990.

[17] R. Cleaveland and M.C.B. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[18] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[19] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1983.

[20] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989/1990.

[21] R. Gerber and I. Lee. A resource-based prioritized bisimulation for real-time systems. *Information and Computation*, 113(1):102–142, August 1994.

[22] R. Gorrieri, M. Roccetti, and E. Stancampiano. A theory of processes with durational actions. *Theoretical Computer Science*, 140(1):73–94, March 1995.

[23] J.F. Groote and F. Vaandrage. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

[24] P. Harrison and J. Hillston. Process algebras and their application to performance modelling. *The Computer Journal*, 38(7):489–491, 1995.

[25] M.C.B. Hennessy. *Algebraic Theory of Processes.* MIT Press, Boston, 1988.

[26] M.C.B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, January 1985.

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

[28] J. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[29] P. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[30] K. Larsen and W. Yi. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75–101, May 1997.

[31] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, September 1991.

[32] M. Main. Trace, failure and testing equivalences for communicating processes. *International Journal of Parallel Programming*, 16(5):383–400, 1987.

[33] G. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[34] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[35] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

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

[37] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114(1):131–178, October 1994.

[38] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[39] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[40] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[41] R. van Glabbeek and P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the Association for Computing Machinery*, 43(3):555–600, March 1996.

[42] R.J. van Glabbeek. *Comparative Concurrency Semantics, with Refinement of Actions*. PhD thesis, Free University, Amsterdam, 1990.

[43] R.J. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, August 1995.