ISSN 0105-8517

A Taxonomy for Programming Languages with Multi-Sequential Processes

Kristine Stougård Thomsen
Jørgen Lindskov Knudsen

DAIMI PB-175
May 1984

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

PB-175

Thomsen & Knudsen: A Taxonomy

TRYK: DAIMI/RECAU

A TAXONOMY FOR PROGRAMMING LANGUAGES WITH MULTI-SEQUENTIAL PROCESSES

Kristine Stougård Thomsen, and
Jørgen Lindskov Knudsen

Abstract

The purpose of this paper is to describe a high level conceptual framework - a taxonomy - for programming languages with language constructs for specification of co-sequential and concurrent/interleaved processes.
Using a semi-formal model for processes we identify the major differences and similarities between co-sequential, interleaved and concurrent processes. We discuss the essential aspects of co-sequential processes, concerning different patterns in which control can be transferred between the processes. Moreover, we discuss the important common properties of co-sequential and concurrent/interleaved processes: Synchronization and communication.
The form of this taxonomy is a tree of orthogonal aspects where each aspect may be further subdivided into orthogonal aspects. Each aspect is precisely defined and can be discussed without reference to the other aspects. This makes it easy to use the taxonomy when analyzing and comparing language constructs.
We find that the taxonomy is useful in at least three areas: When choosing a language for a specific application, when designing a language for a specific application area, and when teaching high level languages with multi-sequential processes.

Authors' addresses: Kristine Stougård Thomsen and Jørgen Lindskov Knudsen, Computer Science Department, Aarhus University, Ny Munkegade, DK-8000 Aarhus C, Denmark.

CONTENTS

1 Introduction

The purpose of this paper is to develop a conceptual framework - a taxonomy - for programming languages with language constructs for specification of multi-sequential processes.

The issues discussed are: Composition, synchronization and communication of multi-sequential processes.

Traditionally, programming languages are described either informally by means of natural language, for example in a reference manual for the language, or formally by means of a description of the language syntax and semantics using some formalism, e.g. BNF and denotational semantics.
The disadvantage of an informal description of a programming language is that the language is described using inherently ambiguous natural language and primarily on its own premises, which makes it very difficult to compare it with other languages. However, the informal description has the advantage of being able to point out the important conceptual properties of a language before describing the details of the language.
A formal description has quite the opposite advantages and disadvantages: The formalism provides a common framework by means of which languages can be understood and compared, but the details of a formal description, especially its semantics, will often be so overwhelming that comparison beyond the detailed level of abstraction is nearly impossible. An excellent example of this is the semantic description of Ada given in [4]. The conceptual differences between languages are extremely difficult to isolate from superficial differences between the ways in which conceptually similar language constructs are modelled.

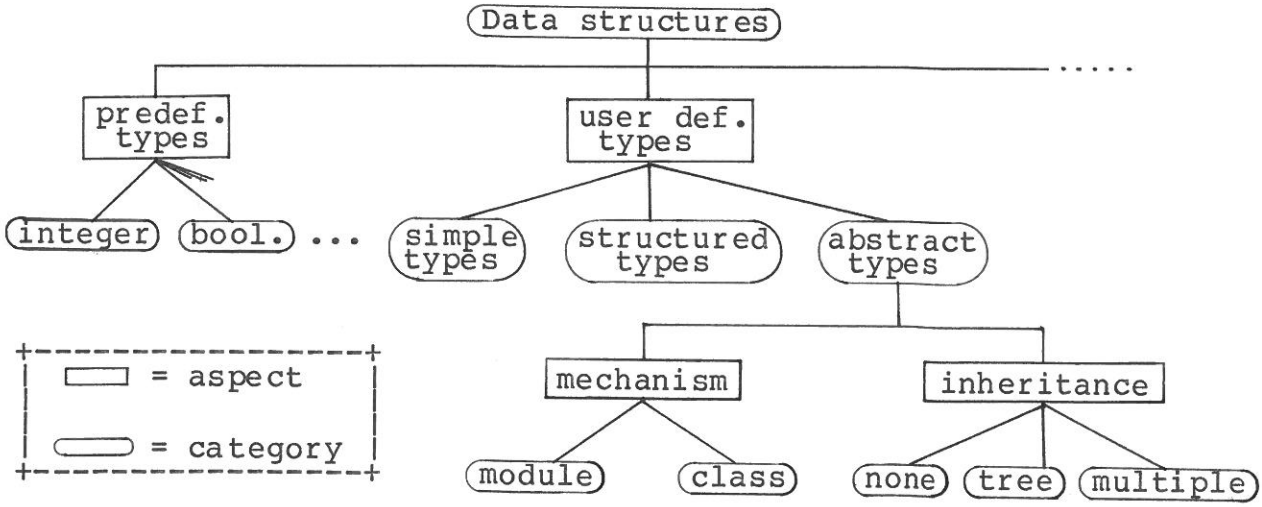
This paper takes a middle course in an attempt to combine the advantages of the informal and the formal way of describing programming languages. The idea is to focus on high level conceptual properties of programming languages and thereby develop a conceptual framework at a higher level of abstraction than traditional semantic models for processes.

Approach

Our approach to the development of the taxonomy is to identify a number of orthogonal aspects of languages for multi-programming. Each aspect involves a number of categories and can be thought of as a question about some specific property of the language, together with a number of possible answers to that question. By orthogonality of the aspects, we mean that the answers of different questions are independent - i.e. all combinations of answers make sense.
Different answers to the same question are not necessarily disjoint. That is, the same language may support several categories of an aspect, normally by means of different language constructs.

A specific answer to a question sometimes makes it relevant to pose further questions. That is, for a category there may be a number of orthogonal aspects related to this specific category. We therefore get a hierarchy of aspects as our taxonomy.

We will illustrate our taxonomy graphically in the following way: The taxonomy (or a part of the taxonomy) can be illustrated by a graph with two different kinds of nodes: aspects and categories. From an aspect, the graph can branch into a number of different categories, whereas from a category, the graph can branch into a number of new orthogonal aspects relevant for this category. Such a graph is also called an AND/OR graph [15]. Example of a taxonomy for data structures:



Goal

The goal of this paper is to develop a high level conceptual framework, but on the other hand we want the framework to be detailed enough to distinguish between languages that we find differ in important ways. How far to go is an act of balance, and surely new aspects can easily be defined under the most detailed categories of our taxonomy. However, we feel that the chosen level of detail suffices for many purposes.

Application

The framework can be of value at least in the following three situations:

1) When choosing a language for a specific application. The potential languages are analyzed and easily compared using the framework, which will highlight the conceptual differences between the languages. Moreover, the focus on the central aspects of the languages, identified by the framework, makes it easier to decide which properties of the language are important to the considered application.
2) When designing a language for a specific application area. The framework can be useful when trying to get an overview of the features wanted in the language, and when evaluating which goals are achieved by different potential language constructs.
3) When teaching high level languages for multi-programming. The framework can be used to convey a very useful overview of similarities and differences between languages.
   Our own experiences with use of the taxonomy at a graduate course on programming languages are encouraging. The taxonomy directs the attention directly to the central aspects of languages for multi-programming, and it is our impression that the students use the concepts from the taxonomy when recalling properties of the languages discussed at the course.

How to use the taxonomy

When using the taxonomy, it is important to understand that the taxonomy is intended as a tool for analysis of programming language constructs, not for specific programs or program executions. That is, we want to discuss the direct semantic properties of language constructs independently of a concrete use of the construct or a special discipline in the use of the construct. Often one language construct can be simulated by means of a disciplined use of another construct. For instance, semaphores can be used to simulate critical regions, whereas the "when" statement in Edison [7] supports critical regions directly.
We only regard a language construct to support a given concept (e.g. abstract datatypes) if the semantics of the construct gives the properties of the concept (e.g. encapsulation of data and operations) independently of the discipline in the use of the language construct.

Organization of the paper

In section 2 we introduce a model for processes and define the terms: sequential, nested, co-sequential and concurrent composition of processes. The term multi-sequential composition will be used to cover the last two. In section 3 we discuss the essential features of co-sequential process sequencing. The prime issue here is a discussion of various ways in which control may be shifted from one co-sequential process to another.
Section 4 deals with synchronization and communication between multi-sequential processes.
Throughout the paper, we illustrate our concepts by giving examples of analysis of language constructs from a number of different programming languages - e.g. Simula67 [9], Concurrent Pascal [5], Distrubuted Processes [6] and Ada [14]. In section 5 we give a full analysis of three programming languages: CSP [12], PLITS [10] and SR [1] to illustrate the strength of this taxonomy. We assume that the reader is familiar with the languages mentioned.
In section 6 we discuss the relation between our taxonomy and

related works by others.

2 A model for processes

One of the problems involved in establishing a taxonomy is to define the fundamental concepts upon which the taxonomy will be based. One of the drawbacks of other taxonomies is that they tend to define their fundamental concepts rather imprecisely. We have chosen to define our fundamental concepts in terms of a model for processes based on the notion of events.

An event is an atomic and instantaneous change of state. The events are partially ordered in time. We will talk about the ordering as a "comes before" relation and write: el<e2, if el comes before e2 in time. If two events are unordered, they may occur simultaneously.

We consider processes as being composed of events in the following way: A process is a coherent set of partially ordered events. That is, a process is a set of events that we have chosen to consider as a conceptual unit.

A process can at a higher level of abstraction be considered as composed of a number of other processes. Compositions of processes can be formed in different ways resulting in different orderings of the events in the processes.

We distinguish between four ways of composing processes: sequential composition, nested composition, co-sequential composition and concurrent composition.

2.1 Sequential composition

In a sequential composition of a number of processes, the processes are executed one after the other in strict order.
In terms of the partial order of events, this can be expressed as follows: Two processes are sequentially composed if all events

from one process - say P1 - comes before all events from the other process - say P2.
Formally, P1 is sequentially composed with P2 if

$$\forall e1 \text{ in } P1: (\forall e2 \text{ in } P2: e1 < e2)$$

2.2 Nested composition

In a nested composition of two processes, the one is executed as part of the other, like a procedure call.
In terms of the partial order of events, P2 is nested in P1 if P2 is executed as a whole between two events of P1.
Formally, P2 is nested in P1 if

$$\forall e1 \text{ in } P1: ((\forall e2 \text{ in } P2: e1 < e2) \text{ or } (\forall e2 \text{ in } P2: e1 > e2))$$

2.3 Co-sequential composition

In a co-sequential composition of a number of processes, execution alternates between the processes, so that their partially ordered sets of events are merged, but only one process is executing at a time.
In terms of the partial order of events, any two events e1 and e2 from two different processes in a co-sequential composition are ordered either (e1<e2) or (e2<e1), but as opposed to sequential composition, not all such pairs need to be ordered in the same way.
Formally, P1 is co-sequentially composed with P2 if

$$\forall e1 \text{ in } P1: (\forall e2 \text{ in } P2: ((e1 < e2) \text{ or } (e1 > e2)))$$

Shift of execution from one process to another is made at the so-called changeover points.

2.4 Concurrent composition

In a concurrent composition of a number of processes, the processes are executed overlapping in time.

In terms of the partial order of events, every pair of events from two different processes are unordered.

2.5 Discussion

We have defined four composition forms but how are they interrelated? If we consider sequential composition, it is easy to see that it is a special case of nested composition and furthermore it is easy to see that nested composition is a special case of co-sequential composition.
However, the nature of both sequential and nested composition is that the processes involved conceptually share one single thread of control.
We say that sequentially and nested composed processes are uni-sequential (shares one thread of control) whereas co-sequential and concurrent processes are multi-sequential (the processes have individual threads of control).

We do not discuss uni-sequential processes further in this paper but focus our attention on multi-sequential processes.
We want to use the concepts co-sequential and concurrent composition of processes to characterize programming languages, and not individual processes or individual implementations of a language. That is, we want to be able to determine whether a language supports co-sequential or concurrent composition of processes by considering only the semantics of the language and not the implementation. For instance, we will say that Concurrent Pascal [5] supports concurrent composition independently of whether the language is implemented by means of interleaving on a single processor or by means of true parallelism using several processors. An interleaved implementation could be thought of as a co-sequential execution, but the changeover points are not part of the semantics of the language (and thereby not determinable by the programmer). We will therefore say that, conceptually, Concurrent Pascal only supports concurrent composition.
In general, we decide to consider a language as supporting co-

sequential composition of processes only if the changeover points are explicitly specified in the process descriptor*.

We will first discuss the special aspects of co-sequential processes due to the explicit changeover points, and afterwards we will discuss the common aspects of all multi-sequential processes due to the presence of multiple control flows and the need for the processes to interact.

3 Special characterization of co-sequential processes

We will characterize co-sequentially composed processes by two orthogonal aspects of the specification of changeover points: Freedom of changeover points and choice of successor.

3.1 Freedom of changeover points

When we consider the freedom of changeover points we focus on the intra-process control aspects of the point; that is, we want to capture the control behaviour relative to the process itself. The possibilities are that the control always leaves the process in the changeover point or that the control may leave the proces in the changeover point. Intuitively, in the first case the process states that it is unable to progress until another process has been in control whereas in the latter case the process just allows the other processes to take over control.
More formally speaking, a language construct may support specification of changeover points as either obligatory or optional.

A. A changeover point is said to be specified as obligatory if it will always cause a shift of execution when an instance of the process descriptor reaches the changeover point.

B. A changeover point is said to be specified as optional if it

---

*) A descriptor is a template from which instances are created. For example, a procedure is a descriptor for procedure invocations.

may cause a shift of execution when an instance of the process descriptor reaches the changeover point.

3.2 Choice of successor

When we consider the choice of successor in a changeover point we focus on the inter-process control aspects of the point. That is, we focus on how the successor is determined. We find that there are two important cases. In the first case, the choice is totally under the control of the process reaching the changeover point. That is, the process knows the exact identity of the successor process. In the second case, the choice is only partially under the control of the process reaching the changeover point. That is, the process does not know the exact identity of the successor process.
This can be precisely stated by distinguishing between language constructs that support deterministic choice of successor, and language constructs that support non-deterministic choice of successor.

A. The choice of successor is said to be specified as deterministic if the successor is unambiguously determined from the state of the process when it reaches the changeover point.

   Variability of successor

   The process may reach the same changeover point in the process descriptor several times in different states. If the choice of successor is deterministic, it is of interest whether the same process is chosen each time, or not.
   We say that a language construct may support constant or variable successor specification.

   a. The successor at a changeover point is said to be specified as constant if the same successor is chosen each time the process reaches the changeover point.
   b. The successor at a changeover point is said to be

specified as variable if different successors may be selected each time the process reaches the changeover point.

B. The choice of successor is said to be specified as non-deterministic if the choice is not determinable from the state of the process when it reaches the changeover point.
It is important whether the language provides any means for the programmer to restrict the set of processes among which the successor is chosen or whether the semantics of the language restricts the set of possible successors in some way.
We will not try to give an exhaustive discussion of possible ways in which languages provide mechanisms for control of the non-determinism, but as an example we can mention that the programmer may be able to specify that control may be transferred to any process within some group of processes, or to some process satisfying some condition (e.g. maximum priority). An example of a language restrictions is that control may only be transferred to one of the processes in the same scope as the currently executing process, or again, to the process with the highest priority.
Note, that if the choice of successor is non-deterministic then it is obviously also variable in the sense defined above.

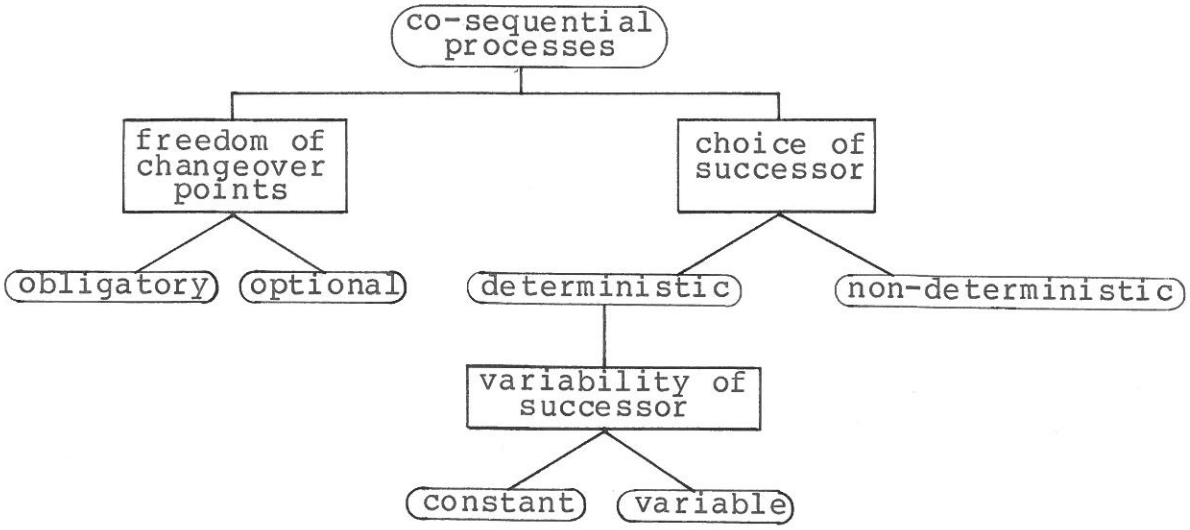
Examples: The bodies of class instances in Simula67 [9] are co-sequentially composed, and the changeover points are described by means of "detach" and "resume" statements. The changeover points are obligatory. Resume is parameterized by a reference to the receiver of the control, whereas detach always returns control to the surrounding program unit. That is, the choice of successor at a resume is deterministic and variable, whereas the choice of successor at a detach is deterministic and constant.

In Distributed Processes (DP) [6], the initial statement of a process and the external requests to the process are co-sequen-

tially composed. The changeover points are located at the "when" and "cycle" statements. If the condition in a when- or cycle-statement is true, there will be no transfer of control, so the changeover points are optional. The choice of successor is non-deterministic.

- oOo -

We can illustrate our characterization of co-sequential processes graphically as follows (cf. section 1):



4 Common characterization of multi-sequential processes

The purpose of composing processes multi-sequentially is either that they must co-operate to do some task, or that they must share some resources or a mixture hereof.
In a multi-sequential composition, the processes must be mutually controlled in time. Besides, they often have to transfer information to each other.
We will characterize multi-sequential processes by characterizing their mechanisms for synchronization and communication.

4.1 Synchronization

When two processes are multi-sequentially composed, their execution has to be mutually controlled.

When processes share some resources (e.g. processor, I/O-devices, data structures, etc.), each process often requires exclusive access to the resources in order to avoid collision with the other processes. If such exclusive access is ensured, the processes are said to be synchronized by mutual exclusion with respect to the resource.

When a number of processes co-operate to do some task, they may sometimes need to agree about their state of execution. We say that two multi-sequential processes P1 and P2 synchronize by mutual admission at two points p1 in P1 and p2 in P2, if P1 is not allowed to continue execution after p1 until P2 has reached p2, and vice versa.

Examples: In Concurrent Pascal, mutual exclusion of processes is enforced with respect to monitors.
Mutual admission is involved in synchronous communication as will be discussed later.

4.2 Communication

Several principles of communication between multi-sequential processes have been proposed. In the following, we will present a scheme for classification of communication principles.

The following discussion will be eased if we introduce the concepts: Communication point, initiator and recipient.

- A communication point in a process is a point where the next possible event may involve communication between the process and some other process/processes.
- The initiator of a communication is the process that supplies the stimulus (see below) that causes the communication to take place.
- A recipient of a communication is a process that receives the stimulus.

To classify languages with respect to principles of communication, we identify five major orthogonal aspects of a communication: The medium and the timing of a communication, the choice of partner, the choice of message or operation, and the visibility of communication points.

4.2.1 Medium of a communication

The stimulus of a communication is a message that is sent from the initiator to the recipient. The recipient decodes and reacts upon the stimulus.
By decoding we mean identifying the semantic contents of the stimulus. The semantics of the programming language defines the semantic domain of the stimulus (e.g. whether it is a bit-stream, an integer, a compound message, an abstraction, etc.). The decoding is often specified using the type-system of the programming language.
By the reaction upon the stimulus we mean the events of the recipient that happens as a direct consequence of the reception of the stimulus (e.g. computing values based on the received

information).

A language construct may support communication by means of either messages or operations. The two different media of communication differ in that they cause different reactions upon the stimulus.

A. In a message communication, the involved multi-sequential processes transfer information by means of a message that is sent between them or by means of shared data structures in which messages are placed.
The initiator of a message communication is the process that delivers the message, whereas the recipient receives or reads the message. That is, we consider the message as the stimulus of a message communication.
An important property of a message communication is that the reaction upon the stimulus is the responsibility of the recipient. That is, the reaction upon the stimulus is not defined by the semantics of the language construct but must be programmed explicitly for the recipient.
An interesting aspect of a language construct for message communication is what kinds of messages it allows. For instance, simple values, references to variables, types, procedures, classes, etc., and composite messages.

B. In an operation communication, the involved multi-sequential processes request operations to be executed by each other. Information may be exchanged by means of the parameter mechanisms for operations.
The initiator of an operation communication is the process that requests the operation to be executed. That is, we consider the request as the stimulus of an operation communication. The recipient is the process that accepts the request.
An operation communication can be considered as a message communication in which both the decoding and the reaction upon the stimulus is defined by the semantics of the language. The decoding of the stimulus is defined to be an

operation name and possibly some parameters. It is clearly of interest which parameter modes the language allows (e.g. in, out, etc.) and which kinds of parameters are allowed (e.g. values, references to variables, types, etc.).
The reaction upon the stimulus of an operation communication is execution of the operation named in the stimulus. We call the execution of the operation the response of an operation communication.

Operation composition

The execution of an operation associated with an operation communication can be considered as constituting a process. The communication can be further characterized by the way in which this process is composed with the initiator and the recipient, respectively. In both cases, the following possibilities exist.

a. nested
   Note that sequential composition is a special case of nested composition.
b. co-sequential
c. concurrent

Examples: Concurrent Pascal uses message communication. The values involved are accessed through monitors.
Both Ada [14] and DP use operation communication. In Ada, the operation is executed as a procedure call from both the recipient and the initiator; that is, the operation is nested with both the recipient and the initiator.
In DP, the operation is executed alternating with the recipient (and perhaps a number of other operations); that is, in co-sequential composition with the recipient. This co-sequential composition was further characterized in section 3. The composition with the initiator is nested.

4.2.2 Timing of a communication

When we consider the timing of a communication we focus on the relative timing of the initiator and the recipient as they reach their correponding communication points. It is of interest whether a communicating process knows anything about the state of its partner at the communication point.
A language construct may support either synchronous or asynchronous communication.

A. A communication is said to be synchronous if the communicating processes must be synchronized by mutual admission at their corresponding communication points, and the stimulus is delivered and received before any of them continue execution. That is, both the initiator and the recipient may be delayed at their communication points.
It is a property of synchronous communication that information received by a communicating process may mirror the state of the communication partner at the time of the reception.
Note, that a synchronous operation communication does not require that the response is completed before the communicating processes can continue.

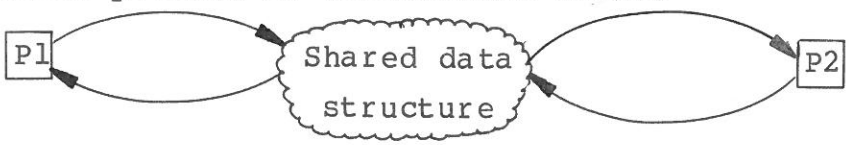
B. A communication is said to be asynchronous if the communicating processes need not be synchronized by mutual admission at their corresponding communication points.
If a process in an asynchronous communication receives information it cannot conclude anything about the state of its partner at the time of the reception.

To implement an asynchronous communication some shared data structures are needed to store the stimulus of the communication.
We characterize an asynchronous communication further by considering two orthogonal aspects of these shared data structures: The visibility of the shared data structures and the influence on process progression.

B.1 Visibility of shared data structures

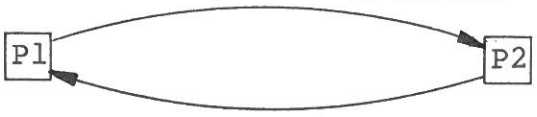
Though shared datastructures are necessary to implement asynchronous communication, they are not necessarily conceptually part of the programming language. By the visibility aspect we want to distinguish between languages in which the shared data structures are explicitly referenced, and languages in which the shared data structures are part of the implementation of the language only. This aspect is interesting because it implies different conceptual communication patterns between processes.

a. The shared data structures are said to be visible if they are directly referenced by the communicating processes. That is, by naming of the data structures. Often this implies that the partner need not be specified at all.
Visible shared data structures give a conceptual communication pattern as illustrated below.



That is, the shared data structure is a visible component in the communication system.

b. The shared data structures are said to be hidden if they are never directly referenced by the communicating processes. Usually, programming languages that support hidden shared data structures require some kind of partner specification instead (see section 4.2.3 for a further discussion of partner specification).
Hidden shared data structures give a conceptual communication pattern as illustrated below.



That is, the processes are the only components in the communication system, connected by communication lines.

B.2 Influence on process progression

The second aspect of the shared data structures is whether they can delay the progression of the initiator and the recipient, respectively, at a communication point.
Especially in a real-time environment it is of great importance to the overall performance of the program that the programmer is aware of any potential delay caused by the shared data structures.

a. Shared data structures are said to be blocking if the state of the data structures may cause any of the communicating processes to be delayed at its communication point. There can be several reasons for such a delay. For example, the delay may be due to protection of the shared data structures by mutual exclusion, or it may be due to a full or empty buffer. We have considered a subdivision of asynchronous, blocking communication with respect to the reason for the blocking, but we have not been able to give an exhaustive classification of possible reasons. Still, the reason for blocking is important so we will talk about "blocking caused by 'some reason'".

b. Shared data structures are said to be non-blocking if they never cause any of the communicating processes to be delayed.

Examples: In Ada, the rendezvous is a synchronous operation communication. Both in and out parameters are available.
In Concurrent Pascal, asynchronous communication is used with monitors as visible shared data structures. The communication is blocking caused by mutual exclusion both for the initiator and the recipient.
In an article by Roper and Barter [16], a programming language derived from CSP is described. Here we see an example of asynchronous message communication with hidden shared data struc-

tures. Furthermore, the communication is blocking for the initiator caused by limited buffer capacity (the buffer capacity is implementation defined) and blocking for the recipient since messages must be sent by the initiator before they can be received by the recipient.

4.2.3 Choice of partner

When we consider the choice of partner by a process we focus on how flexibly a language construct allows a communication to be specified. We want to distinguish whether the process specifies one specific or several possible partners. In the first case, it is totally under the control of the process which partner is chosen, whereas in the second case we can say intuitively that the process is relatively indifferent to the choice of partner.
The choice of partner must be considered both for the initiator and the recipient.
Note, that the following discussion is similar to the discussion of the choice of successor in co-sequential processes (cf. section 3.2).
We distinguish between language constructs that support deterministic and non-deterministic choice of partner in a communication.

A. The choice of partner is said to be specified as deterministic if the partner is unambiguously determined from the state of the process when it reaches the communication point.

   Variability of the partner

   The process may reach the same communication point several times in different states. If the choice of partner is deterministic, it is of interest whether or not the same partner is chosen each time the process reaches the communication point.

   The most tightly connected communicating processes are ob-

tained in the case where the same partner is chosen each time. That is, the process is at that communication point constantly connected to one specific process. A more flexible situation occurs when the process has the possibility of specifying different partners each time it reaches the communication point.

a. The partner in a communication is said to be specified as constant if the same partner is chosen each time the process reaches the communication point.
b. The partner in a communication is said to be specified as variable if different partners may be selected each time the process reaches the communication point.

B. The choice of partner is said to be specified as non-deterministic if the process is unable to determine its partner from its own state when it reaches the communication point
It is important whether the language provides any means for the programmer to restrict the set of possible partners or whether the semantics of the language restricts the set of possible partners in some way. In the first case, the programmer may have the opportunity to specify that the stimulus of the communication must satisfy some condition, or that the partner must satisfy some condition (e.g. highest priority, belonging to some specific group of processes - say printers). Examples of restrictions improved by the language semantics are that the partner is chosen among processes in the scope, or the processes with the highest priority.
Note, that if the choice of partner is non-deterministic it is obviously also variable in the sense defined above.

Examples: Let us consider communication in Ada. The communication points in a initiator looks like this:

<TaskName>.<EntryName>[(<Parameters>]

The partner is uniquely named, which implies that the choice of partner by the initiator is deterministic and constant. However,

if the task is an instance of a task access type or a component of an array of tasks, then it is possible for the initiator to have a variable choice of partner.
The communication points in a recipient is specified like this:

accept <entryname>[(<parameters>)] do <block>

The partner of the recipient is non-deterministically chosen among the processes in the scope.

4.2.4 Choice of message or operation

By choice of message or operation we consider the flexibility avaiable when specifying the message or the operation involved in the communication. The discussion of the choice of message or operation is equivalent to the above discussion of choice of partner. We will therefore carry the discussion through only for the choice of partner as the discussion of the choice of message or operation can be obtained by substituting "partner" with either "message" or "operation" depending on whether the communication under consideration is message or operation oriented.

Examples : In Ada, the choice of operation in the initiator is non-deterministic since the entry name may be associated with different blocks depending on the state of the recipient. That is, the choice is not determinable from the state of the initiator.
In the recipient, the choice of operation is deterministic if the accept statement is standing alone, and constant since the block is statically associated with the accept statement. If the accept statement is part of a select statement, the choice of operation in the recipient is non-deterministic among the operations involved in the select statement.

4.2.5 Visibility of communication points

A final important aspect of the specification of communication points is whether the language construct causes the communication points to be specified implicitly or explicitly in the process descriptor.

A. Implicitly specified communication points are communication points that are invisible in the process descriptor. That is, the implicitly specified communication points are controlled by the implementation, not the programmer.
Implicitly specified communication points may at first sight seem to be a useless construct and in fact make no sense for the initiator. On the other hand, implicitly specified communication points in the recipient are very relevant, especially in a real-time environment. If the recipient specifies communication points implicitly, it renounces control of when the communication is to occur and states that the communication is so urgent that it may take place at any time. This corresponds to the interrupt mechanism in assembly languages but we will later discuss high-level constructs supporting implicitly specified communication points in the recipient.

B. Explicitly specified communication points are communication points that are visible in the process descriptor. That is, it is the programmer that controls the location of the communication points in the process descriptor, and this implies that the process controls when to communicate.

Freedom of communication points

A language construct that supports explicitly specified communication points may be further characterized by whether the communication points are specified as obligatory or optional in the process descriptor.
The concept of optionally specified communication points are especially important in a real-time environment in that they make it possible for the programmer to specify communication

points in a more flexible manner and thereby hopefully achive more efficient programs.

a. A communication point is said to be specified as obligatory if a process must communicate when it reaches the point.

b. A communication point is said to be specified as optional if a process may communicate when it reaches the point.
Several possibilities of optionality of communication points exists. Let us mention two common examples. First, the optionality may be such that the choice of whether to communicate or not is an arbitrary choice made by the implementation. (This is the case in CSP [12]). Secondly, the optionality may be such that a communication will only take place if it does not delay the process. (This is the case in Ada).

Note, that this discussion is similar to the discussion of freedom of changeover points (cf. section 3.1).

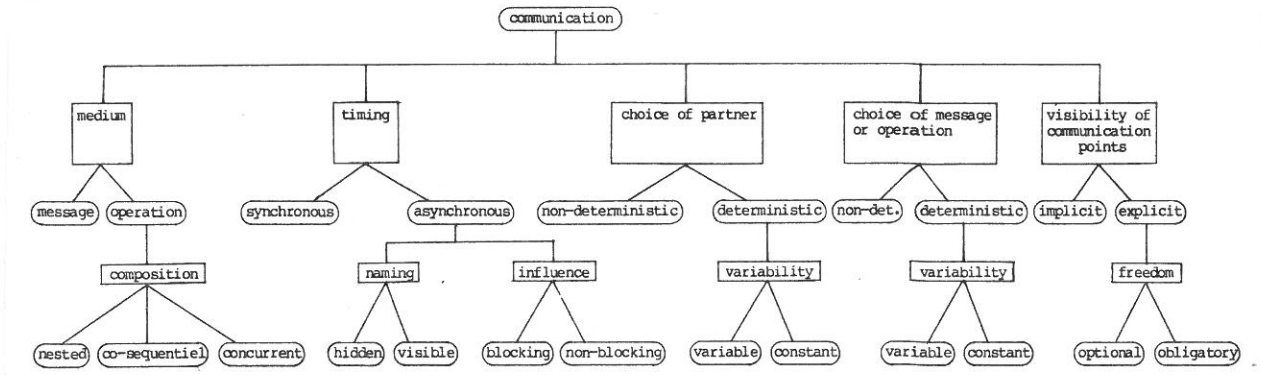
Examples: In Ada, the communication points are explicit and obligatory in the initiator, unless the entry call is a conditional or a timed entry call in which case it is optional.
The communication points are explicit in the recipient and obligatory unless the accept statement is part of a select statement with an else, delay or terminate alternative in which case it is optional.

As an illustration of implicitly specified communication points we can look at StarMod [8]. One of the ways in which processes can communicate in StarMod is by means of activation of local processes in each other. In our terminology this kind of communication is an operation communication where the operation (StarMod process) is concurrently composed with both the initiator and the recipient. However, in the case of result-returning operations the composition with the recipient is

nested. The communication points are implicit in the recipient meaning that operations may be called any time independent of the state of the recipient.

- oOo -

We can illustrate our classification of communication principles graphically as follows (cf. section 1).



5 Extended examples

During the description of the taxonomy we have already given a number of examples of analysis of language constructs. Partial analysis has been given for language constructs in Simula67 [9], Distributed Processes [6], Concurrent Pascal [5], Ada [14], Roper's variant of CSP [16] and StarMod [8].
We will not in this section complete all these examples, since it would be mostly to repeat ourselves. Instead we choose three new languages for a detailed analysis. We will analyze the languages CSP [12], PLITS [10] and SR [1]. The reason why we have chosen these languages is that a comparative analysis of CSP and PLITS very convincingly illustrates the ability of the taxonomy to highlight the conceptual similarities and differences between languages. SR is chosen because it illustrates how compactly and precisely a complicated language can be described by means of the taxonomy.
We assume that the reader is familiar with the example languages.

5.1 Analysis of CSP

CSP supports concurrent composition of processes but not co-sequential composition.

Synchronization: Mutual admission is supported as part of the communication mechanism.

Communication:

Medium: Message communication. Messages are lists of simple values.

Timing: Synchronous.

Choice of partner: The initiator specifies its partner by means of a unique name or by indexing into an uniquely named array of processes. That is, the specification is deterministic and constant (or variable if process arrays are used).

The recipient can do the same, but can also use a guarded command which gives a non-deterministic choice of partner. All the possible partners are explicitly named by the recipient, which gives the process control over the set of processes among which the partner is chosen.

Choice of message: The initiator of a communication specifies the message as an expression that may contain variables. That is, the initiator specifies the message as deterministic and either variable or constant.
The recipient specifies the type of the message, so the choice of message is non-deterministic from the recipient with the type specification as the means of controlling the non-determinism. Further non-determinism can be obtained by use of the guarded command implying that any among a specified number of messages with different types may be selected.

Visibility of communication points: The communication points are explicit in both initiator and recipient. Communication points are obligatory in the initiator, but can be optional in the recipient by use of a guarded command with guards that contain no input commands. The criterion for whether to communicate or not is not defined by the semantics of the language.

5.2 Analysis of PLITS

PLITS supports concurrent composition of processes only.

Synchronization: Mutual admission as part of the communication mechanism.

Communication

Medium: Message communication. Messages are sets of pairs consisting of a name and a value. The value may be a reference to a process.

Timing: Asynchronous. The shared data structures are invisible

to both processes. They, are non-blocking to the initiator but blocking to the recipient when no messages have arrived.

Choice of partner: The initiator specifies the partner by means of an expression that may be either a constant or a variable identification of a process. That is, the initiator specifies its partner deterministically, and the language supports both variable and constant choice of partner.
The recipient can specify its partner either deterministically in the same way as the initiator or non-deterministically by omitting the process identification in the communication statement. Moreover, a key identifying the subject of the communication can be specified to control the set of processes among which the partner can be chosen.

Choice of message: The initiator specifies the message deterministically and variably (or constantly) as in CSP.
The recipient specifies the message non-deterministically. All name fields of the message must be known to the recipient, and the corresponding values must match the specified types of these names. If a subject key is specified, the non-deterministic choice is limited to messages with this subject key.

Visibility of communication points: Explicit and obligatory in both initiator and recipient.

- oOo -

This analysis makes it easy to sum up the conceptual differences between CSP and PLITS:

1) Communication in CSP is synchronous whereas it is asynchronous in PLITS.
2) The structure of messages differ a little in CSP and PLITS, most importantly by the property that PLITS allows references to processes to be transferred in a message.
3) The partner specification in the initiator is more flexible in PLITS than in CSP. Both allow the choice of partner to be

variable, but in CSP only by a fixed array of processes. In PLITS, a process can communicate with a process whose identity has been received through a message.

4) The recipient's control of the non-deterministic choice of partner differ in the two languages. PLITS allows a choice among all processes that send messages about a specific subject. Subject identification is completely under the control of the processes, so this gives a very flexible control of the non-determinism.
   CSP is more restrictive in that the recipient must list the unique names of all possible partners.

5) In CSP a communication point can be optional in the recipient, whereas it is always obligatory in PLITS. However, the semantics of CSP makes it completely arbitrary whether or not a communication is made at an optional communication point. If optional communication points should be of any use in a real time application, the choice of whether to communicate or not should depend on whether or not any communications are ready - perhaps within a timeout period, as in Ada.

5.3 Analysis of SR (Synchronizing Resources)

Only concurrent composition of processes is supported.
Processes in SR are grouped into resources, which are abstractions of physical processors. Different resources correspond to distributed physical processors.

Synchronization: Mutual admission is supported as part of one of the three communication mechanisms in SR.
Mutual exclusion is supported between processes in the same resource with respect to common variables in the resource.

Communication:

Medium: Both operation communication and message communication is supported in SR.

In an operation communication of SR, the operation is always nested composed with the recipient. The composition with the initiator may be either nested or concurrent depending on whether the operation is "called" or "sent", respectively. Both "in" and "out" parameters are allowed, but in case of a "sent" operation, the "out" parameters are ignored.
Message communication in SR is only allowed between processes in the same resource, and messages are structured as arbitrary data structures in SR.

Timing: The "call" mechanism for operations support synchronous communication, whereas the "send" mechanism for operations support asynchronous communication. Furthermore, message communication in SR is asynchronous.
In the "send" operation communication, the shared data structures are hidden in the implementation of the asynchronous communication, and whether they are blocking or not depends on the implementation.
The message communication involves visible shared data structures that are blocking because of protection. The reason why SR only allows this kind of communication between processes in the same resource, is the lack of common storage between processes in different resources in a distributed system.

Choice of partner: In an operation communication (both "send" and "call"), the initiator specifies its partner deterministically, either constantly by means of a resource name and an operation name, which uniquely determines a partner, or variably by means of a capability, which is an assignable reference to a specific operation in a specific process.
The recipient specifies the partner non-deterministically in an operation communication. The non-determinism is controlled by means of a scheduling expression. Only the processes with the minimal value of the scheduling expression are considered.
In a message communication both processes choose their partner non-deterministically. The non-determinism is controlled by the convention that only processes in the same resource can be

partners.

Choice of message or operation: In an operation communication the operation is chosen deterministically by the initiator. The choice is either constant or variable depending on whether or not capabilities are used.
The recipient in an operation communication chooses the operation non-deterministically by means of a kind of guarded command. That is, the operations among which the choice is to be made are listed explicitly and the non-determinism can be further controlled by a boolean synchronization expression. Only operations for which the synchronization expression is true are considered.
In the message communication, the initiator describes the message by means of an expression. That is, deterministically and either constantly or variably depending on the kind of expression. The recipient expects a specific type of the variable that contains the message, and thereby specifies the message non-deterministically with type control.

Visibility of communication points: The communication points are explicit in both initiator and recipient in all the three different kinds of communications in SR.
A message communication point is obligatory for both processes, and an operation communication point is obligatory to the initiator. Whether an operation communication point is also obligatory to the recipient depends on whether or not the guarded command in the recipient contains an else part. If there is no else part, the point is obligatory. If there is an else part, this may be selected instead of a communication if no communications are ready. In this case, the point is optional.

6 Relation to other works

Other taxonomies exist but we find that they all serve a slightly different purpose than ours. Lauer and Needham [13] present a classification of communication principles into two groups: "Message-oriented" and "Procedure-oriented". This approach is followed by Andrew and Schneider [2] who augment the classification with one additional group: "Operation-oriented". We find such classifications too simplified. They make it possible to classify programming languages into two or three major groups but is of no help when two programming languages from the same group are to be compared. For example, Ada and Distributed Processes are both classified as operation oriented languages according to Andrews and Scneiders concepts, and no means are provided to distinguish Ada and DP though there are many important differences between the two languages.

Our approach is more in the line of Bacon [3] and Filman & Friedman [11] with respect to the level of detail of the taxonomy. However, our classification differ substantially from theirs with respect to both contents and form.
If we first focus our attention on Bacons paper, we observe that concerning contents, the most important difference is the emphasis in our taxonomy on the specification of a communication by initiator and recipient. We discuss in detail different strategies that give the processes different degrees of influence on the time, sequence, partners and contents of communications. These aspects are only superficially and imprecisely described by Bacon.
The form of our taxonomy also differs considerably from that of Bacons. Our consistent use of precisely defined orthogonal aspects makes it clear how concepts may be combined, independently of existing programming languages.
Bacons classification relates very closely to existing languages for distributed systems. The concepts are not precisely defined and it is unclear how they can be combined. Therefore, Bacons

classification is less suited for analysis of other languages than those considered when the classification was developed, and the classification is more likely to need changes to adapt to each new language considered. Moreover, it is less suited to use in the design of new languages.

If we now look at the paper by Filman and Friedman, we see first of all that their form is very similar to ours. They, too, develop a set of orthogonal aspects, but there are two major differences. First, their classification is mainly a one-level classification. That is, it is not possible to further classify a given aspect. Secondly, their definitions tend to be imprecise - e.g. the concept of equal communicators is not defined.
If we look at the contents of their classification we see that several of their aspects are similar to aspects in our taxonomy, but we find that they lack treatment of some important areas: Medium of a communication, choice of partner, choice of message or operation, and visibility of communication points. On the other hand, Filman and Friedman treat aspects like process creation, information flow, fairness and failure, which we have ignored. Lastly, they lack any treatment of co-sequential composition.

7 Conclusion

We have described a taxonomy for programming languages with support of multi-sequential processes. The taxonomy focuses on high level conceptual aspects like composition of processes, synchronization and communication. Besides, special aspects of co-sequential processes are discussed concerning the transfer of control between the processes.
The taxonomy is structured as a hierarchy of orthogonal aspects defined by means of an event based model for processes.
Examples of analysis of programming languages and language constructs have been shown. The examples show how the taxonomy highlights the conceptual similarities and differences between

programming languages, and it is seen how precisely and compactly complicated languages can be analyzed by means of the taxonomy. It is our view that the taxonomy can be useful when choosing a language for a specific application, when designing a language for some application area and when teaching programming languages with multi-sequential processes.

Acknowledgment

The work reported here has benefited from many discussions with Ole Lehrmann Madsen and Peter Mosses. Thanks are due to Brian H. Mayoh for several comments and to the students who used an earlier version of this article.

8 References

[1] : G.R.Andrews
"The Distributed Programming Language SR - Mechanisms, Design and Implementation"
Software Practice and Experience, Vol.12, 719-753 (1982)

[2] : G.R.Andrew & F.B.Schneider
"Concepts and Notations for Concurrent Programming"
Computing Surveys, Vol.15, 3-43 (1983)

[3] : J.Bacon
"An Approach to Distributed Software Systems"
Operating Systems Reviews, Vol.15, No.4 (Oct.81)

[4] : D.Bjørner & O.N.Oest (Eds.)
"Towards a Formal Description of Ada"
Lecture Notes in Computer Science, Vol.98, 1980

[5] : P.Brinch Hansen
"The Architecture of Concurrent Programs"
Eaglewood Cliff: Prentice-Hall, 1977

[6] : P.Brinch Hansen
"Distributed Processes: A Concurrent Programming Concept"
Communications of the ACM, Vol.21, 934-941 (1978)

[7] : P.Brinch Hansen
"Edison - a Multiprocessor Language"
Software Practice and Experience, Vol.11, 325-361 (1981)

[8] : R.P.Cook
"*Mod - A Language for Distributed Programming"
IEEE Transactions on Software Engineering, Vol.6, 563-571 (1980)

[9] : O.J.Dahl, B Myhrhaug & K.Nygaard
"Simula67 Common Base Language"
Publication no. S-22, Norwegian Computing Center, Oslo 1970

[10]: J.A.Feldman
"High Level Programming for Distributed Computing"
Communications of the ACM, Vol.22, 353-368, (1979)

[11]: R.E.Filman & D.P.Friedman
"Models, Languages and Heurestics for Distributed Computing"
National Computer Conference,
AFIPS Conference Proceedings, vol.51, (1982)

[12]: C.A.R.Hoare
"Communicating Sequential Processes"
Communications of the ACM, Vol.21, 666-677 (1978)

[13]: H.C.Lauer & R.M.Needham
"On the Duality of Operating System Structures"
Operating System Reviews, Vol.13, no.2 (Apr.79)

[14]: Military Standard
"Ada Programming Language"
ANSI/MIL-STD-1815A-1983
American National Standards Institute, 1983

[15]: N.J.Nilsson
"Problem-Solving Methods in Artificial Intelligence"
McGraw-Hill, Inc, 1971

[16]: T.J.Roper & C.J.Barter
"A Communicating Sequential Process Language and Implementation",
Software Practice and Experience, Vol.11, 1215-1234 (1981)