Languages for Concurrency

Catuscia Palamidessi, Frank Valencia

LANGUAGES FOR CONCURRENCY

Catuscia Palamidessi
INRIA and LIX, École Polytechnique
catuscia@lix.polytechnique.fr

Frank D. Valencia
CNRS and LIX, École Polytechnique
frank.valencia@lix.polytechnique.fr

Abstract

This essay offers an overview of basic aspects and central development in Concurrency Theory based on formal languages. In particular, it focuses on the theory of Process Calculi.

1 Introduction

Concurrency is concerned with the fundamental aspects of systems consisting of multiple computing agents, usually called *processes*, that interact among each other. This covers a vast variety of systems which nowadays, due to technological advances such as the Internet, programmable robotic devices and mobile computing, most people can easily relate to. Some examples are:

- *Message-passing* communication based systems: Agents interact by exchanging messages. For instance, e-mail communication on the Internet, or robot point-to-point exchange of messages via infra-red communication.
- *Shared-Variables* communication based systems: Agents communicate by posting and reading information from a central location. For instance, reading and posting information on a server as in an Internet newsgroup. In the context of co-operative robotic devices, there can be a central control, usually a PC, on which the robots can post and read information (e.g., their relative positions).
- *Synchronous* systems: As opposed to *asynchronous* systems, in synchronous systems, agents need to synchronize with one another. In Internet telephony services the caller and the callee's terminal need to synchronize to establish communication. In systems of mobile robotic devices, robots most certainly need to synchronize, e.g., to avoid bumping into each other. An example of asynchrony is SMS communication on mobile phones.

- *Reactive* systems: Involve systems that maintain an ongoing interaction with their environment. For instance, reservation systems and databases on the Internet. Co-operative robotic devices are typically programmed to react to their surroundings, e.g., going backwards whenever a touch sensor is pressed.
- *Timed* systems: Systems in which the agents are constrained by temporal requirements. For example, browser applications are constrained by timer-based exit conditions (i.e., *time-outs*) for the case in which a server cannot be contacted. E-mailer applications can be required to check for messages every k time units. Also, robots can be programmed with time-outs (e.g., to wait for some signal) and with timed instructions (e.g., to go forward for 42 time units).
- *Mobile* systems: Agents can change their communication links. This is the essence of mobile computing devices. For example, portable computers can connect to the Internet from different locations. Robotic devices also exhibit mobility since, as they are on the move they may change their communication configuration. E.g., robots, which could initially communicate with one another, may sometime later be too far away to continue to do so.
- *Secure* systems: Systems in which critical resources of some sort (e.g., secret information) must not be accessed, misused or modified by unwanted agents. Credit card usage on the Internet is now a common practice involving secure systems. To a more physical level, one now hears of robotic security systems [9] which involve mobile devices that are strategically programmed to patrol, detect intruders and respond accordingly.

The above are but a few representatives of systems exhibiting concurrency, often referred to as *concurrent* systems. Furthermore, they can be combined to give rise to very complex concurrent systems; for example the Internet itself.

1.1 Problem: Reasoning about Concurrency

The previous examples illustrate the practical relevance, complexity and ubiquity of concurrent systems. It is therefore crucial to be able to describe, analyze and, in general, reason about concurrent behavior. This reasoning must be precise and reliable. Consequently, it ought to be founded upon mathematical principles in the same way as the reasoning about the behavior of sequential programs is founded upon logic, domain theory and other mathematical disciplines.

Nevertheless, giving mathematical foundations to concurrent computation has become a serious challenge for computer science. Traditional mathematical models of (sequential) computation based on functions from inputs to outputs no longer apply. The crux is that concurrent computation, e.g., in a reactive system, is seldom expected to terminate, it involves constant interaction with the environment, and it is *non-deterministic* owing to unpredictable interactions among agents.

1.2 Solution: Models of Concurrency

Computer science has therefore taken up the task of developing *models*, conceptually different from those of sequential computation, for the precise understanding of the behavior of concurrent systems. Such models, as other scientific models of reality, are expected to satisfy the following criteria:

- They must be *simple*, i.e., based upon few basic principles.
- They must be *expressive*, i.e., capable of capturing interesting real-world situations.
- They must be *formal*, i.e., founded upon mathematical principles.
- They must provide *techniques* to allow reasoning about their particular focus.

In order to develop a model of concurrency one could suggest the following general strategy: Seize upon a few pervasive aspects of concurrency (e.g., synchronous communication), make them the focus of a model, and then submit the model to the above criteria. This strategy can be claimed to have been involved in the development of a mature collection of models for various aspects of concurrency. Some representatives of this collection are mentioned next.

Representative models for synchronous communication. Some of the most mature and well-known models of concurrency are process calculi like Milner's CCS [16], Hoare's CSP [12], and ACP (developed by Bergstra and Klop [4] and also by Baeten [6]). The common focus of these models is synchronous communication.

Process calculi treat processes much like the λ-calculus treats computable functions. They provide a language in which the structure of *terms* represents the structure of processes together with an *operational semantics* to represent computational steps. For example, the term $P \parallel Q$, which is built from P and Q with the *constructor* $\parallel$, represents the process that results from the parallel execution of those represented by P and Q. An operational semantics may dictate that if P can evolve into P' in a computational step then $P \parallel Q$ can also evolve into $P' \parallel Q$ in a computational step.

An appealing feature of process calculi is their *algebraic* treatment of processes. The constructors are viewed as the *operators* of an algebraic theory whose equations and inequalities among terms relate process behavior. For instance, the construct $\parallel$ can be viewed as a commutative operator, hence the equation $P \parallel Q \equiv Q \parallel P$ states that the behavior of the two parallel compositions are the same. Because of this algebraic emphasis, these calculi are often referred to as *process algebras*.

A representative model for true-concurrency. Another important model of concurrency is Petri Nets [23]. The focus of Petri Nets is the simultaneous occurrence of actions (i.e., *true concurrency*). The theory of Petri Nets, which was the first well-established theory of concurrency, is an elegant generalization

of classic automata theory in which the concept of concurrently occurring transitions can be expressed.

1.3 Model Extensions

Science has made progress by extending well established theories to capture new and wider phenomena. For instance, computability theory was initially concerned only with functions on the natural numbers but it was later extended to deal with functions on the reals [11]. Also, classical logic was extended to various modal logics to study reasoning involving modalities such as possibility, necessity and temporal progression. Another example of great relevance is automata theory, initially confined to finite sequences, but later generalized to reason about infinite ones as in Büchi automata theory [5].

Similarly, several mature models of concurrency have been extended to treat additional issues. These extensions should not come as a surprise since the field is indeed large and subject to the advents of new technology.

One example of these additional issues is the notions of mobility and security which now pervade the informational world; none of the representative models mentioned above dealt with these notions. It was later found that a CCS extension, the π-calculus [18], could treat mobility in a very satisfactory way. A further extension, the spi-calculus [1], was also designed to model security.

Another prominent example is the notion of *time*. This notion not only is a fundamental concept in concurrency but also in science at large. Just like modal extensions of logic for temporal progression study time in logic reasoning, theories of concurrency were extended to study time in concurrent activity. For instance, neither CCS, CSP nor Petri Nets, in their basic form, were concerned with temporal behavior but they all have been extended to incorporate an explicit notion of time, leading for instance Timed CCS [33], Timed CSP [28], Timed ACP [3] and Timed Petri Nets [34].

2 The Theory of Process Calculi

This section describes some fundamental concepts from process calculi. We do not intent to give an in-depth review of these calculi (the interested reader is referred to [17]), but rather to describe those issues which influenced their development.

There are many different process calculi in the literature mainly agreeing in their emphasis upon algebra. The main representatives are CCS [16] , CSP [12] and the process algebra ACP [4, 6]. The distinctions among these calculi arise from issues such as the process constructions considered (i.e., the language of processes), the methods used for giving meaning to process terms (i.e. the semantics), and the methods to reason about process behavior (e.g., process equivalences or process logics). Some other issues addressed in the theory of these calculi are their expressive power, and analysis of their behavioral equivalences. In what follows we discuss some of these issues briefly.

2.1 The Language of Processes

A common feature of the languages of process calculi is that they pay special attention to economy. That is, there are few operators or combinators, each one with a distinct and fundamental role. Process calculi usually provide the following combinators:

- *Action*, for representing the occurrence of atomic actions.
- *Product*, for expressing the parallel composition.
- *Summation*, for expressing alternate course of computation.
- *Restriction* (or *Hiding*), for delimiting the interaction of processes.
- *Recursion*, for expressing infinite behavior.

A process language. For the purposes of the exposition of the next sections we shall define a basic process language which exemplifies the above.

We presuppose an infinite set $\mathcal{N}$ of *names* $a, b, \ldots$. and then introduce a set of *co-names* $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$ disjoint from $\mathcal{N}$. The set of *labels*, ranged over by l and l', is $\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}}$. The set of *actions Act*, ranged over by the boldface symbols **a** and **b** extends $\mathcal{L}$ with a new symbol τ. The action τ is said to be the *silent* (*internal* or *unobservable*) action. The actions a and $\overline{a}$ are thought of as being *complementary*, so we decree that $\overline{\overline{a}} = a$. The syntax of processes is given by:

$$P, Q, \ldots ::= 0 \mid \mathbf{a}.P \mid P + Q \mid P \parallel Q \mid P\backslash a \mid A\langle b_1, \ldots, b_n\rangle$$

Intuitive Description. The intuitive meaning of the process terms is as follows. The process 0 does nothing. $\mathbf{a}.P$ is the process which performs an atomic action **a** and then behaves as P. The summation $P + Q$ is a process which may behave as either P or Q. $P \parallel Q$ represents the parallel composition of P and Q. Both P and Q can proceed independently but they can also synchronize if they perform complementary actions. The restriction $P\backslash a$ behaves as P except that it cannot perform the actions a or $\overline{a}$. The names a and $\overline{a}$ are said to be *bound* in $P\backslash a$. $A\langle b_1, \ldots, b_n\rangle$ denotes the invocation to a unique recursive definition of the form $A(a_1, \ldots, a_n) \stackrel{\texttt{def}}{=} P_A$ where all the non-bound names of process P_A are in $\{a_1, \ldots, a_n\}$. Obviously P_A may contain invocations to A. The process $A\langle b_1, \ldots, b_n\rangle$ behaves as $P_A[b_1, \ldots, b_n/a_1, \ldots, a_n]$, i.e., P_A with each a_i replaced by b_i – with renaming of bound names wherever necessary to avoid captures.

2.2 Semantics of Processes

The methods by which process terms are endowed with meaning may involve at least three approaches: *operational*, *denotational* and *algebraic* semantics. Traditionally, CCS and CSP emphasize the use of the operational and denotational method, respectively, whilst the emphasis of ACP is upon the algebraic method.

Operational semantics. The methods was pioneered by Plotkin in his Structural Operational Semantics (SOS) work [24, 25, 26]. An operational semantics interprets a given process term by using transitions (labeled or not) specifying its computational steps. A labeled transition $P \xrightarrow{\mathbf{a}} Q$ specifies that P performs $\mathbf{a}$ and then behaves as Q. The relations $\xrightarrow{\mathbf{a}}$ are defined to be the smallest which obey the rules in Table 1. In these rules the transition below the line is to be inferred from those above the line.

$$\text{ACT}\ \frac{}{\mathbf{a}.P \xrightarrow{\mathbf{a}} P}$$

$$\text{SUM}_1\ \frac{P \xrightarrow{\mathbf{a}} P'}{P + Q \xrightarrow{\mathbf{a}} P'} \qquad \text{SUM}_2\ \frac{Q \xrightarrow{\mathbf{a}} Q'}{P + Q \xrightarrow{\mathbf{a}} Q'}$$

$$\text{COM}_1\ \frac{P \xrightarrow{\mathbf{a}} P'}{P \parallel Q \xrightarrow{\mathbf{a}} P' \parallel P} \qquad \text{COM}_2\ \frac{Q \xrightarrow{\mathbf{a}} Q'}{P \parallel Q \xrightarrow{\mathbf{a}} P \parallel Q'}$$

$$\text{COM}_3\ \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

$$\text{RES}\ \frac{P \xrightarrow{\mathbf{a}} P'}{P\backslash a \xrightarrow{\mathbf{a}} P'\backslash a}\ \text{if } \mathbf{a} \neq a \text{ and } \mathbf{a} \neq \overline{a}$$

$$\text{REC}\ \frac{P_A[b_1, \ldots, b_n/a_1, \ldots, a_n] \xrightarrow{\mathbf{a}} P'}{A\langle b_1, \ldots, b_n\rangle \xrightarrow{\mathbf{a}} P'}\ \text{if } A(a_1, \ldots, a_n) \stackrel{\texttt{def}}{=} P_A$$

Table 1: An operational semantics for a process calculus.

The rules in Table 1 are easily seen to realize the intuitive description of processes given in the previous section. Let us describe some. The rules SUM_1 and SUM_2 say that the first action of $P + Q$ determines which alternative is selected, the other is discarded. The rules for composition COM_1 and COM_2 describe the concurrent performance of P and Q. The rule COM_3 describes a synchronizing communication between P and Q. For recursion, the rule REC says that the actions of (an invocation) $A\langle b_1, \ldots, b_n\rangle$ are just those that can be inferred by replacing every a_i with b_i in (the definition's body) P_A where $A(a_1, \ldots, a_n) \stackrel{\texttt{def}}{=} P_A$.

Behavioral equivalences. Having defined the operational semantics, we can now introduce some typical notions of process equivalence. Here we shall recall *trace*, *failures* and *bisimilarity* equivalences. Although these equivalences can be defined for both CSP and CCS, traditionally the first two are associated with CSP and the last one is associated with CCS.

We need a little notation: The empty sequence is denoted by ϵ. Given a sequence of actions $s = \mathbf{a}_1.\mathbf{a}_2.\ldots \in Act^*$, define $\stackrel{s}{\Longrightarrow}$ as

$$(\stackrel{\tau}{\longrightarrow})^* \stackrel{\mathbf{a}_1}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^* \ldots (\stackrel{\tau}{\longrightarrow})^* \stackrel{\mathbf{a}_n}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^*$$

Notice that $\stackrel{\epsilon}{\Longrightarrow} = \stackrel{\tau}{\longrightarrow}^*$. We use $P \stackrel{s}{\Longrightarrow}$ to mean that there exists a P' s.t., $P \stackrel{s}{\Longrightarrow} P'$ and similarly for $P \stackrel{s}{\longrightarrow}$.

- *Trace equivalence*. This equivalence is perhaps the simplest of all. Intuitively, two processes are deemed trace equivalent if and only if they can perform exactly the same sequences of non-silent (or observable) actions. Formally, we say that P and Q are *trace equivalent*, written $P =_T Q$, if for every $s \in \mathcal{L}^*$,

  $$P \stackrel{s}{\Longrightarrow} \text{ iff } Q \stackrel{s}{\Longrightarrow}.$$

  A drawback of $=_T$ is that it is not sensitive to deadlocks. For example, let $P_1 = a.b.0 + a.0$ and $Q_1 = a.b.0$. Notice that $P_1 =_T Q_1$ but unlike Q_1, after doing a, P_1 can reach a state in which it cannot perform any action, i.e., a *deadlock*.

- *Failures equivalence*. This equivalence is more discriminating (stronger or finer) than trace equivalence. In particular it is sensitive to deadlocks.

  A *failure* is a pair (s, L) where $s \in \mathcal{L}^*$ (called a trace) and L is a set of labels. Intuitively, (s, L) is a failure of P if P can perform a sequence of observable actions s evolving into a P' in which no action from $L \cup \{\tau\}$ can be performed.

  Formally, we say that (s, L) is a *failure of* P if there exists P' such that

  $$(1)\ P \stackrel{s}{\Longrightarrow} P',\ (2)\ P' \stackrel{\tau}{\not\longrightarrow} \text{ and } (3) \text{ for all } l \in L,\ P' \stackrel{l}{\not\longrightarrow}.$$

  We then say that P and Q are *failures-equivalent*, written $P =_F Q$, iff they posses the same failures.

  Notice that $=_F \subseteq =_T$ as a trace is part of a failure. To see the strict inclusion, notice that for the trace equivalent processes P_1 and Q_1 given in the previous point, we have $P_1 \neq_F Q_1$ as P_1 has the failure $(a, \{b\})$ but Q_1 does not. Another interesting example is given by the processes $P_2 = a.(b.0 + c.0)$ and $Q_2 = a.b.0 + a.c.0$. They have the same traces, however $P_2 \neq_F Q_2$ since Q_2 has the failure $(a, \{c\})$ but P_2 does not.

- *Bisimilarity*. Here we first recall the strong version of the equivalence. Intuitively, P and Q are strongly bisimilar if whenever P performs an action $\mathbf{a}$ evolving into P' then Q can also perform $\mathbf{a}$ and evolve into a Q' strongly bisimilar to P', and similarly with P and Q interchanged.

The above intuition can be formalized as follows. A symmetric relation B between process terms is said to be a strong *bisimulation* iff for all $(P, Q) \in B$,

$$\text{if } P \xrightarrow{\mathbf{a}} P' \text{ then for some } Q',\ Q \xrightarrow{\mathbf{a}} Q' \text{ and } (P', Q') \in B.$$

We say that P is *strongly bisimilar* to Q, written $P =_{SB} Q$ iff there exists a strong bisimulation containing the pair (P, Q).

A weaker version of strong bisimilarity, called *weak bisimilarity* or simply *bisimilarity*, abstracts away from silent actions. Bisimilarity can be obtained by replacing the transitions $\xrightarrow{\mathbf{a}}$ above with the (sequences of observable) transitions $\overset{s}{\Longrightarrow}$ where $s \in \mathcal{L}^*$. We shall use $=_B$ to stand for (weak) bisimilarity. Notice that $P =_B \tau.P$ but $P \neq_{SB} \tau.P$.

Bisimilarity is more discriminating than trace equivalence. It is easy to see that $=_B \subseteq =_T$. The usual example to see the strict inclusion is P_2 and Q_2 as given above. Also, bisimilarity is more discriminating than failures equivalence wrt the *branching* behavior (i.e., nondeterminism); take $P_3 = a.(b.c.0 + b.d.0)$ and $P_3 = a.b.c.0 + a.b.d.0$; they have the same failures but one can verify that $P_3 \neq_B Q_3$. However, failures equivalence is more discriminating than bisimilarity wrt *divergence* (i.e., the execution of infinite sequences of silent actions). Notice that the divergent process **Div**, with $\mathbf{Div} \overset{\text{def}}{=} \tau.\mathbf{Div}$, is bisimilar to the non-divergent $\tau.0$, however $\mathbf{Div} \neq_F \tau.0$ since $\tau.0$ has the failure $(\epsilon, \emptyset)$ but **Div** does not.

Denotational Semantics. The method was pioneered by Strachey and provided with a mathematical foundation by Scott. A denotational semantics interprets processes by using a function $[\![.]\!]$ which maps them into a more abstract mathematical object (typically, a structured set or a category). The map $[\![.]\!]$ is *compositional* in that the meaning of processes is determined from the meaning of its sub-processes.

A strategy for defining denotational semantics advocated in works such as [13] involves the identification of what can be observed of a process; what behavior is deemed relevant (e.g., failures, traces, divergence, deadlocks). A process is then equated with the set of observations that can be made of it. For example, if the observation is the traces of processes, the denotation of the prefix construct $\mathbf{a}.P$ can be defined as

$$[\![\mathbf{a}.P]\!] = \{\epsilon\} \cup \{\mathbf{a}.s \in \mathcal{L}^* \mid s \in [\![P]\!]\}$$

and the denotation of the summation can be defined as

$$[\![P + Q]\!] = [\![P]\!] \cup [\![Q]\!].$$

It easy to see that these denotations realize the operational intuition of traces; any trace of $\mathbf{a}.P$ is either empty or it starts with $\mathbf{a}$ followed by a trace of P; any trace of $P + Q$ is either a trace of P or one of Q. Note that the compositional nature is

illustrated by stating the denotations of $\mathbf{a}.P$ and $P + Q$ in terms of those of P and Q.

Once the denotation has been defined one may ask whether it is in complete agreement with a corresponding operational notion. For example, for the trace denotation one would like the following correspondence wrt the operational notion of trace equivalence,

$$[\![P]\!] = [\![Q]\!] \text{ iff for all contexts } C,\ C[P] =_T C[Q]$$

(A *context* is an expression with a hole [.] such that placing a process in the hole produces a well-formed process term, e.g., if $C = R \parallel [.]$ then $C[P] = R \parallel P$.) If a denotational-operational agreement like the one above can be proven, we say that the denotation is *fully-abstract* [15] wrt the chosen operational notion.

Denotational semantics are more abstract than the operational ones in that they generally distant themselves from any specific implementation. However, the operational semantics approach is, in some informal sense, more elemental in that when developing a denotational semantics one usually has an operational semantics in mind.

Algebraic semantics. This method has been advocated by Baeten and Weijland [6] as well as Bergstra and Klop [4]. An algebraic semantics attempts to give meaning by stating a set of laws (or axioms) equating process terms. The processes and their operations are then interpreted as structures that obey these laws. As remarked by Baeten and Weijland [6], the algebraic approach answers the question "What is a process?" with a seemingly circular answer: "A process is something that obeys a certain set of axioms...for processes".

As an example consider the following axioms for parallel composition:

$$P \parallel 0 \equiv P,\quad P \parallel Q \equiv Q \parallel P,\quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$$

In other words parallel composition is seen as a commutative, associative operator with 0 being the unit. Notice that the above axioms basically equate processes that are the same except for irrelevant syntactic differences, thus one may expect that any reasonable notion of equivalence validates them. But consider the following distribution axiom

$$\mathbf{a}.(P + Q) \equiv \mathbf{a}.P + \mathbf{a}.Q$$

This axiom is valid if we are content with trace equivalence, but not in general (e.g., it does not hold for failures equivalence or bisimilarity).

Given a set of algebraic laws, one may be interested in looking into the correspondence with a denotational semantics or with some operational notion of equivalence. An interesting property is whether the equalities derived from the laws are exactly those which hold for a natural notion of process equivalence. If this property holds, the set of algebraic laws is said to be *complete* wrt the notion of process equivalence under consideration.

In the algebraic approach one can simply *postulate* process equalities while in the operational (or denotational) approach one would need to *prove* them. On the advantages of postulation Russell [29] remarked the following:

The method of postulation has many advantages: they are the same as the advantages of theft over honest toil
— Bertrand Russell

Algebraic semantics, however, is a convenient framework for the study of process equivalences; postulating a set of laws, and then investigating the consistency of that set and what process equivalence it produces. Some frameworks (e.g., [18]) combine the operational semantics with the algebraic one by, for example, considering processes modulo the equivalence produced by a set of axioms.

2.3 Specification and Process Logics

One often is interested in verifying whether a given process satisfies a property, i.e., a specification. But process terms themselves specify behavior, so they can also be used to express specifications. Then this verification problem can be reduced to establishing whether the process and the specification process are related under some behavioral equivalence (or pre-order).

Hennessy-Milner's modal logic. Another way of expressing process specifications is by using a process logic. One such logic is the Hennessy-Milner's modal logic. The basic syntax of formulae is given by:

$$F \ := \texttt{true} \mid \texttt{false} \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \langle K \rangle F \mid [K]F$$

where K is a set of actions. Intuitively, the modality $\langle K \rangle F$, called *possibility*, asserts (of a given P) that: It is possible for P to do $\mathbf{a} \in K$ and then evolve into a Q which satisfies F. The modality $[K]P$, called *necessity*, expresses that if P can do $\mathbf{a} \in K$ then it must thereby evolve into a Q which satisfies F.

Formally, the compliance of P with the specification F, written $P \models F$, is recursively given by:

$$\begin{array}{lll}
P \not\models \texttt{false} & & \\
P \models \texttt{true} & & \\
P \models F_1 \wedge F_2 & \text{iff} & P \models F_1 \text{ and } P \models F_2 \\
P \models F_1 \vee F_2 & \text{iff} & P \models F_1 \text{ or } P \models F_2 \\
P \models \langle K \rangle F & \text{iff} & \text{for some } Q,\ P \xrightarrow{\mathbf{a}} Q,\ \mathbf{a} \in K \text{ and } Q \models F \\
P \models [K]F & \text{iff} & \text{if } P \xrightarrow{\mathbf{a}} Q \text{ and } \mathbf{a} \in K \text{ then } Q \models F
\end{array}$$

As an example consider our familiar trace equivalent (but not bisimilar) processes $P_1 = a.(b.0 + c.0)$ and $P_2 = a.b.0 + a.c.0$. Notice that the formula

$$F = \langle \{a\} \rangle \left(\langle \{b\} \rangle \, \texttt{true} \wedge \langle \{c\} \rangle \, \texttt{true} \right)$$

discriminates among them, i.e. $P_1 \models F$ but $P_2 \not\models F$. In fact the discriminating power of this logic wrt a finite processes (i.e., recursion-free processes) coincides with strong bisimilarity (see [31]). That is, two finite processes are strongly

bisimilar iff they satisfy the same formulae in the Hennessy-Milner's logic. The result can be extended to image-finite processes by considering infinite disjunctions and conjunctions in the Hennessy-Milner's logic.

Temporal logics. The above logic can express local properties such as "an action must happen next" but it cannot express long-term properties such as "an action eventually happens". This kind of property, which falls into the category of *liveness properties* (expressing that "something good eventually happens"), and also *safety properties* (expressing that "something bad never happens") have been found to be useful for reasoning about concurrent systems. The modal logics attempting to capture properties of the kind above are often referred to as *temporal logics*.

Temporal logics were introduced into computer science by Pnueli [27] and thereafter proven to be a good basis for specification as well as for (automatic and machine-assisted) reasoning about concurrent systems. Temporal logics can be classified into linear and branching time logics. In the *linear* case at each moment there is only one possible future whilst in the *branching* case at each moment time may split into alternative futures.

Below we consider a very simple example of a linear-time temporal logic based on [20]. The syntax of the formulae is given by

$$F := \texttt{true} \mid \texttt{false} \mid L \mid F_1 \vee F_2 \mid F_1 \wedge F_2 \mid \Diamond F \mid \Box F$$

where L is a set of non-silent actions. The formulae of this logic express properties of sequences of non-silent actions; i.e. traces. For the sake of uniformity, we are interested only in infinite traces. Intuitively, the modality $\Diamond F$, pronounced *eventually* F, asserts of a given trace s that at some point in s, F holds. Similarly, $\Box F$, pronounced *always* F, asserts of a given trace s that in every point of s, F holds.

The models of the formulae are taken to be infinite sequence of actions; elements of Act^ω. Formally, the infinite sequence of actions $s = \mathbf{a}_1.\mathbf{a}_2 \ldots$ satisfies (or is a model of) F, written $s \models F$, iff $\langle s, 1\rangle \models F$, where

$$\begin{array}{lll}
\langle s,i\rangle \models \texttt{true} & & \\
\langle s,i\rangle \not\models \texttt{false} & & \\
\langle s,i\rangle \models L & \text{iff} & \mathbf{a}_i \in L \cup \tau \\
\langle s,i\rangle \models F_1 \vee F_2 & \text{iff} & \langle s,i\rangle \models F_1 \text{ or } \langle s,i\rangle \models F_2 \\
\langle s,i\rangle \models F_1 \wedge F_2 & \text{iff} & \langle s,i\rangle \models F_1 \text{ and } \langle s,i\rangle \models F_2 \\
\langle s,i\rangle \models \Box F & \text{iff} & \text{for all } j \geq i \;\; \langle s,j\rangle \models F \\
\langle s,i\rangle \models \Diamond F & \text{iff} & \text{there is a } j \geq i \;\; \text{s.t. } \langle s,j\rangle \models F
\end{array}$$

Intuitively, P satisfies a linear-temporal specification F, written $P \models F$, iff all of its traces are models of F. Recall, however, that the traces are finite sequences of non-silent actions. But since formulae say nothing about silent actions, we can just interpret a finite trace s as the infinite sequence $\hat{s} = s.(\tau^\omega)$ which results from s followed by infinitely many silent actions. This leads to the definition: $P \models F$ iff whenever $P \stackrel{s}{\Longrightarrow}$ then $\hat{s} \models F$.

Let us consider the definitions $A(a,b,c) \stackrel{\texttt{def}}{=} a.(b.A\langle a,b,c\rangle + c.A\langle a,b,c\rangle)$ and $B(a,b,c) \stackrel{\texttt{def}}{=} a.b.B\langle a,b,c\rangle + a.c.B\langle a,b,c\rangle$. Notice that the trace equivalent processes $A\langle a,b,c\rangle$ and $B\langle a,b,c\rangle$ satisfy the formula $\Box\Diamond(b \vee c)$; i.e. they always eventually do b or c. In general, for every two processes (finite or infinite) if they are trace equivalent then they satisfy exactly the same formulae of this temporal logic. The other direction does not hold in general since the logic is not powerful enough to express, for example, facts about the immediate (or next) future. Take the processes $a.a.0$ and $a.0$; they are not trace equivalent, but they satisfy the same formulae in this simple logic.

2.4 Analyzing Equivalences: Decidability and Congruence Issues

Much work in the theory of process calculi, and concurrency in general, involves the analysis of process equivalences. Let us say that our equivalence under consideration is denoted by $\sim$. Two typical questions that arise are:

1. Is $\sim$ decidable ?
2. Is $\sim$ a congruence ?

The first question refers to the issue as to whether there can be an algorithm that fully determines (or decides) for every P and Q if $P \sim Q$ or $P \nsim Q$. Since most process calculi can model Turing machines most natural equivalences are therefore undecidable. So, the interesting question is rather for what subclasses of processes is the equivalence decidable. For example, bisimilarity is undecidable for full CCS, but decidable for finite state processes (of course) and also for the families of infinite state processes including context-free processes [8], pushdown processes [30] and basic parallel processes [7]. Obviously, the decidability of an equivalence leads to another related issue: the complexity of verifying the equivalence.

The second question refers to the issue as to whether the fact that P and Q are ($\sim$) equivalent implies that they are still ($\sim$) equivalent in any context. The equivalence $\sim$ is a congruence if $P \sim Q$ implies $C[P] \sim C[Q]$ for every context C (as said before, a context C is an expression with a hole [.] such that placing a P in the hole yields a process term). The congruence issue is fundamental for algebraic as well as practical reasons; one may not be content with having $P \sim Q$ equivalent but $R \parallel P \nsim R \parallel Q$.

For example, trace equivalence and strong bisimilarity for the process language here considered is a congruence (see [17]) but weak bisimilarity is not because is not preserved by summation contexts. Notice that we have $b.0 =_B \tau.b.0$, but $a.0 + b.0 \neq_B a.0 + \tau.b.0$. In this case new questions arise: In what restricted sense is the equivalence a congruence? What contexts is the equivalence preserved by? What is the closest congruence to the equivalence? The answer to these questions may lead to a re-formulation of the operators. For instance, the problem with weak bisimilarity can be avoided by using a somewhat less liberal summation called guarded-summation (see [18]).

2.5 Process Calculi Variants

Given a process calculus it makes sense to consider variants of it (e.g., subclasses of processes, new process constructs, etc) to seek for simpler presentations of the calculus or different applications of it. Having these variants one can ask, for example, whether the process equivalences become simpler or harder to analyze (as argued in the previous section) or whether there is loss or gain of *expressive power*.

To compare *expressive power* one has to agree on what it means for a variant to be as expressive as the other. A natural way of doing this is by comparing wrt some process equivalence: If for every process P in one variant there is a Q in the other equivalent to P then way say that the latter variant is as expressive (wrt to the equivalence under consideration) as the former one.

Several studies of variants of CCS and their relative expressive power have been reported in [2]. Also several variants of the π-calculus (itself a generalization of CCS) have been compared wrt weak-bisimilarity (see [32]). An interesting result is that the π calculus construction $!P$ whose behavior is expressed by the law $!P \equiv P \,||\, !P$ can replace recursion without loss of expressive power. This is rather surprising since the syntax of $!P$ and its description are so simple. Other interesting result is that of Palamidessi [22] showing that under some reasonable assumptions the asynchronous version of the π-calculus is strictly less expressive than the synchronous one.

3 Conclusions

The λ-calculus is a canonical model of sequential computation. Unfortunately, there is no canonical model for concurrent computation at the present time. In spite of promising progress in towards canonicity (e.g., [10,21,19]) an all-embracing theory of concurrency has yet to emerge. According to Petri [23] such a general model may attain a range of application comparable to that of physics. As argued in [17], however, even after the discovery of it, we shall need to choose different special models for different applications. Here is an analogy from [14]: Newtonian mechanics is not a suitable framework for describing the flow of fluids, for which one needs a theory containing mathematical concepts corresponding to friction and viscosity. Concurrency, as physics, is a field with a myriad of aspects for which we may require different terms of discussion and analysis.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] R. Amadio and C. Meyssonnier. On decidability of the control reachability problem in the asynchronous $\pi-$calculus. *Nordic Journal of Computing*, 9(2), 2002.

[3] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3:142–188, 1991.

[4] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[5] J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Cong. on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[6] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[7] S. Christensen, H. Huttel, and F. Moller. Bisimulation equivalence is decidable for basic parallel processes. In *CONCUR'93*, LNCS 715, pages 143–157. Springer-Verlag, 1993.

[8] S. Christensen, H. Huttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, 96:203–224, 1992.

[9] P. Cory, E. Hobart, and H. Tracy. Radar-based intruder detection for a robotic security system. In *Proc. of SPIE*, volume 3525, 1999.

[10] V. Gupta and V. Pratt. Gates accept concurrent behavior. In *Proc. 34th Annual IEEE Symp. on Foundations of Comp. Sci.*, pages 62–71. IEEE, 1993.

[11] A. Grzegorczyk. On the definition of computable real continuous functions. *Fund. Math.*, 44:61–71, 1957.

[12] C. A. R. Hoare. *Communications Sequential Processes*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1985.

[13] C.A.R. Hoare. Let's make models. In *Proc. of CONCUR '90*, volume 458 of *LNCS*. Springer-Verlag, 1990.

[14] L. Lamport. Answer to Pratt. Concurrency Mailing List Archive, 19, Nov 1990. Availabe via http://www-i2.informatik.rwth-aachen.de/Research/MCS/Mailing_List_archive/.

[15] R. Milner. Processes; a mathematical model of computing agents. In *Proc. Logic Colloquium 73, eds.*, pages 257–274, 1973.

[16] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[17] R. Milner. *Operational and Algebraic Semantics of Concurrent Processes*, pages 1203–1241. Elsevier, 1990.

[18] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[19] R. Milner. Bigraphical reactive systems. In *Proc. of CONCUR '02*, volume 2154 of *LNCS*. Springer-Verlag, 2002.

[20] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer, 1991.

[21] U. Montanari and F. Rossi. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In *Proc. of COORDINATION '96*, volume 1061 of *LNCS*, pages 12–27. Springer-Verlag, 1996.

[22] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In ACM Press, editor, *POPL'97*, pages 256–265, 1997.

[23] C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP Congress '62*, 1962.

[24] G. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, University of Aarhus, 1981.

[25] G. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

[26] G. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[27] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57. IEEE, IEEE Computer Society Press, 1977.

[28] G.M. Reed and A.W. Roscoe. A timed model for communication sequential processes. *Theoretical Computer Science*, 8:249–261, 1988.

[29] B. Russell. Introduction to Mathematical Philosophy, 1919.

[30] C. Stirling. Decidability of bisimulation equivalence for normed pushdown processes. In *CONCUR'96*, LNCS 1119, pages 217–232. Springer-Verlag, 1996.

[31] C. Stirling. Bisimulation, model checking and other games. Notes for Mathfit Instructural Meeting on Games and Computation, 1998.

[32] D. Sangiorgi and D. Walker. *The $\pi-$calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[33] W. Yi. *A Calculus for Real Time Systems*. PhD thesis, Chalmers University of Technology, 1991.

[34] W. M. Zuberek. Timed petri nets and preliminary performance evaluation. In *Proc. of the 7th Annual Symposium on Computer Architecture*, pages 88–96. ACM and IEEE, 1980.