Process Algebras

Rocco De Nicola

Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
Viale Morgagni 65, 50134 Firenze, Italia
rocco.denicola@unifi.it

DEFINITION: Process Algebras are mathematically rigorous languages with well defined semantics that permit describing and verifying properties of concurrent communicating systems. They can be seen as models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. The agents may be real-world objects (even people), or they may be artifacts, embodied perhaps in computer hardware or software systems. Many different approaches (operational, denotational, algebraic) are taken for describing the meaning of processes. However, the operational approach is the reference one. By relying on the so called Structural Operational Semantics (SOS), labelled transition systems are built and composed by using the different operators of the many different process algebras. Behavioral equivalences are used to abstract from unwanted details and identify those systems that react similarly to external experiments.

SYNONYMS: Process Calculi, Process Description Languages.

RELATED ENTRIES: Actors, Behavioral Equivalences, Bisimulation, CCS, CSP, Pi-Calculus.

1 Introduction

The goal of software verification is to assure that developed programs fully satisfy all the expected requirements. Providing a formal semantics of programming languages is an essential step toward program verification. This activity has received much attention in the last 40 years. At the beginning the interest was mainly on sequential programs, then it turned also on concurrent program that can lead to subtle errors in very critical activities. Indeed, most computing systems today are concurrent and interactive.

Classically, the semantics of a sequential program has been defined as a function specifying the induced input-output transformations. This setting becomes, however, much more complex when concurrent programs are considered, because they exhibit nondeterministic behaviors. Nondeterminism arises from programs interaction and cannot be avoided. At least, not without sacrificing expressive power. Failures do matter, and choosing the wrong branch might result in an "undesirable situation". Backtracking is usually not applicable, because the control might be distributed. Controlling nondeterminism is very important. In sequential programming, it is just a matter of efficiency, in concurrent programming it is a matter of avoiding getting stuck in a wrong situation.

The approach based on process algebras has been very successful in providing formal semantics of concurrent systems and proving their properties. The success is witnessed by the Turing Award given to two of their pioneers and founding fathers: Tony Hoare and Robin Milner. Process algebras are mathematical models of processes, regarded as agents that act and interact continuously with other similar agents and with their common environment. Process algebras provide a number of constructors for system descriptions and are equipped with an operational semantics that describes systems evolution in terms of labelled transitions. Models and semantics are built by taking a compositional approach that permits describing the "meaning" of composite systems in terms of the meaning of their components.

Moreover, process algebras often come equipped with observational mechanisms that permit identifying (through behavioral equivalences) those systems that cannot be taken apart by external observations (*experiments* or *tests*). In some cases, process algebras have also algebraic characterizations in terms of equational axiom systems that exactly capture the relevant identifications induced by the beavioral operational semantics.

The basic component of a process algebra is its syntax as determined by the well-formed combination of operators and more elementary terms. The syntax of a process algebra is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language. There are many approaches to providing a rigorous mathematical understanding of the semantics of syntactically correct process terms. The main ones are those also used for describing the semantics of sequential systems, namely operational, denotational and algebraic semantics.

An *operational semantics* models a program as a labelled transition system (LTS) that consists of a set of states, a set of transition labels and a transition relation. The states of the transition system are just process algebra terms while the labels of the transitions between states represent the actions or the interactions that are possible from a given state and the state that is reached after the action is performed by means of visible and invisible actions. The operational semantics, as the name suggests, is relatively close to an abstract machine-based view of computation and might be considered as a mathematical formalization of some implementation strategy.

A *denotational semantics* maps a language to some abstract model such that the meaning/denotation (in the model) of any composite program is determinable directly from the meanings/denotations of its subcomponents. Usually, denotational semantics attempt to distance themselves from any specific implementation strategy, describing the language at a level intended to capture the "essential meaning" of a term.

An *algebraic semantics* is defined by a set of algebraic laws which implicitly capture the intended semantics of the constructs of the language under consideration. Instead of being derived theorems (as they would be in a denotational semantics or operational semantics), the laws are the basic axioms of an equational system, and process equivalence is defined in terms of what equalities can be proved using them. In some ways it is reasonable to regard an algebraic semantics as the most abstract kind of description of the semantics of a language.

There has been a huge amount of research work on process algebras carried out during the last 30 years that started with the introduction of CCS [31, 32], CSP [11] and ACP [6]. In spite of the many conceptual similarities, these process algebras have been developed starting from quite different viewpoints and have given rise to different approaches (for an overview see, e.g. [2]).

CCS takes the operational viewpoint as its cornerstone and abstracts from unwanted details introduced by the operational description by taking advantage of behavioral equivalences that allow one to identify those systems that are indistinguishable according to some observation criteria. The meaning of a CCS term is a labeled transition system factored by a notion of observational equivalence. **CSP** originated as the theoretical version of a practical language for concurrency and is still based on an operational intuition which, however, is interpreted w.r.t. a more abstract theory of decorated traces that model how systems react to external stimuli. The meaning of a CSP term is the set of possible runs enriched with information about the interactions that could be refused at intermediate steps of each run. **ACP** started from a completely different viewpoint and provided a purely algebraic view of concurrent systems: processes are the solutions of systems of equations (axioms) over the signature of the considered algebra. Operational semantics and behavioral equivalences are seen as possible models over which the algebra can be defined and the axioms can be applied. The meaning of a term is given via a predefined set of equations and is the collection of terms that are provably equal to it.

At first, the different algebras have been developed independently. Slowly, however, their close relationships have been understood and appreciated, and now a general theory can be provided and the different formalisms (CCS, CSP, ACP, etc. .) can be seen just as instances of the general approach. In this general approach, the main ingredients of a specific process algebra are:

1. A minimal set of carefully chosen operators capturing the relevant aspect of systems behavior and the way systems are composed in building process terms;
2. A transition system associated with each term via structural *operational semantics* to describe the evolution of all processes that can be built from the operators;
3. An equivalence notion that allow one to abstract from irrelevant details of systems descriptions.

Verification of concurrent system within the process algebraic approach is carried out either by resorting to behavioral equivalences for proving conformance of processes to specifications or by checking that processes enjoy properties described by some temporal logic formulae [28, 14]. In the former case, two descriptions of a given system, one very detailed and close to the actual concurrent implementation, the other more abstract (describing the sequences or trees of relevant actions the system has to perform) are provided and tested for equivalence. In the latter case, concurrent systems are specified as process terms while properties are specified as temporal logic formulae and model checking is used to determine whether the transition systems associated with terms enjoy the property specified by the formulae.

In the next section, many of the different operators used in process algebras will be described. By relying on the so called structural operational semantic (SOS) approach [37], it will be shown how labelled transition systems can be built and composed by using the different operators. Afterward, many behavioral equivalences will be introduced together with a discussion on the induced identifications and distinctions. Next, the three most popular process algebras will be described; for each of them a different approach (operational, denotational, algebraic) will be used. It will, however, be argued that in all cases the operational semantics plays a central rôle.

2 Process Operators and Operational Semantics

To define a process calculus, one starts with a set of uninterpreted action names (that might represent communication channels, synchronization actions, etc.) and with a set of basic processes that together with the actions are the building blocks for forming newer processes from existing ones. The operators are used for describing sequential, nondeterministic or parallel compositions of processes, for abstracting from internal details of process behaviors and, finally, for defining infinite behaviors starting from finite presentations. The operational semantics of the different operators is inductively specified through SOS rules: for each operator there is a set of rules describing the behavior of a system in terms of the behaviors of its components. As a result, each process term is seen as a component that can interact with other components or with the external environment.

In the rest of this section, most of the operators that have been used in some of the best known process algebras will be presented with the aim of showing the wealth of choices that one has when deciding how to describe a concurrent system or even when defining one's "personal" process algebra. A new calculus can, indeed, be obtained by a careful selection of the operators while taking into account their interrelationships with respect to the chosen abstract view of process and thus of the behavioral equivalence one has in mind.

A set of operators is the basis for building process terms. A labelled transition system (LTS) is associated to each term by relying on structural induction by providing specific rules in for each operator. Formally speaking, an LTS is a set of nodes (corresponding to process terms) and (for each action a in some set) a relation $\xrightarrow{a}$ between nodes, corresponding to processes transitions. Often LTSs have a distinguished node n_0 from which computations start; when defining the semantics of a process term the state corresponding to that term is considered as the initial state. To associate an LTS to a process term inference systems are used, where the collection of transitions is specified by means of a set of syntax-driven inference rules.

Inference Systems An inference system is a set of inference rule of the form:

$$\frac{p_1, \cdots, p_n}{q}$$

where $p_1, \cdots, p_n$ are the *premises* and q is the *conclusion*. Each rule is interpreted as an implication: if all premises are true then also the conclusion is true. Sometimes, rules are decorated with predicates and/or negative premises that specify when the rule is actually appliable.

A rule with an empty set of premises is called *axiom* and written as:

$$\frac{\qquad}{q}$$

Transition Rules In the case of an operational semantics the premises and the conclusions will be triples of the form (P, α, Q), often rendered as $P \overset{\alpha}{\rightarrow} Q$, and thus the rules for each operator op of the process algebras will be of the following form, where $\{i_1, \cdots, i_m\} \subseteq \{1, \cdots, n\}$ and $E'_i = E_i$ when $i \notin \{i_1, \cdots, i_m\}$:

$$\frac{E_{i_1} \xrightarrow{\alpha_1} E'_{i_1} \quad \cdots \quad E_{i_m} \xrightarrow{\alpha_m} E'_{i_m}}{op(E_1, \cdots, E_n) \overset{\alpha}{\rightarrow} C[E'_1, \cdots, E'_n]}$$

In the rule above the target term $C[\ ]$ indicates the new context in which the new subterms will be operating after the reduction and α represents the action performed by the composite system when some of the components perform actions $\alpha_1, \ldots, \alpha_m$. Sometimes, these rules are enriched with side conditions that determine their applicability. By imposing syntactic constraints on the form of the allowed rules, *rule formats* are obtained that can be used to establish results that hold for all process calculi whose transition rules respect the specific rule format.

A small number of SOS inference rules is sufficient to associate an LTS to each term of any process algebra. The set of rules is fixed once and for all. Given any process, the rules are used to derive its transitions. The transition relation of the LTS is the **least** one satisfying the inference rules. It is worth remarking that *structural induction* allows one to define the LTS of complex systems in terms of the behavior of their components.

Basic Actions An elementary action of a system represents the **atomic** (non-interruptible) abstract step of a computation that is performed by a system to move from one state to the next. Actions represent various activities of concurrent systems, like sending or receiving a message, updating a memory cell, synchronizing with other processes, ... In process algebras two main types of atomic actions are considered, namely *visible* or external actions and invisible or *internal* actions. In the sequel, visible actions will be denoted by Latin letters $a, b, c, \ldots$, invisible actions will be denoted by the Greek letter τ. Generic actions will be denoted by μ or other, possibly indexed, Greek letters. In the following, A will be used to denote the set of visible actions while A_τ will denote the collection of generic actions.

Basic processes Process algebras generally also include a null process (variously denoted as *nil*, 0, *stop*) which has no transition. It is inactive and its sole purpose is to act as the inductive anchor on top of which more interesting processes can be generated. The semantics of this process is characterized by the fact that there is no rule to define its transition: it has no transition.

Other basic processes are also used: *stop* denotes a deadlocked state, $\surd$ denotes, instead, successful termination.

$$\frac{}{\surd \xrightarrow{\surd} stop}$$

Sometimes, uninterpreted actions μ are considered basic processes themselves:

$$\frac{}{\mu \xrightarrow{\mu} \surd}$$

Sequential composition Operators for sequential composition are used to temporally order processes execution and interaction. There are two main operators for this purpose. The first one is *action prefixing*, $\mu.$-, that denotes a process that executes action μ and then behaves like the following process.

$$\frac{}{\mu.E \xrightarrow{\mu} E}$$

The alternative form of sequential composition is obtained by explicitly requiring *process sequencing* , - ; -, that requires that the first operand process be fully executed before the second one.

$$\frac{E \xrightarrow{\mu} E'}{E; F \xrightarrow{\mu} E'; F} \quad (\mu \neq \surd) \qquad \frac{E \xrightarrow{\surd} E' \qquad F \xrightarrow{\mu} F'}{E; F \xrightarrow{\mu} F'}$$

Nondeterministic composition The operators for nondeterministic choice are used to express alternatives among possible behaviors. This choice can be left to the environment (*external choice*) or performed by the process (*internal choice*) or can be mainly external but leaving the possibility to the process to perform an internal move to prevent some of the choices by the environment (*mixed choice*).

The rules for mixed choice are the ones below. They offer both visible and invisible actions to the environment; however, only the former kind of actions can be actually controlled.

$$\frac{E \xrightarrow{\mu} E'}{E + F \xrightarrow{\mu} E'} \qquad \frac{F \xrightarrow{\mu} F'}{E + F \xrightarrow{\mu} F'}$$

The rules for internal choice are very simple, they are just two axioms stating that a process $E \oplus F$ can silently evolve into one of its subcomponents.

$$\frac{}{E \oplus F \xrightarrow{\tau} E} \qquad \frac{}{E \oplus F \xrightarrow{\tau} F}$$

The rules for external choice are more articulate. This operator behaves exactly like the mixed choice in case one of the components executes a visible action, however, it does not discard any alternative upon execution of an invisible action.

$$\frac{E \xrightarrow{\alpha} E'}{E \,\square\, F \xrightarrow{\alpha} E'}\,(\alpha \neq \tau) \qquad \frac{F \xrightarrow{\alpha} F'}{E \,\square\, F \xrightarrow{\alpha} F'}\,(\alpha \neq \tau)$$

$$\frac{E \xrightarrow{\tau} E'}{E \,\square\, F \xrightarrow{\tau} E' \,\square\, F} \qquad \frac{F \xrightarrow{\tau} F'}{E \,\square\, F \xrightarrow{\tau} E \,\square\, F'}$$

Parallel composition Parallel composition of two processes, say E and F, is the key primitive distinguishing process algebras from sequential models of computation. Parallel composition allows computation in E and F to proceed simultaneously and independently. But it also allows interaction, that is synchronization and flow of information between E and F on a shared channel. Channels may be synchronous or asynchronous. In the case of synchronous channels, the agent sending a message waits until another agent has received the message. Asynchronous channels do not force the sender to wait. Here, only synchronous channels will be considered.

The simplest operator for parallel composition is *interleaving* , - ||| -, that aims at modeling the fact that two parallel processes can progress by alternating at any rate the execution of their actions.

$$\frac{E \xrightarrow{\mu} E'}{E|||F \xrightarrow{\mu} E'|||F} \qquad \frac{F \xrightarrow{\mu} F'}{E|||F \xrightarrow{\mu} E|||F'}$$

Another parallel operator is *binary parallel composition*, - | -, that not only models the interleaved execution of the actions of two parallel processes but also the possibility that the two partners synchronize whenever they are willing to perform complementary visible actions (below represented as a and $\overline{a}$). In this case, the visible outcome is a τ-action that cannot be seen by other processes that are acting in parallel with the two communication partners. This is the parallel composition used in CCS.

$$\frac{E \xrightarrow{\mu} E'}{E|F \xrightarrow{\mu} E'|F} \qquad \frac{F \xrightarrow{\mu} F'}{E|F \xrightarrow{\mu} E|F'} \qquad \frac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\overline{\alpha}} F'}{E|F \xrightarrow{\tau} E'|F'}(\alpha \neq \tau)$$

Instead of binary synchronization some process algebras, like CSP, make use of operators that permit *multiparty synchronization*, - |[L]| -. Some actions, those in L, are deemed to be synchronization actions and can be performed by a process only if all its parallel components can execute those actions at the same time.

$$\frac{E \xrightarrow{\mu} E'}{E \;|[L]|\; F \xrightarrow{\mu} E' \;|[L]|\; F}\,(\mu \notin L) \qquad \frac{F \xrightarrow{\mu} F'}{E \;|[L]|\; F \xrightarrow{\mu} E \;|[L]|\; F'}\,(\mu \notin L)$$

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{a} F'}{E \;|[L]|\; F \xrightarrow{a} E' \;|[L]|\; F'}\,(a \in L)$$

It is worth noting that the result of a synchronization, in this case, yields a visible action and that by setting the synchronization alphabet to $\emptyset$ the multiparty synchronization operator $|\emptyset|$ can be used to obtain pure interleaving, |||.

A more general composition is the *merge* operator, $-\ \|\ -$ that is used in ACP. It permits executing two process terms in parallel (thus freely interleaving their actions), but also allows for communication between its process arguments according to a *communication function* $\gamma : A \times A \rightarrow A$, that, for each pair of atomic actions a and b, produces the outcome of their communication $\gamma(a,b)$, a partial function that states which actions can be synchronized and the outcome of such a synchronization.

$$\frac{E \xrightarrow{\mu} E'}{E \parallel F \xrightarrow{\mu} E' \parallel F} \qquad \frac{F \xrightarrow{\mu} F'}{E \parallel F \xrightarrow{\mu} E \parallel F'} \qquad \frac{E \xrightarrow{a} E' \qquad F \xrightarrow{b} F'}{E \parallel F \xrightarrow{\gamma(a,b)} E' \parallel F'}$$

ACP has also another operator called *left merge*, - $\llfloor$ -, that is similar to $\parallel$ but requires that the first process to perform an (independent) action be the left operand.

$$\frac{E \xrightarrow{\mu} E'}{E \llfloor F \xrightarrow{\mu} E' \parallel F}$$

The ACP *communication merge*, - $|_c$ -, requires instead that the first action be a synchronization action.

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{b} F'}{E|_c F \xrightarrow{\gamma(a,b)} E' \parallel F'}$$

Disruption An operator that is between parallel and nondeterministic composition is the so called *disabling* operator, $-\ [>\ -$, that permits interrupting the evolution of a process. Intuitively, $E\ [>\ F$ behaves like E, but can be interrupted at any time by F, once E terminates F is discarded.

$$\frac{E \xrightarrow{\mu} E'}{E\ [>\ F \xrightarrow{\mu} E'\ [>\ F}\ (\mu \neq \surd) \qquad \frac{E \xrightarrow{\surd} E'}{E\ [>\ F \xrightarrow{\tau} E'} \qquad \frac{F \xrightarrow{\mu} F'}{E\ [>\ F \xrightarrow{\mu} F'}$$

Value Passing The above parallel combinators can be generalized to model not only synchronization but also exchange of values. As an example, below, the generalization of binary communication is presented.

There are complementary rules for sending and receiving values. The first axiom models a process willing to input a value and to base its future evolutions on it. The second axiom models a process that evaluates an expression (via the valuation function $val(e)$) and outputs the result.

$$\frac{}{a(x).E \xrightarrow{a(v)} E\{v/x\}}\ (v \text{ is a value}) \qquad \frac{}{\overline{a}\, e.E \xrightarrow{\overline{a}\ val(e)} E}$$

The next rule, instead, models synchronization between processes. If two processes, one willing to output and the other willing to input, are running in parallel, a synchronization can take place and the perceived action will just be a τ-action.

$$\frac{E \xrightarrow{\overline{a}\ v} E' \quad F \xrightarrow{a(v)} F'}{E|F \xrightarrow{\tau} E'|F'} \qquad \frac{E \xrightarrow{a(v)} E' \quad F \xrightarrow{\overline{a}\ v} F'}{E|F \xrightarrow{\tau} E'|F'}$$

In case the exchanged values are channels, this approach can be used to provide also models for mobile systems.

Abstraction Processes do not limit the number of connections that can be made at a given interaction point. But interaction points allow interference. For the synthesis of compact, minimal and compositional systems, the ability to restrict interference is crucial.

The *hiding* operator, $-/L$, hides (i.e., transforms into τ-actions) all actions in L to forbid synchronization on them. However, itallows the system to perform the transitions labelled by hidden actions.

$$\frac{E \xrightarrow{\mu} E'}{E/L \xrightarrow{\mu} E'/L} (\mu \notin L) \qquad \frac{E \xrightarrow{\mu} E'}{E/L \xrightarrow{\tau} E'/L} (\mu \in L)$$

The *restriction* operator, $-\backslash L$ is a unary operator that restricts the set of visible actions a process can perform. Thus, process $E\backslash L$ can perform only actions not in L. Obviously, invisible actions cannot be restricted.

$$\frac{E \xrightarrow{\mu} E'}{E \backslash L \xrightarrow{\mu} E' \backslash L} (\mu , \overline{\mu} \notin L)$$

The operator $[f]$, where f is a *relabelling* function from A to A, can be used to rename some of the actions a process can perform to make it compatible with new environments.

$$\frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$$

Modelling infinite behaviours The operations presented so far describe only finite interaction and are consequently insufficient for providing full computational power, in the sense of being able to model all computable functions. In order to reach full power, one certainly needs operators for modeling non-terminating behavior. Many operators have been introduced that allow finite descriptions of infinite behavior. However, it is important to to remark that most of them do not fit the formats used so far and cannot be defined by structural induction.

One of the most used is the construct *rec x.* -, well-known from the sequential world. If E is a process that contains the variable x, then *rec x. E* represents the process that behaves like E once all occurrences of x in E are replaced by *rec x. E*. In the rule below, that models the operational behaviour of a recursively defined process, the term $E[F/x]$ denotes exactly the above mentioned substitutions.

$$\frac{E[rec\, x.\, E/x] \xrightarrow{\mu} E'}{rec\, x.\, E \xrightarrow{\mu} E'}$$

The notation *rec x.E* for recursion sometimes makes the process expressions more difficult to parse and less pleasant to read. A suitable alternative is to allow for the (recursive) definition of some fixed set of constants, that can then be used as some sort of procedure calls inside processes. Assuming the existence of an environment (a set of process definitions)

$$\Gamma = \{X_1 \triangleq E_1, X_2 \triangleq E_2, \ldots, X_n \triangleq E_n\}$$

the operational semantics rule for a process variable becomes:

$$\frac{X \triangleq E \in \Gamma \quad E \xrightarrow{\mu} E'}{X \xrightarrow{\mu} E'}$$

Another operator used to describe infinite behaviors is the so called *bang* operator, ! −, or *replication*. Intuitively, $!E$ represents an unlimited number of instances of E running in parallel. Thus, its semantics is rendered by the following inference rule:

$$\frac{E|!\,E \xrightarrow{\mu} E'}{!E \xrightarrow{\mu} E'}$$

3 Three Process Algebras: CCS, CSP and ACP

A process algebra consists of a set of terms, an operational semantics associating LTS to terms and an equivalence relation equating terms exhibiting "similar" behavior. The operators for most process algebras are those described above. The equivalences can be traces, testing, bisimulation equivalences or variants thereof, possibly ignoring invisible actions.

Below, three of the most popular process algebras are presented. First the syntax, i.e., the selected operators, will be introduced, then their semantics will be provided by following the three different approaches outlined before: operational (for CCS), denotational (for CSP) and algebraic (for ACP). For CSP and ACP, the relationships between the proposed semantics and the operational one, to be used as a yardstick, will be mentioned. To denote the LTS associated to a generic CSP or ACP process p via the operational semantics, the notation $\mathbf{LTS}(p)$ will be used.

Reference will be made to specific behavioral equivalences over LTSs that consider as equivalent those systems that rely on different standing about which states of an LTS have to be considered equivalent. Three main criteria have been used to decide when two systems can be considered equivalent:
1. the two systems perform the same sequences of actions,
2. the two systems perform the same sequences of actions and after each sequence are ready to accept the same sets of actions,
3. the two systems perform the same sequences of actions and after each sequence exhibit, recursively, the same behavior.

These three different criteria lead to three groups of equivalences that are known as *traces* equivalences, *decorated-traces* equivalences (testing and failure equivalence), and *bisimulation-based* equivalences (strong bisimulation, weak bisimulation, branching bisimulation).

CCS: Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) is a process algebra introduced by Robin Milner around 1980. Its actions model indivisible communications between exactly two participants and the set of operators includes primitives for describing parallel composition, choice between actions and scope restriction. The basic semantics is operational and permits associating an LTS to each CCS term.

The set A of basic actions used in CCS consists of a set Λ, of labels and of a set $\overline{\Lambda}$ of complementary labels. A_τ denotes $A \cup \{\tau\}$. The syntax of CCS, used to generate all terms of the algebra, is the following:

$$P \;::=\; nil \mid x \mid \mu.P \mid P\backslash L \mid P[f] \mid P_1 + P_2 \mid P_1|P_2 \mid rec\, x.\, P$$

where $\mu \in Act_\tau$; $L \subseteq \Lambda$; $f : Act_\tau \to Act_\tau$; $f(\bar{\alpha}) = \overline{f(\alpha)}$ and $f(\tau) = \tau$. The above operators are taken from those presented in Section 2:

- the atomic process (nil),

- action prefixing ($\mu.P$),
- mixed choice ($+$),
- binary parallel composition ($|$),
- restriction ($P \backslash L$),
- relabelling ($P[f]$) and
- recursive definitions ($rec\ x.\ E$).

The operational semantics of the above operators is exactly the same as the one of those operators with the same name described before and it is thus not repeated here. CCS has been studied with bisimulation and testing semantics that are used to abstract from unnecessary details of the LTS associated to a term. Also denotational and axiomatic semantics for the calculus have been extensively studied. A denotational semantics in terms of so called *acceptance trees* has been proved to be in full agreement with the operational semantics abstracted according to testing equivalences. Different algebraic semantics have been provided that are based on sound and complete axiomatizations of bisimilarity, testing equivalence, weak bisimilarity and branching bisimilarity.

CSP: A Theory of Communicating Sequential Processes

The first denotational semantics proposed for CSP associates to each term just the set of the sequences of actions the term could induce. However, while suitable to model the sequences of interactions a process could have with its environment, this semantics is unable to model situations that could lead to deadlock. A new approach, basically denotational but with a strong operational intuition, was proposed next. In this approach, the semantics is given by associating a so called refusal set to each process. A refusal set is a set of failure pairs $\langle s, F \rangle$ where s is a finite sequence of visible actions in which the process might have been engaged and F is a set of action the process is able to reject on the next step. The semantics of the various operators is defined by describing the transformation they induce on the domain of refusal sets.

The meaning of processes is then obtained by postulating that two processes are equivalent if and only if they cannot be distinguished when their behaviors are observed and their reactions to a finite number of alternative possible synchronization is considered. Indeed, the association of processes to refusal sets is not one-to-one; the same refusal set can be associated to more than one process. A congruence is then obtained that equates processes with the same denotation.

The set of actions is a a finite set of labels, denoted by $\Lambda \cup \{\tau\}$. There is no notion of complementary action. The syntax of CSP is reported below and for the sake of simplicity only finite terms (no recursion) are considered:

$$E ::= \textsc{Stop} \mid skip \mid a \rightarrow E \mid E_1 \sqcap E_2 \mid E_1 \Box E_2 \mid E_1 \mid[L]\mid E_2 \mid E/a$$

- two basic processes: successful termination ($skip$), null process (Stop),
- action prefixing here denoted by $a \rightarrow E$,
- internal choice ($\oplus$) here denoted by $\sqcap$ and external choice ($\Box$),
- parallel composition with synchronization on a fixed alphabet ($\mid[L]\mid$, $L \subseteq \Lambda$),
- hiding ($/a$, an instance of the more general operator $/L$ with $L \subseteq \Lambda$).

Parallel combinators representing pure interleaving and parallelism with synchronization on the full alphabet can be obtained by setting the synchronization alphabet to $\emptyset$ or to Λ, respectively.

The denotational semantics of CSP compositionally associates a set of failure pairs to each CSP term generated by the above syntax. A function, $\mathcal{F}[\![-]\!]$, maps each CSP process (say P) to set of pairs (s, F) where s is one of the sequences of actions P may perform and F represents the set of actions that P can refuse after performing s. As anticipated, there is a strong correspondence between the denotational semantics of CSP and the operational semantics that one could define by relying on the one presented in the previous section for the specific operators.

- $\mathcal{F}[\![P]\!] = \mathcal{F}[\![Q]\!]$ if and only if $\mathbf{LTS}(P) \simeq_{test} \mathbf{LTS}(Q)$.

ACP: An Algebra of Communicating Processes

The methodological concern of ACP was to present "first a system of axioms for communicating processes ... and next study its models" ([6], p. 112). The equations are just a means to realize the real desideratum of abstract algebra, which is to abstract from the nature of the objects under consideration. In the same way as the mathematical theory of rings is about arithmetic without relying on a mathematical definition of number, ACP deals with process theory without relying on a mathematical definition of process.

In ACP a process algebra is any mathematical structure, consisting of a set of objects and a set of operators, like, e.g., sequential, nondeterministic or parallel composition, that enjoys a specific set of properties as specified by given axioms.

The set of actions Λ_τ consists of a finite set of labels $\Lambda \cup \{\tau\}$. There is no notion of complementary action. The syntax of ACP is reported below and for the sake of simplicity only finite terms (no recursion) are considered:

$$P \quad ::= \surd \mid \delta \mid a \mid P_1 + P_2 \mid P_1 {\cdot} P_2 \mid P_1 \| P_2 \mid P_1 \mathbin{\|\!\lfloor} P_2 \mid P_1 |_c P_2 \mid \partial_H(p)$$

- three basic processes: successful termination ($\surd$), null process, here denoted by δ, and atomic action (a),
- mixed choice,
- sequential composition (;), here denoted by $\cdot$,
- hiding ($\backslash H$ with $H \subseteq \Lambda$), here denoted by $\partial_H(-)$,
- three parallel combinators: merge ($\|$), left merge ($\|\!\lfloor$) and communication merge ($|_c$).

The system of axioms of ACP is presented as a set of formal equations, and some of the operators, e.g., left merge ($\|\!\lfloor$) have been introduced exactly for providing finite equational presentations. Below, the axioms relative to the terms generated by the above syntax are presented. Within the axioms, x and y denote generic ACP processes, and v and w denote generic actions.

(A1) $x + y = y + x$	(A2) $(x + y) + z = x + (y + z)$
(A3) $x + x = x$	(A4) $(x + y){\cdot}z = x{\cdot}z + y{\cdot}z$
(A5) $(x{\cdot}y){\cdot}z = x{\cdot}(y{\cdot}z)$	(A6) $x + \delta = x$
(A7) $\delta{\cdot}x = \delta$	

The set of axioms considered above induces an *equality relation*, denoted by =. A *model* for an axiomatization is a pair $\langle \mathcal{M}, \phi \rangle$, where $\mathcal{M}$ is a set and ϕ is a function (the unique isomorphism) that associates elements of $\mathcal{M}$ to ACP terms. This leads to the following definitions:

1. A set of equations is *sound* for $\langle \mathcal{M}, \phi \rangle$ if $s = t$ implies $\phi(s) = \phi(t)$;
2. A set of equations is *complete* for $\langle \mathcal{M}, \phi \rangle$ if $\phi(s) = \phi(t)$ implies $s = t$.

Any model of the axioms seen above is an ACP process algebra. The simplest model for ACP has as elements the equivalence classes induced by =, i.e. all ACP terms obtained starting from atomic actions, sequentialization and nondeterministic composition and mapping each term t to its equivalence class $[\![t]\!]$ as determined by =. This model is correct and complete and is known as the *initial model* for the axiomatization.

Different, more complex, models can be obtained by first using the SOS rules to give the operational semantics of the operators, building an LTS in correspondence of each ACP term and then using bisimulation to identify some of them. This construction leads to establishing a strong correspondence between the axiomatic and the operational semantics of ACP. Indeed, if we consider the language with the null process, sequential composition and mixed choice we have:

- Equality = as induced by (A1)-(A7) is *sound* relative to bisimilarity $\sim$, i.e., if $p = q$ then $\mathbf{LTS}(p) \sim \mathbf{LTS}(q)$;
- Equality = as induced by (A1)-(A7) is *complete* relative to bisimilarity $\sim$, i.e., if $\mathbf{LTS}(p) \sim \mathbf{LTS}(q)$ then $p = q$.

Similar results can be obtained when new axioms are added and weak bisimilarity or branching bisimilarity are used to factorize the LTS's.

4 Future Directions

The theory of process algebra is by now well developed, the reader is referred to [7] to learn about its developments since its inception in the late 1970's to the early 2000. Currently, in parallel with the exploitation of the developed theories in classic areas such as protocol verification and in new ones such as biological systems, there is much work going on concerning:

- extensions to model mobile, network aware systems;
- theories for assessing quantitative properties;
- techniques for controlling state explosion.

In parallel with this, much attention is dedicated to the development of software tools to support specification and verification of very large systems and to the development of techniques that permit controlling the state explosion phenomenon that arise as soon as one considers the possible configurations resulting from the interleaved execution of (even a small number of) processes.

Mobility and network awareness Much of the ongoing work is relative to the definition of theories and formalisms to naturally deal with richer classes of systems, like, e.g. mobile systems and network aware applications. The π-calculus [33] the successor of CCS, developed by Milner and co-workers with the aim of describing concurrent systems whose configuration may change during the computation has attracted much attention. It has laid the basis for research on process networks whose processes are mobile and the configuration of communication links is dynamic. It has also lead to the development of other calculi to support network aware programming: Ambient [13], Distributed π [25], Join [19], Spi [1], Klaim [9], ... There is still no unifying theory and the name process calculi is preferred to process algebras because the algebraic theories are not yet well assessed. Richer theories than LTS (Bi-graph [34], Tiles [20], ...) have been developed and are still under development to deal with the new dimensions considered with the new formalisms.

Quantitative Extensions Formalisms are being enriched to consider not only qualitative properties, like correctness, liveness or safety, but also properties related to performance and quality of service. There has been much research to extend process algebra to deal with a quantitative notion of time and probabilities and integrated theories have been considered. Actions are enriched with information about their duration and formalisms extended in this way are used to compare systems relatively to their speed. For a comprehensive description of this approach, the reader is referred to [4]. Extensions have been considered also to deal with systems that in their behavior depend on continuously changing variables other than time (*hybrid systems*). In this case, systems descriptions involve differential algebraic equations, and connections with dynamic control theory are very important. Finally, again with the aim of capturing quantitative properties of systems and of combining functional verification with performance analysis, there have been extensions to enrich actions with rates representing the frequency of specific events and the new theories are being (successfully) used to reason about system performance and system quality.

Tools To deal with non-toy examples and apply the theory of process algebras to the specification and verification of real systems, tool support is essential. In the development of tools, LTSs play a central rôle. Process algebra terms are used to obtain LTSs by exploiting operational semantics and these structures are then minimized, tested for equivalence, model checked against formulae of temporal logics, ... One of the most known tools for process algebras is CADP (Construction and Analysis of Distributed Processes) [21]: together with minimizers and equivalence and model checkers it offers many others functionalities ranging from step-by-step simulation to massively parallel model checking. CADP has been employed in an impressive number of industrial projects. CWB (Concurrency Workbench) [35] and CWB-NC (Concurrency Workbench New Century) [15] are other tools that are centered on CCS, bisimulation equivalence and model checking. FDR (Failures/Divergence Refinement) [40] is a commercial tool for CSP that has played a major role in driving the evolution of CSP from a blackboard notation to a concrete language. It allows the checking of a wide range of correctness conditions, including deadlock and livelock freedom as well as general safety and liveness properties. TAPAs (Tool for the Analysis of Process Algebras) [12] is a recently developed software to support teaching of the theory of process algebras; it maintains a consistent double representation as term and as graph of each system. Moreover, it offers tools for the verification of many behavioural equivalences, possibly with counterexamples, minimization, step-by-step execution and model checking. TwoTowers, is instead a versatile tool for the functional verification, security analysis, and performance evaluation of computer, communication and software systems modeled with the stochastic process algebra EMPA [4]. μCRL [23] is a toolset that offers an appropriate treatment of data and relies also on theorem proving. Moreover, it make use of interesting techniques for visualizing large LTSs.

5 Relationships to Other Models of Concurrency

In a private communication, in 2009, Robin Milner, one of the founding fathers of process algebras, wrote:

> The concept of process has become increasingly important in computer science in the last three decades and more. Yet we still don't agree on what a process is. We probably agree that it should be an equivalence class of interactive agents, perhaps concurrent, perhaps non-deterministic.

This quote summarizes the debate on possible models of concurrency that has taken place during the last thirty years and has been centered on three main issues:

- interleaving vs true concurrency;
- linear-time vs branching-time;
- synchrony vs asynchrony.

Interleaving vs true concurrency The starting point of the theory of process algebras has been automata theory and regular expressions and the work on the algebraic theory of regular expressions as terms representing finite state automata [16] has significantly influenced its developments. Given this starting point, the underlying models of all process algebras represent possible concurrent executions of different programs in terms of the nondeterministic interleaving of their sequential behaviors. The fact that a system is composed by independently computing agents is ignored and behaviors are modeled in terms of purely sequential patterns of actions. It has been demonstrated that many interesting and important properties of distributed systems may be expressed and proved by relying on interleaving models. However, there are situations in which it is important to keep the information that a system is composed of the independently computing components. This possibility is offered by the so-called non-interleaving or true-concurrency models, with Petri nets [39] as the prime example. These models describe not only temporal ordering of actions but also their causal dependences. Non-interleaving semantics of process algebras have also been provided, see e.g. [36].

Linear-time vs branching-time Another issue, again ignored in the initial formalization of regular expressions, is how the concept of nondeterminism in computations is captured. Two possible views regarding the nature of nondeterministic choice induce two types of models giving rise to the linear-time and branching-time dichotomy. A linear-time model expresses the full nondeterministic behavior of a system in terms of the set of possible runs; time is treated as if each moment there is a unique possible future. Major examples of structures used to model sets of runs are Hoare traces (captured also by traces equivalence) for interleaving models [26], and Mazurkiewicz traces [30] and Pratt's pomsets [38] for non-interleaving models. The branching-time model is the main one considered in process algebras and considers the set of runs structured as a computation tree. Each moment in time may split into various possible futures and semantic models are computation trees. For non interleaving models, event structures [44] are one of the best known models taking into account both nondeterminism and true concurrency.

Synchrony vs asynchrony There are two basic approaches to describing interaction between a sender and a receiver of a message (signal), namely synchronous and asynchronous interaction. In the former case, before proceeding, the sender has to make sure that a receiver is ready. In the latter case, the sender leaves track of its action but proceeds without any further waiting. The receiver has to wait in both cases. Process algebras are mainly synchronous, but asynchronous variants have been recently proposed and are receiving increasing attention. However, many other successful asynchronous models have been developed. Among these, it is important to mention Esterel [8], a full fledged programming language that allows the simple expression of parallelism and preemption and is very well suited for control-dominated model designs; Actors [3], a formalism that does not necessarily records messages in buffers and puts no requirement on the ordering of message delivery; Linda [22], a model of coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual, associative memory; and, to conclude, Klaim [17], a distributed variant of Linda with a strong process algebraic flavor.

6 Bibliographic Notes and Further Reading

A number of books describing the different process algebras can be consulted to obtain deeper knowledge of the topics sketched here. Unfortunately most of them are concentrating only on one of the formalisms rather than on illustrating the unifying theories.

CCS The seminal book on CCS is [31], in which sets of operators equipped with an operational semantics and the notion of observational equivalence have been presented for the first time. The, by now, classical text book on CCS and bisimulation is [32]. A very nice, more recent, book on CCS and the associated Hennessy Milner Modal Logic is [29]; it also presents timed variants of process algebras and introduces models and tools for verifying properties also of this new class of systems.

CSP The seminal book on CSP is [27], where all the basic theory of failure sets is presented together with many operators for processes composition and basic examples. In [41], the theory introduced in [27] is developed in full detail and a discussion on the different possibilities to deal with anomalous infinite behaviors is considered together with a number of well thought examples. Moreover the relationships between operational and denotational semantics are fully investigated. Another excellent text book on CSP is [43] that also considers timed extensions of the calculus.

ACP The first published book on ACP is [5], where the foundations of algebraic theories are presented and the correspondence between families of axioms, and strong and branching bisimulation are thoroughly studied. This is a book intended mainly for researchers and advanced students, a gentle introduction to ACP can be found in [18].

Other Approaches Apart from these books, dealing with the three process algebras presented in these notes, it is also worth mentioning a few more books. LOTOS, a process algebra that was developed and standardized within ISO for specifying and verifying communication protocols, is the central calculus of a recently published book [10] that discusses also the possibility of using different equivalences and finer semantics for the calculus. A very simple and elegant introduction to algebraic, denotational and operational semantics of processes, that studies in detail the impact of the testing approach on a calculus obtained from a careful selection of operators from CCS and CSP, can be found in [24]. The text [42] is **the** book on the π-calculus. For studying this calculus, the reader is, however, encouraged to consider first reading [33].

References

1. M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.
2. L. Aceto and A.D. Gordon editors. *Proceedings of the Workshop "Essays on Algebraic Process Calculi" (APC 25), Electronic Notes in Theoretical Computer Science vol. 162*. Elsevier, 2005.
3. G. Agha. *Actors: A Model of Concurrent Computing in Distributed Systems*. MIT Press, 1986.
4. A. Aldini, M. Bernardo, and F. Corradini. *A Process Algebraic Approach to Software Architecture Design*. Springer, 2010.
5. J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.
6. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
7. J.A. Bergstra, A. Ponse, and S.A. Smolka editors. *Handbook of Process Algebra*. Elsevier, 2001.
8. Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.
9. L. Bettini, V. Bono, R. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice, 2003.
10. H. Bowman and R. Gomez. *Concurrency Theory: calculi and Automata for Modelling Untimed and Timed Concurrent Systems*. Springer, 2006.
11. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
12. F. Calzolai, R. De Nicola, M. Loreti, and F. Tiezzi. Tapas: A tool for the analysis of process algebras. *Petri Nets and Other Models of Concurrency*, 1:54–70, 2008.
13. L. Cardelli and A.D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

14. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of Logic of Programs*, pages 52–71. Springer–Verlag, 1982.
15. R. Cleaveland and S. Sims. The ncsu concurrency workbench. In *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397. Springer–Verlag, 1996.
16. J.H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.
17. R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.
18. W. Fokkink. *Introduction to Process Algebra.* Springer–Verlag, 2000.
19. C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2000.
20. F. Gadducci and U. Montanari. The tile model. In G. Plotkin, C. Stirling, and Eds. M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 133–166. MIT Press, 2000.
21. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. In *European Association for Software Science and Technology (EASST)*, volume 4 of *Newsletter*, pages 13–24, 2002.
22. D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992.
23. J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. Analysis of distributed systems with mcrl2. In *Process Algebra for Parallel and Distributed Processing (M. Alexander and W. Gardner, eds.)*, pages 99–128. Chapman Hall, 2009.
24. M. Hennessy. *Algebraic theory of Processes*. The MIT Press, 1988.
25. M. Hennessy. *A Distributed Pi-Calculus*. CAmbridge University Press, 2007.
26. C. A. R. Hoare. A calculus of total correctness for communicating processes. *Sci. Comput. Program.*, 1(1-2):49–72, 1981.
27. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.
28. D. Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
29. K. G. Larsen L.Aceto, A. Ingolfsdottir and J. Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007.
30. A. Mazurkiewicz. Introduction to trace theory. In G. Rozenberg V. Diekert, editor, *The Book of Traces*, pages 3–67. World Scientific, Singapore, 1995.
31. R. Milner. *A Calculus of Communicating Systems.*, volume 92 of *Lecture Notes in Computer Science*. Springer–Verlag, 1980.
32. R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
33. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
34. R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
35. F. Moller and P. Stevens. Edinburgh Concurrency Workbench user manual. Available from `http://homepages.inf.ed.ac.uk/perdita/cwb/`.
36. E.-R. Olderog. *Nets, Terms and Formulas*. Cambridge University Press, 1991.
37. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
38. V. Pratt. Modeling concurrency with partial orders. *International Journal on Parallel Processing*, 1:3371, 1986.
39. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.
40. A.W. Roscoe. Model-checking csp. In *A Classical Mind: essays in Honour of C.A.R. Hoare.* Prentice-Hall, Inc., 1994.
41. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, Inc., 1998.
42. D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
43. S.A. Schneider. *Concurrent and Real Time Systems: the CSP Approach*. John Wiley, 1999.
44. G. Winskel. An introduction to event structures. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency - Rex Workshop*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397. Springer, 1989.