2

Formalisms and Distributed Calculi

This second chapter provides a broad overview of formalisms and languages used for distribution, parallelism, and concurrency.

We successively study basic formalisms (such as the λ- and π-calculus), concurrent calculus, and finally formalisms with some object concepts.

Among distributed, concurrent, or parallel languages, mainly those benefiting from a formal definition will be analyzed in this chapter. Programmatic and algorithmic issues related to distributed systems are covered at large in [155]. With respect to concurrent programming, see for instance [16, 108]. Finally, for a comprehensive study of parallel programming languages and associated environments, see [148] or [149] and [59].

2.1 Basic Formalisms

This section reviews the main calculi and formalism from which most of the (formally described) concurrent languages and calculi have been derived.

Figure 2.1 provides an informal classification of calculi considering the different concurrency principles.

2.1.1 Functional Programming and Parallel Evaluation

the λ-calculus and pure functional languages provide a simple framework for designing parallel languages. In fact, it is well known that the λ-calculus is confluent [22] whatever evaluation strategy is chosen. In other words, the absence of side effects allows one to evaluate expressions composing the programs in any order. A parallel evaluation of functional languages is both deterministic and deadlock free.

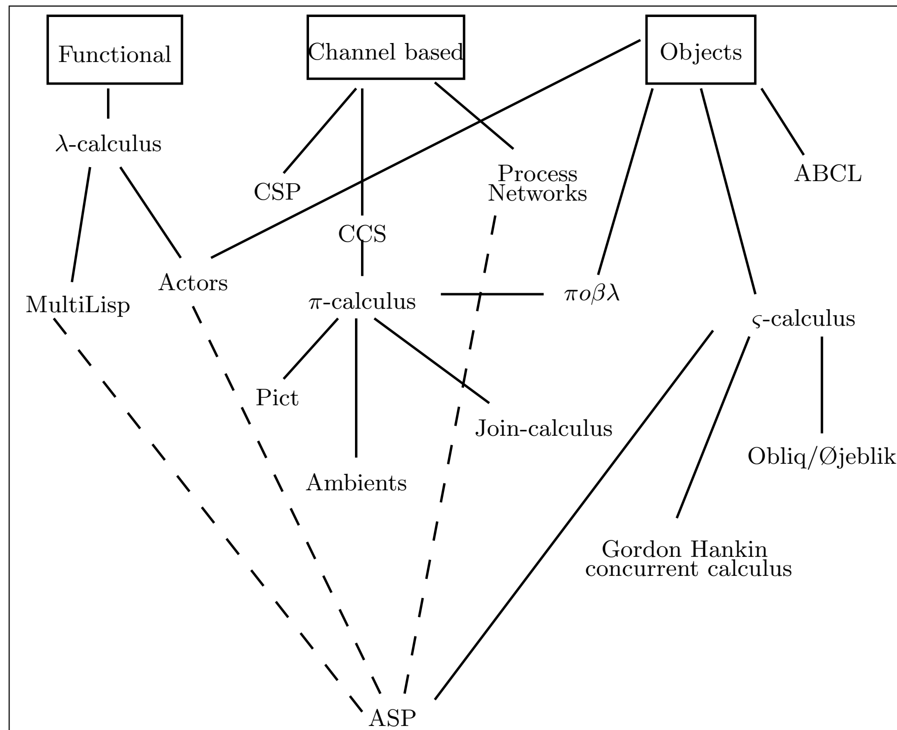

Fig. 2.1. Classification of calculi (informal)

We recall below the syntax of the λ-calculus:

$$\begin{array}{lll} M, N ::= & x & \text{variable} \\ & \mid \lambda x.M & \text{abstraction} \\ & \mid (MN) & \text{application} \end{array}$$

Functional languages are directly inspired and modeled by the λ-calculus. Figure 2.2 gives an example of a binary tree in CAML (a classical functional language). This example is composed of the declaration of a `btree` type, and two functions, one for adding an element and one checking if an element is in the tree. This example is written in a purely functional style and does not contain any side effect. Consequently, the function `insert` has to return the whole tree which could be inefficient in a distributed implementation. As this example is purely functional, any execution of these functions can be performed in parallel without loss of determinacy.

The parallel functional evaluators have been widely studied, see for example [101] for a lazy parallel evaluation, or [84] for a survey of parallel functional programming.

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;

let rec member x btree =
  match btree with
        Empty -> false
      | Node(y, left, right) -> if x = y then true else
          if x < y then member x left else member x right;;

let rec insert x btree =
  match btree with
        Empty -> Node(x, Empty, Empty)
      | Node(y, left, right) ->
          if x <= y then
               Node(y, insert x left, right)
          else
              Node(y, left, insert x right);;
```

Fig. 2.2. A binary tree in CAML

2.1.2 Actors

Actors [9, 10, 11] are a typical example of the *uniform object activity* aspect: each actor is a rather functional process. Actors interact by asynchronous *message passing*. Instead of having an internal state, actors can change their *behavior*, i.e., their reaction to received messages.

Agha, in [9], presented an actor language. More formal syntax, operational semantics, equivalence, and bisimulation techniques for actors are given in [10] and [11]. Actors are based on a functional language but are organized in an object-based style. An actor is an object for which each method is written in a purely functional language. Communication is ensured by a mailbox mechanism and thus is asynchronous. Fairness is an important requirement of actor specification. According to Agha, fairness is a realistic hypothesis that provides semantical properties. Characteristics specific to actors are:

- An actor may *send* a message to another actor; `send(a,v)` creates a message with receiver `a` and contents `v` and sends it.
- An actor may *change* its own behavior. During a message treatment, an actor *must* specify its new behavior (which can be the same as before the message treatment) by the primitive `become(b)`.
- An actor may *create* other actors with the primitives `newaddr()` for creating an address and `initbeh(a,b)` for initializing an actor behavior.

While actors are originally based on a purely functional paradigm, some actor languages (e.g. Thal [103]) use imperative constructs for local behavior in order to provide efficient execution on standard computers. However, since the semantics of imperative sequential programs can be represented in the lambda-calculus using store-passing style translations (see, for example, [135]), this optimization is semantically not significant though of pragmatic importance.

The history-sensitive aspect of actors is performed by the *become* primitive. In other words, the state of an actor is specified by its behavior. This means that the behavior of an actor at invocation time (on message sending) may differ from its behavior at execution time (on message reception). In ASP, such behavior modification will be forbidden and replaced by an imperative aspect allowing objects to modify their state (their data). These two approaches are somewhat opposite. Indeed, Agha and others store the state of the actors inside their functions, while in ASP the state is only stored in the data part of objects (field).

Table 2.1 defines the syntax of an actor language. This syntax only includes the primitives that are characteristic of an actor language.

```
⟨act program⟩ ::= ⟨behavior definition⟩* (⟨command⟩)

⟨behavior definition⟩ ::= (define (id {(with identifier ⟨pattern⟩)}*)
                                      ⟨communication handler⟩*)

⟨communication handler⟩ ::= (Is-communication⟨pattern⟩ do ⟨command⟩)

⟨command⟩ ::= let ⟨let binding⟩* ⟨command⟩
            | (if ⟨expression⟩ then ⟨command⟩ else ⟨command⟩)
            | (send ⟨expression⟩ ⟨expression⟩)
            | (become ⟨expression⟩)
```

Table 2.1. The syntax of an Actors language [9]

```
(define (Factorial ())
  (Is-Communication (a eval (with customer ≡ c)
                            (with number ≡ n)) do
      (become Factorial)
      (if (NOT (= n 0))
         (then (send x 1))
         (else (let (x (new FactCust (with customer c)
                                     (with number n)))
                    (send Factorial (a eval (with customer x)
                                     (with number n-1))))))))
(define (FactCust (with customer ≡ m)
                  (with number ≡ n))
   (Is-Communication (a number k) do
       (send m n*k)))
```

Fig. 2.3. A factorial actor [9]

A notion of futures for actors is also presented in [9]. A λ-calculus-based actor calculus is presented in [10, 11]. Actors are presented as an open system: they can receive messages from outside actors. These articles specify an operational semantics, an operational equivalence, and operational bisimulation techniques for actors.

Figure 2.3 defines an actor computing a factorial that distributes the work to customers. More precisely, a chain of `FactCust` objects is created. Each customer performs one multiplication and forwards the result to the following customer, thus each call to factorial first creates a `FactCust` actor and recursively calls the `Factorial` actor with the newly created customer, for $n - 1$.

In [105], an actor-like concurrent language is presented but is more related to typing theory and can capture "message not understood" errors. It is based on ML-like typing of record calculus. Indeed, this article considers that a concurrent object-oriented programming language is an assemblage where records are added to a concurrent calculus (*à la* π-calculus). Actor semantics can also be built out of the pi-calculus by adding typing restrictions [12, 128].

Aspects	**Possible Values:**
Activity	Actor
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference
Communication Timing	Asynchronous, with a fairness guarantee
Synchronization	Filtering patterns (futures can exist at a higher level)
Object RMI	No
Object Activity	Yes, all objects (uniform)
Wait-by-necessity	No

Table 2.2. Aspects of Actors

A popular actor inspired language in current use is *E* [62], which is particularly useful for scripting P2P systems. It offers *capabilities* for secure distribution. Other systems use a library approach to provide actor functionalities in a conventional language such as Java (see for example, the *Actor Foundry* [8]).

Table 2.2 summarizes the fundamental aspects of Actors. Most of these characteristics are a direct consequence of the uniform model of Actors and of the fact that communications are performed with message passing and pattern matching on message reception. Communications are asynchronous but a fairness hypothesis ensures that every sent message will finally be delivered; however, some implementations of actors ensure a FIFO preserving communication timing.

2.1.3 π-calculus

π-calculus is one of the most popular calculi modeling communications. Compared to actors, π-calculus does not have a built-in notion of object, nor a notion of object identity. Moreover, in its original version π-calculus offers synchronous communications.

π-calculus activities are expressions that communicate synchronously with channels. The expressive power and the concept of mobility in π-calculus come from the possibility to send channels along other channels: the first class entities of π-calculus are the channels.

CCS [117] was introduced by Milner; in this book, we decided to focus on π-calculus which can be considered as an extension of CCS. π-calculus is a concurrent calculus based on communications over channels and introduced by Milner et al. [119, 120, 144]. It is a very small calculus where channels are first-class entities and communication is due to synchronization between a process performing an output on a channel and another process performing a blocking input on the same channel. Channel names are first-class entities and can be passed over channels.

Several versions of π-calculus exist and π-calculus syntax can be presented in different ways. We will use the syntax of Table 2.3 and present the main variants of π-calculus below.

$P, Q ::= \mathbf{0}$	nil
$\mid P \mid Q$	parallel composition
$\mid (\nu x.P)$	restriction of name x
$\mid \tau.P$	unobservable action
$\mid x(y).P$	input
$\mid x\langle y\rangle.Q$	output
$\mid [x = y].Q$	name matching
$\mid P + Q$	choice
$\mid !P$	replication

Table 2.3. The syntax of π-calculus

Informally,

- $x\langle y\rangle.Q$ sends y along channel x and, after synchronization with a process $x(z).P$ listening on channel x, continues as Q. Similarly and synchronously, $x(z).P$ receives y on channel x and continues as P where z is replaced by y.
- $\nu x.P$ creates a fresh channel x with lexical scope P.
- $P|Q$ corresponds to the parallel composition of two processes; $!P$ is an infinite number of P processes running in parallel.
- $P + Q$ denotes an external choice: as soon as P can be reduced, Q is discarded, or vice versa.

- Name matching only exists in some variants of π-calculus. In that case $[x = y].Q$ executes Q if x and y are the same channel.

Table 2.5 defines an operational semantics for (synchronous) π-calculus without matching similar to the one that can be found in [119]. From the structural congruence relation defined in Table 2.4, a reduction relation $\rightarrow$ can be defined ($\{\!\{y \leftarrow z\}\!\}$ denotes the substitution of y by z).

$P \equiv Q$ if P is obtained from Q by change of bound names (alpha conversion)

$$P + \mathbf{0} \equiv P,\ P + Q \equiv Q + P,\ P + (R + R) \equiv (P + Q) + R$$

$$P|\mathbf{0} \equiv P,\ P|Q \equiv Q|P,\ P|(R|R) \equiv (P|Q)|R$$

$$(\nu x.(P|Q)) \equiv P|(\nu x.Q) \text{ if } x \notin fn(P),\ (\nu x.\mathbf{0}) \equiv \mathbf{0},\ (\nu y.(\nu x.P)) \equiv (\nu x.(\nu y.P))$$

$$!P \equiv P|!P$$

Table 2.4. π-calculus structural congruence

In Table 2.5 the TAU rule is associated to unobservable actions. REACT is the (synchronous) communication rule between two processes. PAR means that except for the reaction rule, processes evolve asynchronously. RES allows reduction inside a binder. STRUCT expresses that the terms are reduced modulo structural equivalence; for example, one can reorder processes in order to allow the reaction between two processes. The use of the rules STRUCT and REACT with bound channels allows the restriction scope of these channels to be changed, this mechanism is sometimes called *scope extrusion*.

Most of the π-calculus derived languages and calculi presented below do not include name matching. Name matching makes different equivalence relations on π-calculus [69] equivalent, but this is not directly linked to the subject of this study.

Some variants of π-calculus exclude the choice operator mainly for simplicity.

Nearly all π-calculus theoretical developments are based on the definition of one or several bisimulation relations. The main idea of bisimulation is to define an equivalence between terms based on (potentially infinite) reduction. In very short, two terms are bisimilar if every reduction performed on one term (e.g., receiving z on the channel x) can be performed on the other term, and the two terms after reduction are still bisimilar. Bisimulation definition and techniques are based on a *coinduction* principle. Several variants of bisimulation relations exist which are more or less discriminating (this also depends

TAU:

$$\frac{}{\tau.P + M \to P}$$

REACT:

$$\frac{}{(x(y).P + M)|(x\langle z\rangle.Q + N) \to P\{\!\!\{y \leftarrow z\}\!\!\}|Q}$$

PAR:

$$\frac{P \to P'}{P|Q \to P'|Q}$$

RES:

$$\frac{P \to P'}{(\nu x.P) \to (\nu x.P')}$$

STRUCT:

$$\frac{P \to P' \wedge P \equiv Q \wedge P' \equiv Q'}{Q \to Q'}$$

Table 2.5. π-calculus reaction rules

on the variant of π-calculus on which it is applied). The study of bisimulation is beyond the scope of this book, see for example [144] for a rather complete study of π-calculus variants and some bisimulation relations and techniques.

Variants of π-calculus

Polyadic π-calculus [118] allows one to send/receive several channels on a channel (i.e., $x(y).P$ becomes $x(y_1 \dots y_n).P$). Polyadic π-calculus can be encoded in monadic π-calculus; thus, polyadic π-calculus is mainly useful for introducing the concept of sorts in π-calculus.

π-calculus can be either synchronous or asynchronous. Asynchrony is obtained by disallowing an output prefix. This means that output messages do not have any continuation; that is to say, $x\langle y\rangle.Q$ is replaced by $x\langle y\rangle$ in the syntax above. Asynchronous π-calculus without a choice operator was first proposed by Honda and Tokoro in [90] and Boudol in [35]. Synchronous π-calculus can be encoded in asynchronous π-calculus, but behavioral equivalence differs in synchronous and asynchronous π-calculus. Translation from asynchronous π-calculus to synchronous π-calculus is a convenient model for programming languages where synchronized communications are built upon asynchronous primitives.

Merro and Sangiorgi introduce a (variant of) "local π-calculus" (Lπ) in [115] which forbids the use of a received channel for input or name matching. In [143], Sangiorgi extends this calculus with the capacity of sending processes

through channels (LHOπ: Local Higher Order π-calculus) and shows the compilation from LHOπ to Lπ. A more general compilation of the HOπ (Higher Order π-calculus) into the π-calculus was presented in [141].

PICT is a language based on π-calculus and is briefly described in Sect. 2.2.2.

Linear and Linearized Channels

Linear and linearized channels use typing techniques to provide confluence for some π-calculus terms. A channel with a linear type [104] can only be used once in input and once in output. Thus communication over a linear channel can not be affected by a third process.

Nestmann and Steffen in [124] introduce a generalization of linear channels: "linearized" channels which can be reused after a "unique" usage. This article aims at ensuring, with typing techniques, that at any time only one communication is possible through a given channel.

Communication over linear and linearized channels is deterministic. Thus π-calculus terms only communicating over linear and linearized channels are confluent. This technique seems to be a convenient criterion for deciding statically whether a π-calculus program is deterministic or not.

What is Mobility?

In [119], Milner classifies mobility in three main categories:

(A) *processes* move, in the physical space of *computing sites*;
(B) *processes* move, in the virtual space of *linked processes*;
(C) *links* move, in the virtual space of *linked processes.*

π-calculus adopts the choice (C) and Milner considers that (B) can be reduced to (C). This book partly follows this idea and will not focus on the site where an activity is placed but will rather adopt choice (B) because ASP calculus is based on the notion of activities rather than channels.

Even if some kinds of channels can be implemented in ASP and communicated between processes, this book will be more focused on the concurrency aspects than on mobility of names as presented by Milner. However, a notion of mobility *à la* Milner is intrinsic to ASP through the mobility of global references to some objects: ASP global references can be transmitted between processes.

With respect to the concept of physical place, we consider that the interaction between physical location and the methodology of communication is beyond the scope of this study. Consequently, Chap. 12 will introduce migration in ASP by considering that the content of an activity moves to another activity.

In short, to come back to Milner's classification, this book rather adopts the choice (B) and considers that it can be easily reduced to (A) in practice.

Table 2.6 summarizes the fundamental characteristics of synchronous π-calculus. Of course asynchronous π-calculus features asynchronous communication timing without guarantee. The absence of a buffer for storing messages prevents the existence of other communication timing in π-calculus. Indeed, asynchronous with rendezvous communication would require distinguishing message reception and message treatment which is not compatible with the fact that messages cannot be stored and only an acknowledgment can be returned to the sender. Moreover, asynchronous FIFO preserving communication timing cannot be performed by a simple syntactic modification of π-calculus; for example, asynchronous π-calculus forbids an activity to send several messages. Moreover, the control synchronization can be complemented with simple name matching, if such a primitive is allowed.

Aspects	**Values:**
Activity	Expression evaluation
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference
	Copy of activities (mobility of activities) in HOπ
Communication Timing	Synchronous
Synchronization	Control
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.6. Aspects of π-calculus

2.1.4 Process Networks

Process Networks are mainly characterized by process-based activities communicating with asynchronous FIFO preserving channels (buffers). They feature dataflow synchronization and determinacy but are rather restrictive on the patterns of communications that can be expressed.

The Process Networks of Kahn and others [99, 100, 159] are explicitly based on the notion of *channels* between processes, performing *put* and *get* operations on them. Each process is an independent sequential computing station (no shared memory). They are linked with channels (one to one or one to many) behaving like unbounded FIFO queues making the communications asynchronous.

One Process Network channel can link at most one source process and many destinations. The destinations do not split the channel output, but each

one reads every value put in the channel (a kind of broadcast). The reading on a channel is performed by a blocking *get* primitive. The order of reading on channels is fixed by the source program. Process Networks provide *deterministic* parallel processes, but require that the order of service is predefined and two processes cannot send data on the same channel.

Figure 2.5 shows the example of the sieve of Eratosthenes written in Process Networks. Note that all communications are performed through explicit and blocking `PUT` and `GET` operations which impose a lot of (not always necessary) synchronizations. The behavior of this example is rather simple: an `INTEGER` process generates integers which pass through as many `FILTER` processes as prime numbers found. Integers finally arriving at the `SIFT` process are used to spawn a new `FILTER` process and sent to the `OUTPUT`. Figure 2.4 shows the graph of processes when two prime numbers have been found.

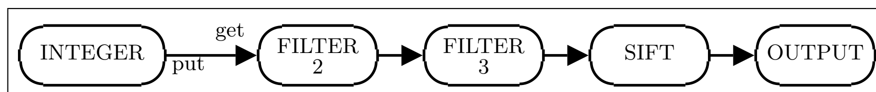

Fig. 2.4. Execution of the sieve of Eratosthenes in Process Networks

Table 2.7 summarizes the fundamental characteristics of Process Networks. The dataflow synchronization aspect is a direct consequence of the blocking *get* primitive.

Aspects	**Values:**
Activity	Process
Sharing	No
Communication Base	Channel
Communication Passing	Copy
Communication Timing	Asynchronous FIFO preserving
Synchronization	Dataflow
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.7. Aspects of Process Networks

2.1.5 ς-calculus

We conclude our basic formalisms with the sequential ς-calculus, at the root of the proposed theory of distributed objects.

```
Process INTEGERS out QO
    Vars N; 1 → N;
    repeat INCREMENT N; PUT(N,QO) forever
Endprocess;

Process FILTER PRIME in QI out QO
    Vars N;
    repeat GET(QI) → N;
           if (N MOD PRIME)≠0 then PUT(N,QO) close
    forever
Endprocess;

Process SIFT in QI out QO
    Vars PRIME; GET(QI) → PRIME;
    PUT(PRIME,QO);  emit a discovered prime
    doco channels Q;
      FILTER(PRIME,QI,Q); SIFT(Q,QO);
    closeco
Endprocess;

Process OUTPUT in QI;
    repeat PRINT(GET(QI)) forever
Endprocess

Start doco channels Q1 Q2;
    INTEGERS(Q1);SIFT(Q1,Q2); OUTPUT(Q2);
    closeco;
```

Fig. 2.5. Sieve of Eratosthenes in Process Networks [100]

Abadi and Cardelli [3, 1, 2] present a calculus for modeling object-oriented languages: ς-calculus. The main contribution of [3] deals with typing in object calculi. Even if this aspect is important, it will not be studied in this book. Indeed in our case, classical typing could be considered as orthogonal with concurrency.

Abadi and Cardelli studied both functional and imperative behavior, starting from an object-based functional calculus (no classes) without typing, then adding imperative aspects and most importantly studied typing. A class-based object calculus can also be translated to ς-calculus; an example of translation is defined in [3]. ς-calculus is a base calculus for several parallel calculi (e.g., Øjeblik [114], the concurrent object calculus of [78], etc.).

ASP calculus is based on an untyped imperative ς-calculus (**imp**ς-calculus). In [3], several equivalent variants of **imp**ς-calculus. The ASP calculus is closer to **imp**ς_f-calculus are discussed. The basic **imp**ς-calculus syntax is described in Table 2.8.

We present below the semantics of **imp**ς-calculus as defined in [3]. It is based on the following syntactic constructs (:: stands for the concate-

$a, b \in L ::= x$	variable
$\mid [m_j = \varsigma(x_i)a_i]^{i \in 1..n}$	object definition
$\mid a.m_i$	method invocation
$\mid a.l_i \Leftarrow \varsigma(x)b$	method update
$\mid clone(a)$	superficial copy
$\mid let\ x = a\ in\ b$	let

Table 2.8. The syntax of **imp**ς-calculus [3]

nation of lists and + the update of an entry in an association map):

ι	store location (e.g., an integer)
$v ::= [m_i = \iota_i]^{i \in 1..n}$	result (m_i distinct)
$\sigma ::= \{\iota_i \mapsto \langle \varsigma(x_i)b_i, S_i \rangle\}^{i \in 1..n}$	store (ι_i distinct)
$S ::= \{x_i \mapsto v_i\}^{i \in 1..n}$	stack (x_i distinct)
$S \vdash \diamond$	well-formed store judgment
$\sigma \bullet S \vdash \diamond$	well-formed stack judgment
$\sigma \bullet S \vdash a \leadsto v \bullet \sigma'$	term reduction judgment

The semantics presented by Abadi and Cardelli is a big-step, closure-based semantics; it is based on three different judgments, and the reduction rules are presented in Table 2.11. Well-formedness judgments are defined in Tables 2.9 and 2.10.

store $\emptyset$:

$$\frac{}{\emptyset \vdash \diamond}$$

store ι:

$$\frac{\sigma \bullet S \vdash \diamond \qquad \iota \notin dom(\sigma)}{\{\iota \mapsto \langle \varsigma(x)b, S \rangle\} :: \sigma \vdash \diamond}$$

Table 2.9. Well-formed store

Gordon et al. presented a substitution-based semantics of **imp**ς-calculus (small step and big step), and proved their equivalence with Abadi and Cardelli closure-based semantics in [79, 80]. We based our semantics on the operational semantics of [79] because it is more intuitive and concise than the one of Abadi and Cardelli. However, such a semantics is based on a substitution that is more difficult to implement efficiently.

stack $\emptyset$:

$$\frac{\sigma \vdash \diamond}{\sigma{\bullet}\emptyset \vdash \diamond}$$

stack x:

$$\frac{\sigma{\bullet}S \vdash \diamond \quad \iota \notin dom(\sigma)}{\sigma{\bullet}\left(\{x \mapsto [m_i = \iota_i]^{i\in 1..n}\} :: S\right) \vdash \diamond}$$

Table 2.10. Well-formed stack

x:

$$\frac{\sigma \cdot (S :: \{x \mapsto v\} :: S') \vdash \diamond}{\sigma{\bullet}\,(S :: \{x \mapsto v\} :: S') \vdash x \rightsquigarrow v{\bullet}\sigma}$$

object:

$$\frac{\sigma{\bullet}S \vdash \diamond \quad \forall i \in i..n,\ \iota_i \notin dom(\sigma)}{\sigma{\bullet}S \vdash [m_j = \varsigma(x_i)a_i]^{i\in 1..n} \rightsquigarrow [m_j = \iota_i]^{i\in 1..n}{\bullet}\{\iota_i \mapsto \langle\varsigma(x_i)a_i, S\rangle^{i\in 1..n}\} :: \sigma}$$

select:

$$\frac{\begin{array}{c}\sigma{\bullet}S \vdash a \rightsquigarrow [m_i = \iota_i]^{i\in 1..n}{\bullet}\sigma' \\ \sigma'(\iota_j) = \langle\varsigma(x_j)a_j, S'\rangle \quad x_j \notin dom(S') \quad j \in 1..n \\ \sigma'{\bullet}(x_j \mapsto [m_j = \iota_i]^{i\in 1..n} :: S') \vdash a_j \rightsquigarrow v{\bullet}\sigma''\end{array}}{\sigma{\bullet}S \vdash a.m_j \rightsquigarrow v{\bullet}\sigma''}$$

update:

$$\frac{\sigma{\bullet}S \vdash a \rightsquigarrow [m_i = \iota_i]^{i\in 1..n}{\bullet}\sigma' \quad j \in 1..n \quad \iota_j \in dom(\sigma')}{\sigma{\bullet}S \vdash a.m_j{\Leftarrow}\varsigma(x)b \rightsquigarrow [m_i = \iota_i]^{i\in 1..n}{\bullet}\,(\{\iota_j \mapsto \langle\varsigma(x)b, S\rangle\} + \sigma')}$$

clone:

$$\frac{\sigma{\bullet}S \vdash a \rightsquigarrow [m_i = \iota_i]^{i\in 1..n}{\bullet}\sigma' \quad \forall i \in 1..n,\ \iota_i \in dom(\sigma') \wedge \iota_i' \notin dom(\sigma')}{\sigma{\bullet}S \vdash clone(a) \rightsquigarrow [m_i = \iota_i']^{i\in 1..n}{\bullet}\,(\{\iota_i' \mapsto \sigma'(\iota_i)\} + \sigma')}$$

let:

$$\frac{\iota\sigma{\bullet}S \vdash a \rightsquigarrow v'{\bullet}\sigma' \quad \sigma'{\bullet}\,(\{x \mapsto v'\}) :: S \vdash b \rightsquigarrow v''{\bullet}\sigma''}{\sigma{\bullet}S \vdash let\ x = a\ in\ b \rightsquigarrow v''{\bullet}\sigma''}$$

Table 2.11. Semantics of **imp**ς-calculus (big-step, closure based)

Note that Gordon et al. [79, 80] also present compilation and CIU (Closed Instance of Use) equivalence on imperative objects. Moreover Gordon and Rees also present a bisimilarity equivalence of the typed object calculi of Abadi and Cardelli in [81]. These aspects deal with the equivalence of *static* terms. In this book, we will not use this framework because, to be as general

as possible, we are interested in relations that are still defined on (partially) reduced terms. Indeed, this book is rather a study of parallel reduction than a study of (statically) equivalent programs.

Figure 2.6 presents an example of a prime number sieve expressed in ς-calculus. It mainly follows the same principle as the Process Network sieve, cloning a new *sieve* object for each prime number found.

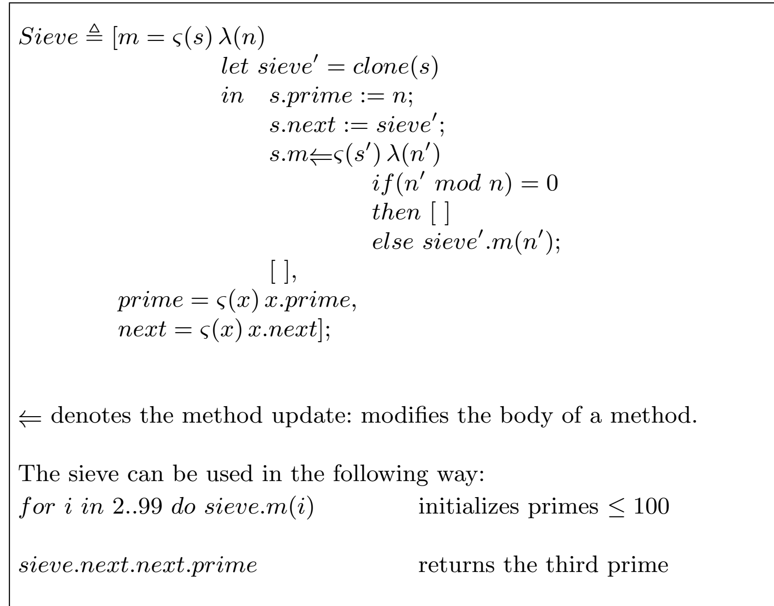

Fig. 2.6. Sieve of Eratosthenes in ς-calculus [3]

Figure 2.7 presents a binary tree class example. It has been slightly modified: types were removed. $\lambda x.b$ and $(a\ b)$ denote abstraction and application of λ-calculus.[1]

Concurrent extension of ς-calculus will be presented in Sect. 2.2.

2.2 Concurrent Calculi and Languages

2.2.1 MultiLisp

To our knowledge, MultiLisp is the first language to define the concept of a *future* and the automatic dataflow synchronization that follows from this notion.

[1] λ-terms can be encoded in ς-calculus.

$$
\begin{array}{l}
binClass \triangleq [new = \\
\qquad \varsigma(z)\,[isleaf = \varsigma(s)\, z.(isleaf\ s), \\
\qquad\quad lft = \varsigma(s)\, z.(lft\ s), \\
\qquad\quad rht = \varsigma(s)\, z.(rht\ s), \\
\qquad\quad consLft = \varsigma(s)\, z.(consLft\ s), \\
\qquad\quad consRht = \varsigma(s)\, z.(consRht\ s)], \\
\quad isLeaf = \lambda(self)\, true, \\
\quad lft = \lambda(self)\, self.lft \\
\quad rht = \lambda(self)\, self.rht \\
\quad consLft = \lambda(self)\, \lambda(newlft) \\
\qquad\qquad ((self.isleaf := false).lft := newlft).rht := self, \\
\quad consRht = \lambda(self)\, \lambda(newrht) \\
\qquad\qquad ((self.isleaf := false).lft := self).rht := newrht]
\end{array}
$$

Fig. 2.7. Binary tree in ς-calculus [3]

Halstead defined MultiLisp [83], a language with *shared memory* and *futures*. The construct $(future\ X)$ immediately returns a future for the value of X and concurrently evaluates X. A future without a value associated to it is said to be *undetermined*; it becomes *determined* when its value has been computed. The combination of shared memory and side effects prevents MultiLisp from being determinate.

The main parallelism primitive is a $PCALL$ that performs an implicit *fork* and *join* and evaluates its arguments concurrently: $(PCALL\ F\ A\ B)$ concurrently evaluates F, A, and B to f, a, and b; and applies f to the arguments a and b.

In MultiLisp, as the addition operator + needs a value of its arguments, the two following expressions yield essentially the same parallelism:

```
(+ (future A) (future B))
```

```
(pcall + A B)
```

Halstead classifies the programming languages by specifying whether they have explicit parallelism, side effects, and shared memory. For example, CCS is characterized by explicit parallelism, side effects and no shared memory, and MultiLisp by explicit parallelism, side effects and shared memory.

According to Halstead [83], the fact that no data is shared between different threads (no shared memory) is one of the failings of CSP. But the interleaving of processes accessing and modifying data can also be considered as disturbing for the programmer as different interleaving between the threads can lead to different results (MultiLisp is not deterministic). Another drawback of CSP pinpointed by Halstead is that it leads to non-uniform access to data. Indeed local accesses can be performed classically, whereas accesses to

data belonging to another process need a communication through channels. In ASP no memory is shared but the copying of shared data is implicit. Consequently, concerning the non-uniform access to data in ASP, the programmer only has to know that objects sent between activities are deeply copied and to deal with the coherence of these deep copies if necessary.

Katz and Weise [102] studied the interactions between futures and continuations and problems arising when those two functionalities are mixed. Flanagan and Felleisen gave a semantics formalization of futures in MultiLisp in [66].

Table 2.12 summarizes the fundamental characteristics of MultiLisp. Most of the items of this classification have a poor signification because of the multithreaded aspect of MultiLisp.

Aspects	**Values:**
Activity	Expression evaluation
Sharing	Yes
Communication Base	No communication
Communication Passing	No communication
Communication Timing	No communication
Synchronization	Future
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.12. Aspects of MultiLisp

2.2.2 PICT

PICT [132] is a language based on π-calculus. It is based on an asynchronous π-calculus without choice and name matching. Thus it features asynchronous communications through channels, and PICT activities are based on expressions.

A core language of PICT is presented in [131] and is used to create more complex objects able to encode classical features. For example, synchronization, locks, choice, and a lot of other primitives can be derived from the core calculus. Typing (sub-typing and type inference) of PICT and higher level features (than π-calculus) are also presented in [132].

From a general point of view, PICT is designed to be used to experiment with new designs of concurrent object structures (like, for example, the one in [131]). Only a few primitives provide the possibility of writing simple objects and their typing. No specific concurrency mechanism for objects has been implemented inside PICT.

Val ::=	Id	Variable
	[*Label* Val ... *Label* Val]	Record
	{ *Type* } Val	Polymorphic package
	(*rec* : `T` Val)	Rectype value
	String	String constant
	Char	Character constant
	Bool	Boolean constant
Label ::=	⟨*empty*⟩	Anonymous label
	Id =	Explicit label
Pat ::=	*Id* : *Type*	Variable pattern
	_ : *Type*	Wildcard pattern
	Id : *Type* @ *Pat*	Layered pattern
	[*Label Pat* ... *Label Pat*]	Record pattern
	{ *Id* < *Type* } *Pat*	Package pattern
	(*rec* : `T` *Pat*)	Rectype pattern
Abs ::=	*Pat* = *Proc*	Process abstraction
Proc ::=	Val ! Val	Output atom
	Val ? *Abs*	Input prefix
	Val ? ∗ *Abs*	Replicated input prefix
	(*Proc* | *Proc*)	Parallel composition
	(*Dec Proc*)	Local declaration
	`if` Val `then` *Proc* `else` *Proc*	Conditional

Table 2.13. A syntax for Pict [132]

Table 2.13 presents the syntax of the Pict language. The main constructors introduce communication (input and output), parallel composition, records, and pattern matching. Note that name matching does not belong in Pict, there is no choice operator, and that replicated processes are necessarily guarded by an input. The presence of records and pattern matching may allow some kind of subtyping and structured programming. *Rectype* is useful to define and type recursive data structures; Package is used to implement polymorphism.

In [130] the presented syntax is slightly different, mainly by the fact that replicated input is removed and somehow replaced by a `def` primitive allowing the introduction a declaration.

Figure 2.8 shows a simple Fibonacci example in Pict using some high-level primitives. This example automatically forks one process for calculating each $fib(i)$ because `def` declaration creates a kind of replicated input process.

Figure 2.9 shows a factorial example written in the core Pict calculus; it is much longer than the example in Fig. 2.8 because it does not use some derived forms also presented in [130] that greatly simplify programming in Pict.

Table 2.14 summarizes the fundamental characteristics of Pict.

```
def fib[n:Int r:!Int] =
  if (|| (== n 0) (== n 1)) then
     r!1
  else
    r!(+ (fib (- n 1)) (fib (- n 2)))

run printi!(fib 7)
```

Fig. 2.8. A simple Fibonacci example in PICT [130]

```
run
(def fact [n:Int r:!Int]=
   (new br:^Bool
      ( {- calculate n=0 -}
        ==![n0 (rchan br)]
      | {- is n=0? -}
        br?b =
          if b then
            {- yes: return 1 as result -}
            r!1
          else
            {- no ... -}
          (new nr:^Int
            ( {- subtract one from n -}
              -![n 1 (rchan nr)]
            | nr?nMinus1 =
              {- make a recursive call to compute fact(n-1) -}
              (new fr?f =
                ( fact!nMinus1 fr]
                | fr?f =
                    {- multiply n by fact(n-1) and send the result
                       on the original result channel r -}
                    *![f n (rchan r)]
                ))))))
  new r:^Int
   ( fact![5 r]
   | r?f = printi!f )
```

Fig. 2.9. A factorial example in the core PICT language [130]

Aspects	**Values:**
Activity	Expression evaluation
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference
Communication Timing	Asynchronous without guarantee
Synchronization	Filtering patterns (pattern matching)
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.14. Aspects of PICT

2.2.3 Ambient Calculus

Ambient calculus [47] is a calculus describing the movement of processes through the explicit notion of *location*: processes execute at an identified place called location. Ambients are convenient for modeling movements through administrative domains (e.g., through firewalls). Consequently ambients naturally feature some kind of mobility as ambients can be moved from one location (an ambient) to another.

An ambient is defined by the following characteristics.

- An ambient is a bounded place.
- Ambients can be nested and can be moved as a whole.
- Computation takes place inside ambients, and can control the ambient itself (e.g., make it move).

The actions of ambients are also called *capabilities*. The capability *in* m allows entry into the ambient m, the capability *out* m allows exit out of m, and the capability *open* m allows the opening of m.

The syntax of ambient calculus is defined in Table 2.15 (n are names, P, Q are processes, and M are capacities).

$P, Q ::=$	$(\nu n)P$	restriction
	$\mathbf{0}$	inactivity
	$P \vert Q$	composition
	$!P$	replication
	$n[P]$	ambient
	$M.P$	action
$M ::=$	$\mathit{in}\ n$	can enter n
	$\mathit{out}\ n$	can exit n
	$\mathit{open}\ n$	can open n

Table 2.15. The syntax of Ambient calculus

Some communication primitives can be added to ambients in a somewhat orthogonal way (in a π-calculus style): $(x).P$ performs an input action that can interact with an asynchronous output action $\langle M\rangle$. Such interactions are only *local* to an ambient. Indeed, long-range communications may need to cross firewalls and should not happen automatically. According to Cardelli and Gordon, long-range communications should be performed by the movement of a "messenger" ambient. With such communication primitives, ambients can encode the asynchronous π-calculus.

Operational semantics is based on the three following rules:

$$n[in\ m.P|Q]|m[R] \rightarrow m[n[P|Q]|r]$$

$$m[n[out\ m.P|Q]|r] \rightarrow n[P|Q]|m[R]$$

$$open\ n.P|n[Q] \rightarrow P|Q$$

and a local communication rule:

$$(x).P|\langle M\rangle \rightarrow P\{x \leftarrow M\}$$

Mobility in π-calculus is a mobility of names: names can be communicated over channels whereas mobility in ambients consists in moving ambients themselves. Thus the notions of mobility in these two calculi are not incompatible; in fact inside each ambient a mobility of names is possible, e.g., by using an encoding of the asynchronous π-calculus.

A strong contribution of [47] concerns the expressiveness of Ambients. This paper also contains many examples of ambients. As such, Fig. 2.10 shows an example of theencoding of locks in ambients [47] and Fig. 2.11 illustrates an encoding of named channels useful to encode the π-calculus in Ambients.

$$acquire\ n.P \triangleq open\ n.P$$
$$release\ n.P \triangleq n[\ ]|P$$

Fig. 2.10. Locks in ambients [47]

$buf\ n$	$\triangleq$	$n[!open\ io]$	a channel buffer
$(ch\ n)$	$\triangleq$	$(\nu n)\,(buf\ n \mid P)$	a new channel
$n(x).P$	$\triangleq$	$(\nu p)\,(io[in\ n.(x).p[out\ n.P]] \mid open\ p)$	channel input
$n\langle M\rangle$	$\triangleq$	$io[in\ n.\langle M\rangle]$	async channel output

Fig. 2.11. Channels in ambients [47]

Note that a meaningless term of the form $n.P$ can arise during reduction, and a type system like the one described in [46] is necessary to avoid such an anomaly.

Table 2.16 summarizes the fundamental characteristics of Ambients. Note that references to ambients are generalized but can only be used locally: for example, before entering an ambient m, another ambient n has first to move in order to be at the same level as m.

Aspects	**Values:**
Activity	Expression evaluation
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference, with local effect
	Copy of ambient (mobility)
Communication Timing	Synchronous
Synchronization	Control
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.16. Aspects of Ambients

2.2.4 Join-calculus

The join-calculus [70, 68, 67] is an asynchronous calculus with mobility and distribution. Synchronization in the join-calculus is based on *filtering patterns* over channels. From the communication point of view, join-calculus can be seen as an asynchronous π-calculus with powerful message receivers (called triggers): a process can be triggered by the presence of several messages simultaneously.

The join-calculus syntax is composed of processes (P), definitions (D), and join patterns (J) as described in Table 2.17.

The join-calculus semantics is based on a reflexive chemical abstract machine (RCHAM) and can be summarized by the rules of Table 2.18. Each rule (of the form $J \triangleright P$) defines a reaction that can occur upon the simultaneous presence of several messages specified by the join pattern J. If the messages are simultaneously pending ($J\sigma$ on the right of $\vdash$) then those messages are consumed and become the term $P\sigma$ defined by the reaction rule. σ is a substitution of names that can be used to unify arguments of pending messages with formal parameters defined in the reaction rule.

Figure 2.12 shows an example of a reference cell in join-calculus. It is based on three channels. *put* and *get* are sent back on κ_0 and will the become

$P ::=$	$\mid x\langle\tilde{y}\rangle$	message emission
	$\mid \mathbf{def}\ D\ \mathbf{in}\ P$	definition of ports
	$\mid P\vert P$	parallel composition
	$\mid \mathbf{0}$	null process
$D ::=$	$J \triangleright P$	rule matching join pattern J (trigger)
	$\mid D \wedge D$	connection of rules
	$\mid \mathbf{T}$	empty definition
$J ::=$	$x\langle\tilde{y}\rangle$	message pattern
	$\mid J\vert J$	joined patterns

Table 2.17. The syntax of the join-calculus

$$\vdash P|P' \leftrightarrow \vdash P, P'$$
$$\vdash \mathbf{0} \leftrightarrow \vdash$$
$$\mathbf{T} \vdash \leftrightarrow \vdash$$
$$\vdash \mathbf{def}\ D\ \mathbf{in}\ P \leftrightarrow D\sigma \vdash P\sigma \qquad \sigma \text{ creates fresh channels}$$
$$J \triangleright P \vdash J\sigma \longrightarrow J \triangleright P \vdash P\sigma$$

Table 2.18. Main rules defining evaluation in the Join calculus

accessible in order to write or read values in the cell. s remains local and is used to store the value contained by the cell.

$$\mathbf{def}\ mkcell\langle v_0, \kappa_0\rangle \triangleq \begin{pmatrix} \mathbf{def} & get\langle\kappa\rangle|s\langle v\rangle \triangleright \kappa\langle v\rangle|s\langle v\rangle \\ \wedge & set\langle u,\kappa\rangle|s\langle v\rangle \triangleright \kappa\langle\rangle|s\langle u\rangle \\ \mathbf{in} & s\langle v_0\rangle|\kappa_0\langle get, set\rangle \end{pmatrix}$$

Fig. 2.12. A cell in the join-calculus [68]

Table 2.19 summarizes the fundamental characteristics of the join-calculus.

2.2.5 Other Expressions of Concurrency

Several other concurrent languages exist, most of them being derived from the basic formalisms. Some of them are object oriented, but in general no unification of objects and activities is clearly identified.

CML

In [136], Reppy presents an extension of SML (Standard ML) called CML (Concurrent ML) for concurrent programming in SML. CML is a threaded

Aspects	**Values:**
Activity	Expression evaluation
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference
Communication Timing	Asynchronous without guarantee
Synchronization	Filtering patterns (join)
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.19. Aspects of the Join-Calculus

language. The synchronization is performed by a `sync` operator which synchronizes on an event. An event is either a *basic event*, i.e., a communication, or an event built by combining several basic event. In the base language, the communications are synchronous but a mailbox (request queue) mechanism can be easily implemented with a buffered channel.

Kell-calculus

Stefani [152] has introduced a calculus that is able to model hierarchical components, especially sub-components control. The *kell-calculus* is based directly on the π-calculus with the possibility to have joins inside triggers (as in *distributed join-calculus* – DJoin [68]).

The **M**-calculus [145] is somehow a preliminary version of the kell-calculus. The kell-calculus is also intended to overcome the limitations of the **M**-calculus presented in [152].

Steele Shared Memory Non-interference

Steele [151] expressed a programming model ensuring the confluence of programs by analyzing (mainly dynamically) the shared memory accesses in order to ensure non-interference. But it is based on a shared memory mechanism with asynchronous threads and not on possibly distributed programs.

Montanari Tile-Based Semantics

Montanari et al. introduced *tile-based semantics* [43, 64] which is based on rewrite rules in side effects. This theoretical framework (based on double categories) has been applied to give a semantics to located CCS in [64]. We think such a framework could be used to provide a modular semantics for ASP.

Functional Nets

Functional nets [127] is a language based on join patterns. It consists of two kinds of functions: synchronous and asynchronous. Asynchronous functions correspond to channels with an asynchronous communication timing as in asynchronous π-calculus. Synchronous functions have a synchronous communication timing and thus a value can be returned by such function (the caller awaits for the result before continuing execution). Only the leftmost operand of a fork or a join operator can return a result. In other words, all operands except the leftmost one must be asynchronous functions. Functional nets also feature an object-oriented view with records.

2.3 Concurrent Object Calculi and Languages

2.3.1 ABCL

ABCL [161, 162] features active objects in an imperative object-oriented language. Placing itself in a uniform model where all objects are viewed as active, it does not have shared objects per se. The communication is RPC, and more specifically based on remote method invocation (object RMI). The language features both synchronous and asynchronous communications, and a *select* construct for filtering and waiting for specific message patterns. Table 2.20 summarizes the fundamental aspects of ABCL.

Figure 2.13 (inspired from [161]) presents a simple bounded buffer in ABCL. The *script* part defines the object behavior: what messages the active object accepts and what actions it performs upon reception. Somehow the script specifies the object activity, with the specificity that the object is initially waiting for a message (it is "dormant"); the buffer initially waits for a `put` or `get` message. The *select* construct allows one to suspend the activity (turning to the "waiting" mode) in order to selectively wait for some message pattern:

```
(select
    (=> [:message-pattern ...]  where constrains ... actions ...)
         .
         .
         .
    (=> [:message-pattern ...]  where constrains ... actions ...)
)
```

The `where` part specifies conditions for the message to be treated. This selective wait is close to the select statements found in languages such as CSP [88] and later on in Ada [91]. In the buffer case, a single message is waited for in each select instruction. On reception of an expected message, the object returns to the "active" mode, treating the message.

```
[object Buffet
   (state declare-the-buffer-state )
   (script

      (=> [:put elt]     ; Put an element in the buffer
        (if full
           then (select   ; Waits for a [:get] message
                   (=>[:get] remove-from-storage-and-return )
                 )
         )
         store-elt
      )

      (=> [:get]         ; Get an element from the buffer
        (if empty
           then (select   ; Waits for a [:put ...] message
                   (=>[:put elt] send-elt-to-get-caller )
                  )
           else remove-from-buffer-send-it-to-get-caller
         )
      )
   )
]
```

Fig. 2.13. Bounded buffer in ABCL [161]

One must notice the *quasi-parallel* nature of ABCL activities: several method activations can exist at the same time, but at most one of them is executing at any given time. This has to be compared to languages with *parallel activities* where several executions (threads) are actually going on simultaneously in the same code. Of course, the other end of the spectrum is *sequential activity* as proposed in ASP: at any time a single method activation, a single thread, a single stack, exist within the object.

The principle of this classification (*sequential, quasi-parallel, parallel activities*) was initially proposed in [160].

Getting back to the code in Fig. 2.13, one can notice the *monitor* [87] nature of this activity: at most one method of the object is executing, while the other activations are suspended on some conditions, respecting a kind of *mutual exclusion*. In the case of a monitor, *condition variables* authorize the expression of wait conditions. For ABCL, the `select` construct permits to wait for specific messages. ABCL seems to get away with the signaling of monitors at the cost of some code duplication. Instead of signaling after a `put` that a potentially blocked `get` can be resumed, the code for putting in an empty buffer is given within the `get` method. Furthermore, such synchronization can be classified as *mainly centralized*: the synchronization code and object activity is somehow gathered in the `script` construct, but still mixed up with

some functional aspects (code for `put` and `get` here). Finally, the mechanism by which a request is treated can be classified as *explicit message acceptance*: there is an instruction that can be placed in the control flow to wait for and execute a request, e.g., the `=>[:get]` in the buffer code. This will be close to the `Serve` primitive of ASP (Fig. 4.2, page 71). Another alternative, used in other languages, is an *implicit message acceptance* where the programmer expresses conditions for accepting a message, but no specific construct authorizes triggering it at a given point in code. An implicit message acceptance authorizes the programming of activity in a declarative style. However, as an explicit message acceptance is rather a primitive construct, and as it also authorizes the construction of declarative abstractions, we believe a concurrent calculus or language should provide an explicit message acceptance primitive.

```
    ; Synchronous communication:
    ; send and wait for message execution
    ; Originally called:  Now Type Message Passing
x := [T  <== M]
        ; Current activity send a message M to T
        ; wait for message execution and return value,
        ; which is stored in x

    ; Asynchronous one-way communication:
    ; send and does not wait for
    ; message execution, no reply
    ; Originally called:   Past Type Message Passing
[T  <= M]
        ; Current activity send a message M to T
        ; no wait

    ; Asynchronous communication with explicit future:
    ; send and does not wait for message execution,
    ; reply to be sent later in  x
    ; Originally called:  Future Type Message Passing
[T  <= M $ x]
      ; Current activity send a meessage M to T
      ; non-blocking, the result will be put
      ; asynchronously in x
...
(ready? x)
      ; Test if x is still awaited
```

Fig. 2.14. The three communication types in ABCL

An important contribution of ABCL was the introduction of three different communication semantics. The language specifies with dedicated syntax three communication timings for an object to send a message to another. From synchronous to asynchronous, ABCL features synchronous communication, asynchronous one-way communication, and asynchronous communication with explicit future. Figure 2.14 presents the corresponding syntax and intuitive semantics. In the last communication type, an explicit future variable can be specified for the reply to be stored. The `ready?` primitive permits one to explicitly test the availability of a value in a future. A future object in ABCL is in fact a queue. It is explicitly declared (`(Future ... x ...)`), and access to it has to be explicit (`[:next-value]`, `[:all-value]`), which of course can be blocking.

An interesting ABCL feature is the capacity to explicitly specify and manipulate the object address where to send a reply. The general syntax for sending a message is in fact the following:

```
[T  <= M  @ dest ]
```

where the variable `dest` specifies where to send the result. For such a communication with a reply, the caller is not interested in the result, but indeed it has to be sent to the `dest` object. Of course, the communication remains non-blocking. Furthermore, the general syntax for accepting a message and returning a reply is:

```
(=> [:M ...] @ dest ... [ dest  <= result ] )
```

An object receiving a message also receives explicitely the destination where it is suppose to send back the result. Turning explicit the reply destination, makes it possible to explicitly *delegate* calls to other objects. For instance, if the message acceptation above is changed into:

```
(=> [:M ...] @dest ... [ T2  <= M  @ dest ] )
```

the call is delegated to the object `T2`, specifying that the reply still has to be sent to the same `dest` object. In that case the reply will be delivered directly, without going through the middle object. Section 10.1 will discuss the possibility to add explicit delegation to ASP and compare it to the implicit delegation existing in ASP because of the first-class futures.

ABCL also features *express mode* message passing: if an object is active and dealing with an *ordinary* message, an express message will suspend the ongoing activity in order for it to be treated right away. By default the suspended treatment will be resumed at the end of the express message treatment. The `atomic` primitive authorizes an object to protect itself against express messages for the execution of a set of instruction. Several versions of the ABCL language were further developed, for instance ABCL/R [158] offers reflective improvements.

The ABCL language is rather significant as one of the first imperative object-oriented languages inspired by the Actors paradigm. Moreover it takes

a different approach to the one proposed here (also in an imperative setting) in considering that all objects are active (uniform model). In ABCL, parallelism is induced by asynchronous communications; even if objects are systematically active, doing standard synchronous communication (now type message passing) does not raise any parallelism in itself. While programming, one has to decide for each call its synchronous or asynchronous nature. The control of parallelism is at the fine granularity of each communication. By contrast, ASP will adopt the control of parallelism at the global level of each object, in a non-uniform active object model.

Table 2.20 summarizes the fundamental characteristics of ABCL.

Aspects	**Values:**
Activity	Active object
Sharing	No
Communication Base	RPC
Communication Passing	Generalized reference
Communication Timing	Synchronous, and
	Asynchronous FIFO preserving
Synchronization	Control
	Filtering patterns (select)
	Future
Object RMI	Yes
Object Activity	Yes, all objects (uniform)
Wait-by-necessity	No

Table 2.20. Aspects of ABCL

2.3.2 Obliq and Øjeblik

Obliq [45] is a language based on the ς-calculus that expresses both parallelism and mobility. Obliq is an object language based on threads communicating with a shared memory (all references to objects are generalized). Thus accesses to objects are strongly concurrent except for serialized objects which can only be accessed by one thread. Figure 2.15 gves an example of a prime number sieve in Obliq.

Øjeblik [123, 41, 114] is a sufficiently expressive subset of Obliq which has a formal semantics. The main results on Øjeblik concern migration but Øjeblik does not take distribution into account.

Øjeblik and Obliq semantics is based on threads ($fork$ and $join$ operators) and all references are global when necessary: When an object reference is passed through the network, a local reference becomes a global reference. As a consequence these languages are based on a shared memory mechanism. Calling a method on a remote object leads to a remote execution of the method

```
let sieve =
{ m =>
   meth(s, n)
     print(n);
     let s0 = clone(s);
     s.m := meth(s1,n1)
              if (n1 % n) is 0 then ok
              else s0.m(n1)
              end
            end;
     end};

 print the primes <100
for i=2 to 100 do sieve.m(i) end;
```

Fig. 2.15. Prime number sieve in Obliq

but this execution is performed by the original thread (or more precisely the original thread is blocked). Thus the parallelism is only based on threads, and is independent of the location of the objects performing operations.

We present in Table 2.21 an untyped syntax of Øjeblik, see [123] for the detailed semantics of Øjeblik.

$a, b \in L ::= s, x, y$	variable
$\mid [m_j = \varsigma(s_i, \tilde{x}_j) a_i]^{i \in 1..n}$	object definition
$\mid a.m_i\langle\tilde{b}\rangle$	method invocation
$\mid a.l_i \Leftarrow \varsigma(s, \tilde{x}) b$	method update
$\mid a.clone$	superficial copy
$\mid a.alias\langle b\rangle$	object aliasing
$\mid a.surrogate$	object surrogation
$\mid a.ping$	object identity
$\mid let\ x = a\ in\ b$	let
$\mid fork\langle a\rangle$	thread creation
$\mid join\langle a\rangle$	thread destruction

Table 2.21. The syntax of Øjeblik

In Øjeblik the notion of argument of a method is introduced: a method has two formal parameters, one is the object itself (*self/this/...*), the other is a function parameter. Such an extra argument is only necessary in the context of remote method calls. Indeed, in a local context, a method call can return a function (e.g., a λ-term) that will be applied to the argument, whereas in the case of remote method invocations, the execution of the method is performed on the distant object. In Obliq and in most of the distributed

calculi the place where a method is executed has a strong influence on its behavior. For example, in a concurrent object calculus like Obliq, it can be useful to protect the state of the object from outside modifications (`protected` keyword explained below). For such protected objects, returning a function and performing operations "inside" the object[2] are not equivalent because only the later solution can modify the state of the invoked object (without losing coherence of the objects or introducing locks). We will see in Part II that in ASP, the argument of methods will also be used as it is deep copied in order to preserve a given topology of links between objects.

In Obliq, the interferences between threads can be limited by serialized objects: if an object is *serialized*, then, at any time, only one thread is inside this object. In other words, a second thread entering an object is blocked until the first one has finished. Serialization may be guaranteed with a mutex. An operation is *self-inflicted* if it addresses the current self. Authorizing reentrant mutexes allows self-inflicted operations to be performed for serialized objects. This allows recursion but not mutual recursion (no call-back).

An Obliq object can be *protected*, as in [122, 123]: "based on self infliction, objects are protected against external modification." That means that for the protected objects, only self-inflicted update cloning and aliasing are allowed. In Øjeblik every object is *protected* and *serialized*.

Migration

In Obliq and Øjeblik migration is the composition of cloning and aliasing:

$$surrogate = \varsigma(s)s.alias\langle s.clone\rangle$$

This composition is deeply studied in Øjeblik, see for example [123].

In [123], Nestmann et al. present different semantics for forwarding and updating. The effect of authorizing (or not) some operations to pass (or not) through the forwarders is studied.

Table 2.22 summarizes the fundamental characteristics of Obliq and Øjeblik. Note that synchronization comes from two aspects: thread destruction and serialization.

The complexity of object interactions that occurs in Obliq and Øjeblik (serialization, self-infliction, protection) is typical of a language without object activity. The orthogonality of activities and objects lead to complex interactions.

2.3.3 The $\pi o\beta\lambda$ Language

Inspired by POOL [14, 15] Jones designed a concurrent object-oriented calculus named $\pi o\beta\lambda$ [95, 96, 94, 97]. $\pi o\beta\lambda$ can be considered as a rather synchronous language where communications are synchronous and asynchrony

[2] Or more precisely, inside a thread belonging to the object invoked.

Aspects	**Values:**
Activity	Process
Sharing	Yes
Communication Base	RPC
Communication Passing	Generalized reference
	Copy of activity (mobility) as cloning+aliasing
Communication Timing	Synchronous
Synchronization	Control
Object RMI	Yes
Object Activity	No
Wait-by-necessity	No

Table 2.22. Aspects of Obliq and Øjeblik

only comes from the possibility to return a result before the end of execution of a method, thus activating two objects at the same time. All references to objects are generalized but objects are protected against external modification; consequently references are shared but not internal states.

The highly synchronous aspect of $\pi o\beta\lambda$ can be summarized by the following facts:

- Only one method of a given object can be active at any time. Using the Obliq vocabulary, one could say that every object is serialized.
- As in Obliq, the calling method is blocked until a result is returned by the called object.

In practice every object is *active* if it is evaluating a method, *waiting* if it is waiting for the result of a method call, or *quiescent* if it has no method currently evaluated. An object only becomes active if it is quiescent and receives a method call or if it is waiting and receives the result of the invoked method.

There is no direct notion of thread in $\pi o\beta\lambda$. Instead, parallelism comes from two facts:

- A function can return a value before the end of its execution. In that case, the calling method obtains the result and can continue its execution while the called function terminates its computation (which will have no consequence on the returned value).
- A function can delegate the task of returning a value to another object by using the *yield* or *commit* or *delegate*[3] primitive. In that case, this object is no longer blocked and the result is directly returned from the last object to the first caller.

These features will have to be compared with automatic and transparent updates of futures in ASP.

Figure 2.16 shows an example of a $\pi o\beta\lambda$ binary tree.

[3] The *yield* primitive of [95, 96] is called *commit* in [110] and *delegate* in [97].

```
class TO
var K:NAT, V:ref(A), L:ref(T), R:ref(T)

method Insert(X:NAT, W:ref(A))
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W);
    return

method Search(X:NAT):ref(A)
    if K=nil then return nil
    else if X=K then return V
         else if X<K then return L!Search(X)
              else return R!Search(X)
```

Fig. 2.16. Binary tree in (a language inspired by) $\pi o\beta\lambda$ [110]

A sufficient condition is given for increasing the concurrency of $\pi o\beta\lambda$ programs without losing confluence, this condition is based on a program transformation. The principle is that an operation can be safely exchanged with a return statement, provided the operation does not *interfere* with the result to be returned. The interference can concern both dataflow aspects – the operation should not affect the result – and control flow ones – the operation should terminate and cannot invoke methods on public objects (because such calls could interfere with calls performed by the caller object which should occur later).

Under this condition, one can return a result from a method before the end of its execution; then the execution of the method continues in parallel with the caller thread. This sufficient condition is expressed by an equivalence between original and transformed program. $\pi o\beta\lambda$ can be translated to (dialects of) the π-calculus (e.g., [94]). From such a translation, Sangiorgi [142] and Liu and Walker [110, 111] proved the correctness of transformations on $\pi o\beta\lambda$ described in [97].

An operational semantics of $\pi o\beta\lambda$ is defined in [97]; this definition seems to be the most adapted to the aspects considered in this book.

Figure 2.17 shows an example with the result of such a transformation applied to the program of Fig. 2.16. Consequently, these two programs behave identically.

There is an equivalent version of the calculus (defining, for example, the sieve of Eratosthenes), with a very different syntax in [95].

Table 2.23 summarizes the fundamental characteristics of $\pi o\beta\lambda$. As for ABCL, all objects are active (uniform), but synchronous communications create awaiting activities, and necessitate a quiescent destination object. Conse-

```
class T0
var K:NAT, V:ref(A), L:ref(T), R:ref(T)

method Insert(X:NAT, W:ref(A))
    return ;
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W)

method Search(X:NAT):ref(A)
    if K=nil then return nil
    else if X=K then return V
         else if X<K then commit L!Search(X)
              else commit R!Search(X)
```

Fig. 2.17. $\pi o\beta\lambda$ parallel binary tree, equivalent to Fig. 2.16 [110]

quently, $\pi o\beta\lambda$ active objects are strongly synchronous. Communication timing is synchronous with early return which is in between "Synchronous" and "Asynchronous FIFO preserving."

Aspects	**Values:**
Activity	Active object
Sharing	No
Communication Base	RPC
Communication Passing	Generalized reference
Communication Timing	Synchronous with early return
Synchronization	Control
Object RMI	Yes
Object Activity	Yes, all objects (uniform)
Wait-by-necessity	No

Table 2.23. Aspects of $\pi o\beta\lambda$

Note that another view of $\pi o\beta\lambda$ could consist in considering activities based on threads which would change the above classification and make it more similar to the one of Øjeblik. Such a classification would be closer to the semantics of $\pi o\beta\lambda$ based on translation into the π-calculus but does not correspond to the original semantics given by Jones.

2.3.4 Gordon and Hankin Concurrent Calculus: concς-calculus

concς-calculus is an archetype of a model where threads are orthogonal to objects. Threads are asynchronous between them, but method calls within

a thread are synchronous. This is highlighted by the fact that threads and objects coexist in every term of **conc**ς-calculus.

Results:		
$u, v \in L ::= x$	variable	
$\mid p$	name	
Denotations:		
$d ::= [m_j = \varsigma(x_i)a_i]^{i \in 1..n}$	object	
Terms:		
$a, b \in L ::= u$		result
	$\mid p \mapsto d$	denomination
	$\mid u.m_i$	method invocation
	$\mid u.l_i \Leftarrow \varsigma(x)b$	method update
	$\mid clone(u)$	superficial copy
	$\mid let\ x = a\ in\ b$	let
	$\mid a \upharpoonright b$	parallel composition
	$\mid (\nu p)a$	restriction

Table 2.24. The syntax of **conc**ς-calculus [78]

Gordon and Hankin [78] proposed a concurrent object calculus: a parallel composition $\upharpoonright$ is added to the ς-calculus. Every object has a name: there is a *denomination* operator. The syntax of **conc**ς-calculus is given in Table 2.24 In such a calculus, a method is executed by the thread that has invoked it. Moreover, objects need to be declared as separate processes that do not perform computation. As a consequence, the notions of object and of executing threads are clearly separate (one could define objects and threads in different spaces).

Moreover a type system is necessary to distinguish terms from expressions as a denominated object can only be a process, but an expression can either be a top-level process (thread) or be included inside another expression. In other words, a denomination ($p \mapsto d$) cannot be used as an expression, it should only appear on the left side of a parallel composition ($a \upharpoonright b$).

Note that an additional synchronization mechanism has to be added to the calculus (via mutexes).

Jeffrey [93] introduced a modified version of Gordon and Hankin's concurrent object calculus, and added the notion of *location* in order to make this calculus distributed.

Table 2.25 summarizes the fundamental characteristics of Gordon and Hankin calculus.

Aspects	Values:
Activity	Process
Sharing	Yes
Communication Base	RPC
Communication Passing	Generalized reference
Communication Timing	Synchronous
Synchronization	Control
Object RMI	Yes
Object Activity	No
Wait-by-necessity	No

Table 2.25. Aspects of **conc**ς-calculus

2.4 Synthesis and Classification

This chapter has reviewed some classical concurrent calculi and languages. We tried to organize a rich set of intertwined languages into a few basic aspects. Table 2.26 summarizes the main aspects of a few calculi and languages.

Among the languages discussed above, we identified three featuring mobility: LHOπ, Ambients, Obliq, and Øjeblik. Note that Silvano Dal Zilio [163] uses a notion of mobile processes different from ours. He classifies mobile processes into those featuring a mobility of names which are called "*labile processes*," and processes with a notion of explicit movement and explicit locations like Ambients calculus, called "*motile processes*." Such a classification explains why the standard π is often credited with capturing mobility, which in fact is only the mobility of names (*lability*). From this point of view, π-calculus is *labile*, while Ambients are *motile*.

The purpose of our study is to examine the impact of different models on the programming paradigm, and especially the methodology to deal with concurrency. From this point of view, Ambients calculus and π-calculus are not very much apart; they both use expression evaluation and channels as fundamental concepts. With respect to locations, we will first fully abstract them away in the core calculus. Then, we will consider true mobility, "*motile processes,*" using the following two-step approach. First, activity identities are considered to be locations; this is consistent with the fact that an identity is directly used to communicate, to find, another activity. Second, a mobility is captured with the deep copy of an activity, and its incarnation with a new identity. This view is rather in accordance with an effective implementation of process mobility. Changing identity/location upon mobility allows us to take into account another practical aspect of mobile processes: localization strategy. Finally, that approach makes it possible to study the impact of mobility on formal semantics, convergence, and determinism (see Chap. 12).

Determinism being an important focus, let us point out the few languages with some convergence or deterministic properties: namely, π-calculus linear

channels, Process Networks. π-calculus is by essence non-determinate, but some programs can be identified as deterministic based on the linear nature of all their channels. On the contrary, Process Networks offer a framework where the programming model enforces determinism; a Process Network program cannot be non-deterministic. It is generally acknowledged that not all concurrent applications must be deterministic; indeed there are some well-known good non-deterministic ones. So the Process Networks approach is probably too dogmatic to be practical. ASP, also featuring determinism, offers a solution somehow in between π-calculus linear channels and Process Networks. There is some part of the ASP programming model that guarantees determinism (e.g., out of order future updates are still determinate), while fully deterministic programs will require extra properties. A specific chapter will identify "*non confluent features*" (Chap. 11, page 143).

In the following parts of this book, the ASP calculus will be presented, both informally and formally. Chapter 21, page 245, will further compare a posteriori most of the languages and calculi presented here with ASP.

		Functional		Channel Based		Object	
Languages Aspects	**ASP**	**Actors**	**MultiLisp**	**π-calculus**	**Process Networks**	**$\pi o\beta\lambda$**	**Obliq Øjeblik**
Activity	Active object	Actor	Exp.	Exp.	Process	Active object	Process
Sharing	No	No	Yes	No	No	No	Yes
Communication Base	RPC	Channel	No com.	Channel	Channel	RPC	RPC
Communication Passing	GR:activities+futures Deep copy of objects Copy of activities (mobility)	GR	No com.	GR	GR only in reconfiguration Copy	GR	GR Copy of Activity (mobility as cloning+aliasing)
Communication Timing	Asynchronous with rendezvous	Asynchronous (fairness)	No com.	Synchronous	Asynchronous FIFO preserving	Synchronous with early return	Synchronous
Synchronization	Blocking service Future	Filtering Patterns	Future	Control	Dataflow Blocking service	Control	Control
Object RMI	Yes	No	No	No	No	Yes	Yes
Object Activity	Yes, non-uniform	Yes, uniform	No	No	No	Yes, uniform	No
Wait-by-necessity	Yes	No	No	No	No	No	No

Exp.= Expression evaluation   No com.= No communication   GR = Generalized Reference

Table 2.26. Summary of a few calculi and languages