

ISSN 0105-8517

A Conceptual Framework for Programming Languages

Jørgen Lindskov Knudsen
Kristine Stougård Thomsen


DAIMI PB - 192
April 1985

PB - 192

Knudsen & Thomsen: Conceptual Framework for Prog. Lang.

TRYK: DAIMI/RECAU

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

The logo of Aarhus University, featuring a stylized, geometric representation of a building or architectural structure composed of various rectangular blocks and lines.

ISSN 0105-8517

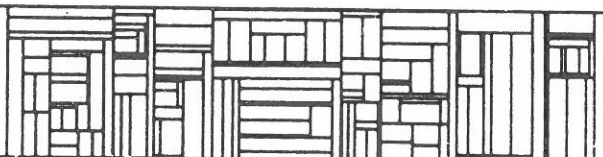
A Conceptual Framework for Programming Languages

Jørgen Lindskov Knudsen
Kristine Stougård Thomsen

DAIMI PB - 192
April 1985

Computer Science Department
AARHUS UNIVERSITY

Ny Munkegade - DK 8000 Aarhus C - DENMARK
Telephone: 06 - 12 83 55



CONTENTS

INTRODUCTION	1
1 CHARACTERIZATION OF THE PROGRAMMING PROCESS	5
1.1 Structures Related to the Programming Process	7
1.2 Functions Related to the Programming Process	10
2 CONCEPT UNDERSTANDING AND ABSTRACTION	12
2.1 Two Different Views of Concepts.	14
2.1.1 The Aristotelian View of Concepts	14
2.1.2 The Fuzzy View of Concepts	16
2.1.3 The Modelling Function, Reconsidered	17
2.2 Abstraction	19
2.2.1 Classification and Exemplification	20
2.2.2 Generalization and Specialization	21
2.2.3 Aggregation and Decomposition	23
2.2.4 Analogies to Mathematical Set Operations	24
2.2.5 Hierarchical Concept Structures	25
2.2.6 The Abstraction Function, Reconsidered	27
3 FUNDAMENTALS OF PROGRAMMING LANGUAGES	28
3.1 Descriptors	30
3.2 Entities	31
3.2.1 Data- and Process-entities	32
3.2.2 Interaction of Data and Process	34
3.3 Values	35
3.3.1 Value Domains	35
3.3.2 Applicative Versus Imperative Programming	37
3.4 Processes	39
3.5 Names	40

3.6	Parameterization	43
3.6.1	Parameterization by Values	46
3.6.2	Parameterization by Entities	46
3.6.3	Parameterization by Descriptors	47
4	ABSTRACTION IN PROGRAMMING LANGUAGES	52
4.1	Classification and Exemplification	53
4.1.1	Classification of Data	54
4.1.2	Classification of Process	55
4.2	Aggregation and Decomposition	56
4.2.1	Different Kinds of Aggregation Components	58
4.2.2	Constraints	61
4.2.3	The Control Aspect of Process Aggregation	62
4.3	Specialization and Generalization	66
4.3.1	Advantages of Specialization	68
4.3.2	Simulation of Generalization by Aggregation	74
4.3.3	Use of Aggregation in Specialization	76
4.3.4	Specialization of Process-descriptors	79
4.3.5	Single Versus Multiple Inheritance	82
4.3.6	Summary Concerning Specialization	83
4.3.7	Analysis of Some Language Constructs	83
5	CONCLUSION	90
6	REFERENCES	92

A CONCEPTUAL FRAMEWORK FOR PROGRAMMING LANGUAGES

by

Jørgen Lindskov Knudsen
Kristine Stougaard Thomsen

Abstract

The conceptual framework presented in this paper contains language independent concepts that enable discussions of programming languages in general, independently of the premises of specific languages. Within the framework, special emphasis is put on analysis of the abstraction techniques: Classification, aggregation and generalization.

The discussion of the various aspects of programming languages is based on a model for program executions - the descriptor-entity model.

The conceptual framework can serve as an aid in gaining a coherent understanding of the programming language area, and in the process of language description and standardization.

The approach to the development of the framework is to attack the programming language area from two different perspectives. Firstly, programming languages are seen as tools for conceptual modelling, and secondly, as tools for instructing the computer.

INTRODUCTION

Aim and Approach

The main goal of this paper is to develop a conceptual framework for the study and design of programming languages. It is our hope that the framework will prove valuable as an aid in gaining a coherent understanding of the programming language area and as a source of inspiration when evaluating and designing programming languages.

Our conceptual framework is concerned with language independent concepts. Instead of developing the framework based on a detailed examination of various existing languages features, we focus on conceptual aspects that are interesting regardless of the kind of programming language in question. This means that the framework enables language independent discussions instead of discussions based on the premises of specific languages, or families of languages.

Our approach to the development of the conceptual framework is to focus on the programming language as a tool for a human programmer in a programming process. We discuss in detail the characteristics of the programming process and observe that programming is an activity in which modelling and abstraction are central parts. When we create a program, we create a model of a structure of concepts related to our problem area.

We consider the programming language as playing a dual role in the programming process: Firstly, as a tool for modelling and abstraction, and secondly, as a way of instructing the computer in what to do.

Introduction

This analysis of the role of the programming language in the programming process inspires us to attack programming languages from two different perspectives: A conceptual modelling perspective, and a computer instruction perspective. Both perspectives contribute to our conceptual framework by showing important aspects of programming languages and by demonstrating basic requirements that must be met by programming languages.

The conceptual modelling perspective leads us into an examination of how to discuss and describe phenomena, concepts, and their interrelations. We discuss in depth some different abstraction techniques that are of primary importance to concept understanding and concept structuring. This approach reveals two kinds of requirements to programming languages: Requirements concerned with concept description, and requirements concerned with support of abstraction.

We discuss how concepts can be described in programming languages and discuss the problems due to the fact that formalized language is to be used.

The majority of existing programming languages take only limited advantage of the power of the different abstraction techniques, although most languages do support some kind of abstraction - e.g. in the form of abstract datatypes or procedures. Therefore, we discuss in detail the advantages of supporting the different abstraction techniques and we show how they can be realized in programming languages. In particular, we give a thorough analysis of the abstraction technique called generalization/specialization which has many important advantages but is supported only by very few languages. We consider our analysis of this abstraction technique as one of the major contributions of our work.

The computer instruction perspective leads us to develop a model of program executions that at the same time expresses the fundamental properties of the computer and acts as a basis for our further discussion of programming languages. Instead of a traditional model that takes its starting point in storage and

manipulations of storage, we choose a descriptor-entity model that is better suited as a basis for the discussion of the relation between the concepts from the problem area to be modeled, the programming language, and the program executions on the computer. With this model as a starting point, we discuss some fundamental aspects of programming languages. In particular, we relate our discussion of abstraction in programming languages to our model of program executions.

Applications

As already mentioned, the language framework is intended as an aid in gaining a coherent understanding of the programming language area and as a source of inspiration when designing new programming languages. One of the most important parts of the design process is the language report and/or the language standard. The standards are usually written in a kind of formalized english that differs from language standard to language standard. That is, the language standard both seeks to define the language and the terminology in which to define the language. This manifold of language terminologies makes it very difficult to learn a new language since one is forced to learn both a new terminology and a new programming language.

Using the conceptual framework presented in this paper in language standards will be a step towards more uniform and comprehensible standards. The languages can be described within the same framework implying that similarities and differences between the languages can be more easily extracted.

We do not claim that a language standard can be based on this framework alone. In order to be precise, a formal method for definition of the details of the semantics must supplement the conceptual presentation of the language.

Organization of the Paper

Chapter 1 contains an analysis of the programming process. The analysis is inspired by the referent/model system approach from

Introduction

the DELTA-project /Hanssen et al./ and the discussion of central aspects of the programming process in /Jørgensen et al./. The process-structure-function terminology from /Mathiassen/ is used throughout the analysis.

Chapter 2 discusses human concept understanding and abstraction with only little reference to the computer. The discussion of the Aristotelian and the fuzzy view of concepts is inspired by /Larsen/ whereas the main source concerning the different abstraction techniques is /Smith et al./.

In chapter 3 we present our model of program executions and discuss some fundamental aspects of programming languages. Our model is inspired by the Beta-project /Madsen et al./.

Chapter 4 represents the synthesis of chapter 2 and chapter 3. Here we give a detailed analysis of programming language support of the different abstraction techniques, based on our discussion of abstraction in chapter 2 and program execution model in chapter 3.

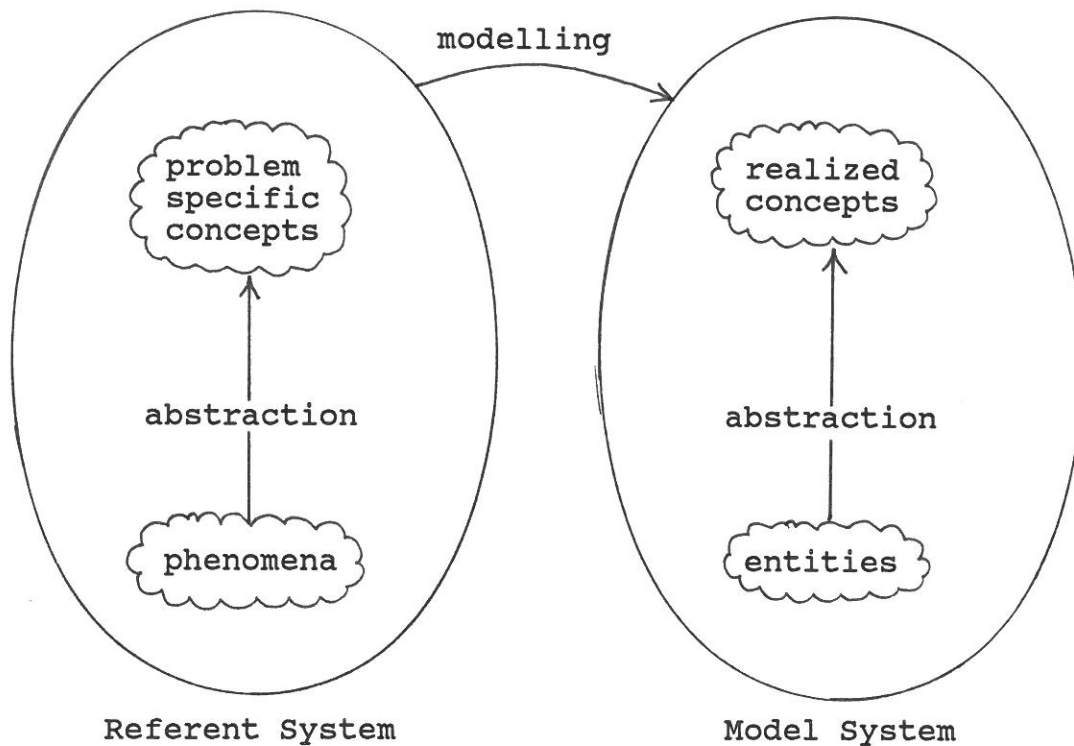
Throughout the paper we give numerous examples from existing programming languages. We give no explicit references to the individual languages when we discuss them, but references can be found in chapter 6.

CHAPTER 1

CHARACTERIZATION OF THE PROGRAMMING PROCESS

Let us start by emphasizing that programming is an intellectual human activity that can be characterized as a modelling process. By this we mean that a programming process takes its starting point in a referent system, which is usually a part of the real world or an abstraction of it, and seeks to create a model system, which is a program execution simulating a part of the referent system on a computer.

Let us picture this in the following way:



Chapter 1

Let us introduce the terms structure, process and function:

A structure is a collection of phenomena and/or concepts that we choose to consider as stable during some period of time.

A process is an activity that alters the state of some structures as time passes.

A function expresses the purpose of one or more processes - detached from their actual implementation and fulfillment.

A function can be fulfilled by one or more processes, and the same process can contribute to the fulfillment of several functions.

As stated above, the programming process is an intellectual human activity, and as such it has a purpose. That is, the programming process can be characterized by the functions that it must fulfill.

The previous figure identifies three important functions: abstraction in the referent system, abstraction in the model system and modelling.

Any process has some structures related to it. Some of these structures are altered by the process, whereas others represent the environment of the process. The environment influences and restricts the process.

In the first section of this chapter we will describe some structures related to the programming process. In the second section, the functions fulfilled by the programming process will be given a detailed description, based on the structures described in the first section.

1.1 Structures Related to the Programming Process

Let us start by considering some referent system structures related to the programming process:

In the referent system, a vision is formulated in relation to a problem area by means of problem specific concepts.

The Problem Specific Concepts

The problem specific concepts are concepts that are necessary to know in order to understand the problem area and the vision.

The problem specific concepts are not always well-understood and precisely defined.

The Problem Area

The problem area represents the domain of the programming process (e.g. "chess", "text processing", "process control", etc.). The problem area is understood through some problem specific concepts.

Example: the problem area "text processing" is understood in terms of the problem specific concepts "line", "page", "typographical quality", etc.

The Vision

The vision is a mental model of the program executions that are wanted as a product of the programming process. The vision is understood through the problem specific concepts.

Example: one vision could be to design and implement a screen-oriented text editor.

The first ideas of the program executions are denoted the initial vision.

Chapter 1

Some important characteristics of a vision are:

- Precision

The degree of precision is determined by the degree of formalization and the level of detail.

The degree of formalization depends on the extent in which artificial, unambiguous language is used for description of the vision. The level of detail depends on the amount of non-redundant information.

- Conceptual Distance to the Machine

The vision is formulated in relation to a given problem area and is intended to be realized on a given machine (for a characterization of machine, see later). Between the problem specific concepts and the realized concepts of the machine, there is a "conceptual distance". This distance can be used as an indication of the difficulties in modelling the vision in terms of the realized concepts. Note that we use the term "distance" in its abstract meaning - that is, not expressing any exact, measurable quantity.

An example showing two initial visions with different level of precision due to different level of detail:

- 1) A vision of a program execution that implements a text editor.
- 2) A vision of a program execution that implements a syntax-directed editor utilizing the graphic facilities of the PERQ computer.

An example showing two initial visions with different conceptual distance to a Pascal machine:

- 1) A vision of a program execution that converts a text in ASCII character code to a text in CDC character code.
- 2) A vision of a program execution that translates a text in some language (e.g. Pascal) into a text in another language (e.g. machine-language).

The problem area and the vision are structures in the referent system. The model system also contains some structures related to the programming process:

The model system is generated by a machine through a program written in a programming language.

The Machine

The machine is understood through the concepts that are made available by the programming language(s) used by the programmers during the programming process.

Any concept that has an interpretation that makes it contribute to program execution is called a realized concept. In addition, the realized concepts include those concepts in the programming language(s) that support abstraction - i.e. generation of new realized concepts.

As an example of a machine, we can talk about a "Pascal machine", i.e. a machine that can be programmed in Pascal. A "Pascal machine" is characterized by realized concepts like "integer", "array", "variable", "type" and "procedure". "Type" and "procedure" are realized concepts that support abstraction.

The Programming Language

The programming language is used by the programmers as a tool for writing programs.

A program is a specification used by the machine in order to generate some program executions.

The realized concepts that are available when the programming process is initiated are called the language defined realized concepts.

The programming language has special importance because it gives access to the language defined realized concepts and thereby determines the conceptual distance between the initial vision and the machine.

Chapter 1

The difficulties in realizing the vision on the machine is partly determined by this distance, but equally much by the support of abstraction provided by the language. The abstraction mechanisms in the language are used to create new realized concepts with a "minor" conceptual distance to the problem specific concepts by means of which the vision is understood.

1.2 Functions Related to the Programming Process

Now that we have described the structures involved, it is possible to give a description of the functions mentioned in the beginning of this chapter.

Abstraction in the Referent System

Abstraction in the referent system has the purpose of increasing the precision of the vision. This is done by establishing and modifying the cognitive structure of problem specific concepts.

The cognitive structure is developed in two directions. The first concentrates on the functional aspects of the vision - i.e. what the program is supposed to do. The other concentrates on the structural aspects of the vision - i.e. how the program is supposed to do it; the structure of algorithms and data. In both cases, new problem specific concepts may be developed by abstraction.

Example of abstraction in the referent system: Let us consider the construction of a syntax-directed editor. Then the functional aspects of the vision may be partly described by the concept : "Syntactically consistent editing", and the structural aspects of the vision may be partly described by the concepts : "Menus", "selection and expansion of nonterminals" and "abstract syntax trees".

Abstraction in the Model System

Abstraction in the model system has the purpose of creating realized concepts that make it possible to fulfill the modelling function. The new realized concepts are developed by abstraction, and their development is partly influenced by the insight gained about the vision, partly influenced by the requirements of the machine.

Example of abstraction in the model system: Creation of dynamic data structures that represent trees with a variable number of successors to nodes, and creation of procedures to manipulate such trees.

Modelling

Modelling is the function in which problem specific concepts (established in the referent system) are given a definition in terms of realized concepts (established in the model system).

Example of the modelling function: Using dynamic tree structures to model abstract syntax trees.

- oOo -

It is deliberate that we give no ordering of the three functions. In a given programming process it is often impossible to separate their fulfillment in time.

Modelling and abstraction in the model system will be subject to further discussion in the following chapters.

CHAPTER 2

CONCEPT UNDERSTANDING AND ABSTRACTION

In chapter 1 we showed that modelling of concepts and abstraction of new concepts from existing concepts are important functions of the programming process. In this chapter we will discuss how concepts can be structured and understood and what kinds of abstractions are important. Our purpose is to gain a deeper insight into the nature of the modelling and abstraction functions of programming.

First we would like to be more accurate about the terms concept and phenomenon:

A phenomenon is a thing that has definite, individual existence in reality (i.e. a manifest phenomenon) or in the mind (i.e. a cognitive phenomenon); anything real in itself.

A concept is a generalized idea of a collection of phenomena, based on knowledge of common properties of instances in the collection.

For example, "Person" and "War" are concepts, whereas "Winston Churchill" and "The Second World War" are phenomena.

There are three important characteristic aspects of every concept: the extension, the intension and the designation. We will start with an informal definition of these aspects:

The extension of a concept refers to the collection of phenomena that the concept somehow covers.

As an example, the extension of the concept "Human being" contains all persons.

The intension of a concept is a collection of properties that in some way characterize the phenomena in the extension of the concept.

As an example, consider the intension of the concept "Human being". Among others, it contains the properties: "able to think", "walks upright", "uses tools", etc.

The designation of a concept is the collection of names by which the concept is known.

The designation of the concept "Human being" contains in addition to the name "Human being" names like "Person" and "Homo sapiens".

Note that phenomena also have a designation whereas it is usually meaningless to talk about the extension or the intension of a phenomenon. However, sometimes it may be the case that something is considered a concept from one viewpoint but a phenomenon from another. That is, the distinction between concepts and phenomena is relative to a specific viewpoint.

As an example, a designer may consider "Chair" as a phenomenon, because from a designers viewpoint the individual chairs have no importance, whereas the general chair have properties interesting to the designer. On the other hand, the craftsman who actually produces the chair would consider "Chair" as a concept and the individual chairs as phenomena.

Often humans shift from one viewpoint to another when thinking about and discussing concepts and phenomena. We abstract from physical phenomena and consider a concept as a phenomenon whose properties we can discuss. Later on, we will find analogies to this ability in programming languages, but at present we will assume that the distinction between phenomena and concepts is clear and well-defined within the chosen viewpoint.

In the following two sections we will discuss concept understanding and abstraction, respectively, in terms of the extension and intension of concepts.

Thus, the use of the Aristotelian view of concepts results in a concept structure in which the concepts are characterized by sharp concept borders - since it is determinable from the intension whether a phenomenon is included in the extension of the concept. Furthermore, the phenomena in the extension of a concept are relatively homogeneous - since they all have at least the defining properties of the concept.

Discussion of the Aristotelian View of Concepts

The Aristotelian view is useful when one wishes to characterize the cognitive structure of concepts within well-established fields like mathematics, physics, zoology or botany. But even within these fields there are some problems, primarily concerned with the evolution of concepts as a consequence of acquisition of new knowledge.

Let us give a well-known example from the field of zoology. For many years it was one of the defining properties of the concept "Mammal" that they "bring forth living young", but the discovery of the Australian duckbill, which is an egg-laying animal with many properties in common with mammals, made the zoologists choose to change the intension of the concept "Mammal".

This example shows that although it is difficult, it is possible to use the Aristotelian view of concepts within well-established sciences, on the condition that one is willing to let the concepts evolve as new insight is gained.

But what then about the cognitive structure of everyday concepts like "Tall person", "Intelligent person", or "Food"? Considering "Tall person", it is not possible to give a precise definition of what a tall person is - someone taller than 175 cm? Considering "Intelligent person", we cannot agree upon how to judge intelligence - the IQ-test mainly measures academic knowledge, but is that sufficient? Considering "Food", we must take into account that whether something is characterized as food depends very much on social background and personal taste - are snakes food ?

Chapter 2

2.1 Two Different Views of Concepts.

In this section we will present two different views of concepts that are relevant when talking about programming: The Aristotelian view, which is well suited for computer models of concepts, and the fuzzy view, which is more realistic when talking about problem specific concepts in the referent system. The two views differ in their way of describing the intension of concepts and in their way of determining whether a particular phenomenon belongs to the extension of a particular concept. We present the two views as two extremes on a scale of possible views of concepts. That is, there exist a variety of views between the Aristotelian view and the fuzzy view.

2.1.1 The Aristotelian View of Concepts

The first of the two views has its origin in the classical Aristotelian logic. The main idea is that the extension of a concept contains phenomena that all satisfy some precisely defined requirements.

Let us characterize the intension and the extension according to the Aristotelian view.

The intension of a concept is a description of common properties of the phenomena in the extension. The nature of the properties is such that it is objectively determinable whether a phenomenon has a certain property or not. The properties may for instance be described by means of predicates.

The intension is divided into two parts: The defining properties that all phenomena in the extension must have, and the characteristic properties that the phenomena in the extension may or may not have.

The extension of a concept is the set of phenomena that have all of the defining properties.

Chapter 2

In summary, the Aristotelian view of concepts is usable when we consider cognitive structures within well-established fields such as natural science, but problems arise when we consider cognitive structures that cannot be precisely defined or that have evolved through everyday life.

2.1.2 The Fuzzy View of Concepts

The limitations of the Aristotelian view of concepts have caused the fuzzy view of concepts to evolve. This view seeks to capture the vague and fuzzy nature of some cognitive structures, by realizing that not all concepts have sharp borders and cover homogeneous phenomena.

Let us start by characterizing the intension and the extension of concepts according to the fuzzy view.

The intension of a concept describes examples of properties that phenomena may have, together with a collection of phenomena: the prototypes.

The intension is as such divided into two parts. The first part is a description of examples of properties. These properties are such that a phenomenon in the extension of a concept has at least some of them, and such that for each property there are at least one phenomenon in the extension that has it. The second part, the prototypes, contains characteristic phenomena from the extension; that is, typical examples of phenomena that are included in the extension.

The extension of a concept is a collection of phenomena that have some properties mentioned in the intension and to some extent resemble one or more of the prototypes.

In the Aristotelian view, the extension is completely determined by the intension, but this is not the case with the fuzzy view. Whether a phenomenon is in the extension of a concept is, in the

fuzzy view, determined by a decision or a choice. This decision or choice is necessary, since it is not sufficient to compare the phenomenon with the properties in the intension. It is also necessary to judge to what extent the phenomenon resembles the prototypes. There are no universal rules for how to make this decision and judgement - that is, there are no universally right or wrong answers.

Thus, the use of the fuzzy view of concepts results in a concept structure in which the concepts are characterized by blurred concept borders - since it is a matter of judgement whether a phenomenon is included in the extension or not. Furthermore, the phenomena in the extension of a concept are of varied typicality - since their similarities are rather vaguely defined in terms of some properties that they may have and some prototypes that they to some extent resemble.

As examples of fuzzy concepts, reconsider the concepts: "Tall person", "Intelligent person" and "Food".

2.1.3 The Modelling Function, Reconsidered

We have indicated that the Aristotelian view of concepts is in many cases too restrictive in its characterization of the intension and extension of concepts and thereby gives a too simplified view of concepts. Concepts that do not have sharp borders and do not cover relatively homogeneous phenomena are difficult or even impossible to describe using the Aristotelian view.

Such concepts are more adequately described using the fuzzy view of concepts.

In the light of the preceding discussion, we want to say something more about the modelling function involved in the programming process:

The problem specific concepts are often of a fuzzy nature. For example, the concept "Typographical quality" in the "Text processing" problem area is a fuzzy concept.

Chapter 2

On the other hand, the vast majority of realized concepts given by programming languages are precisely defined, Aristotelian concepts. This means that a critical aspect of the modelling function will often be to give an Aristotelian definition of problem specific concepts that have a fuzzy nature in the referent system.

For example, in the text processing example, we must define the concept "Typographical quality" in terms of fonts, lines, proportions, etc. although this definition does not fully capture the aesthetic intuition about typographical quality.

When a concept that has a fuzzy nature is given an Aristotelian definition as a result of the modelling function of a programming process, the concept is simplified and important aspects may be lost. Adding more details to the Aristotelian definition may improve the model, but will often not overcome the essential problem of the fuzziness of the concept, since the difference between a fuzzy and an Aristotelian definition is not only a quantitative difference but equally much a qualitative difference. For example, adding further criteria for typographical quality in terms of measurable properties of fonts, lines and pages still does not fully capture the aesthetic overall impression of a text.

No techniques to avoid the problem can be given, but of course we can try to make the models as good as possible to meet the needs of our application. In any case it is very important that we are aware of the limitations of our model when it is obtained by giving Aristotelian definitions of fuzzy concepts. Reasoning using the Aristotelian definitions may yield results that cannot be applied to the original fuzzy concepts.

When we are aware of the limitations of our model, we are able to determine whether it will suffice for our purpose or whether the limitations are so severe that the model system will be useless or even harmful. In some cases a better solution can be obtained by designing a model with extensive interaction with humans. This may for instance be the case in the text processing example where

interaction with a person who is a good judge of typographical quality may eliminate the need to model the concept of typographical quality directly on the computer.

Interaction with humans is one way of simulating fuzzy concepts on a computer. Other attempts to avoid the rigidity of the Aristotelian view exist. The programming language Fuzzy associates probabilities to the membership of entities to sets in order to avoid the sharp concept borders of Aristotelian concepts. Within the field of expert systems research is done on subjects like probabilistic and heuristic reasoning in order to simulate judgement within a fuzzy view of concepts.

Whether attempts in this direction will ever succeed in realizing truly fuzzy concepts on the computer remains an open question. In our discussion of programming languages we choose the Aristotelian view as our basis because no existing programming languages realize truly fuzzy concepts and it is unclarified whether programming languages ever will.

2.2 Abstraction

Concepts are developed by humans who distinguish patterns and recognize similarities between different phenomena. Before we have gained enough experience to recognize similarities, we perceive and apprehend the individual phenomena of the world.

As we gain more experience, we are able to abstract from some of the details that distinguish the phenomena and concentrate on similarities between them. We form concepts that cover collections of similar phenomena.

We abstract and concretize further by forming new more abstract and more concrete concepts based on other concepts. The result is a permanently changing cognitive structure of interrelated concepts.

We distinguish between three fundamental subfunctions of abstraction: Classification, aggregation and generalization, and

Chapter 2

similarly between their three inverse functions: Exemplification, decomposition and specialization, which are subfunctions of concretion. The functions establish different kinds of relationships between concepts and phenomena.

We will define the abstraction and concretion functions in terms of an Aristotelian view of concepts. This will enable us to discuss them more precisely, and since our purpose is to establish a framework for discussion of programming languages, we have already chosen to restrict ourselves to an Aristotelian view.

After the presentation of the abstraction functions, we will discuss their analogy to mathematical set operations.

2.2.1 Classification and Exemplification

To classify is to form a concept that covers a collection of similar phenomena.

The intension of the concept describes selected properties of the phenomena. The selected properties of the phenomena may, according to the Aristotelian view of concepts, be split into two parts: The defining properties that all the phenomena must have, and the characteristic properties that the phenomena may have. The extension contains the considered phenomena that all have the selected, defining properties.

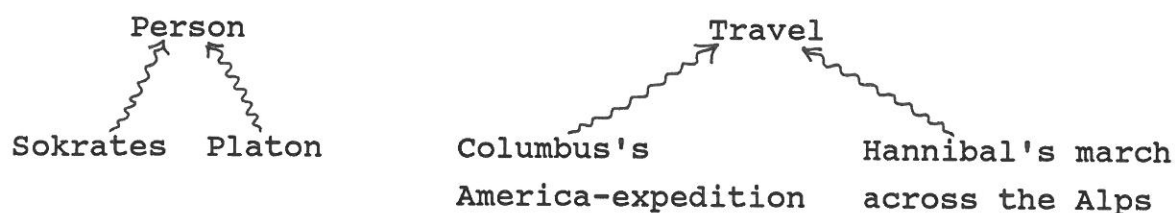
Examples:

The concept "Person" is formed to cover all phenomena with the properties "able to think", "walks upright", "uses tools", etc. The concept "Travel" is formed to cover all movements over long distances.

To exemplify is to focus on a phenomenon in the extension of a concept.

Exemplification is the inverse of classification.

When we talk about classification and exemplification, we focus on the relation between a concept and the phenomena in its extension. We will illustrate this relation graphically as follows:



2.2.2 Generalization and Specialization

To generalize is to form a concept that covers a number of more special concepts.

The intension of the general concept describes selected common properties from the intensions of the more special concepts. The extension contains the union of the extensions of the more special concepts.

To specialize is to form a more special concept from a general one.

Specialization is the inverse of generalization.

Examples:

The concept "Mammal" is a generalization of the concepts "Dog", "Cat", etc.

The concept "Movement" is a generalization of concepts like "Travel", "Jump", etc.

When we talk about generalization and specialization, we focus on the relation between a general concept and more special concepts. We will illustrate this relation graphically as follows:



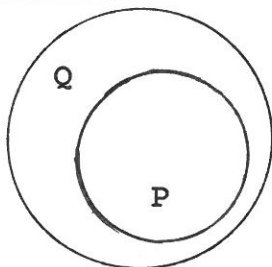
Chapter 2

Let us take a closer look at the role of the defining and characteristic properties in relation to generalization/specialization. A concept P is a specialization of another concept Q (Q is a generalization of P) iff the extension of P is a subset of the extension of Q.

This can be equivalently expressed in terms of the defining properties in the intensions: P is a specialization of Q (Q is a generalization of P) iff any x that satisfies the defining properties in P's intension also satisfies the defining properties in Q's intension.

That is, set inclusion between the extensions is equivalent to logical implication between the defining parts of the intensions.

Extensions:



Intensions:

Defining-intension(P)

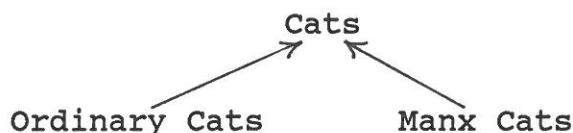


Defining-intension(Q)

Characteristic properties are properties that phenomena in the extension of a concept may have, but are not required to have in order to belong to the extension of the concept. It will therefore often be relevant to develop two specialized concepts that cover those phenomena that have a certain characteristic property and those that do not, respectively.

The first of these specialized concepts has the characteristic property changed into a defining property. The second has a new defining property stating that phenomena in the extension do not have the previous characteristic property.

Example:



Most cats have a tail, which is thus a characteristic property of the concept "Cat". The two specialized concepts "Ordinary Cats" and "Manx Cats" are defined as having and not having a tail, respectively.

Thus, generalization and specialization do not imply logical implication between the full intensions of concepts, only of the defining properties.

2.2.3 Aggregation and Decomposition

To aggregate is to form a concept that covers composite phenomena whose parts (aggregation components) are described by means of other concepts. We say that an aggregated concept composes a number of other concepts. Defining properties in the intension of an aggregated concept describe the components that phenomena in the extension must have, and perhaps some other properties that phenomena must have when considered as a whole - e.g. specific relations between the components. The intension may also contain characteristic properties that describe additional possible components and other properties.

Example:

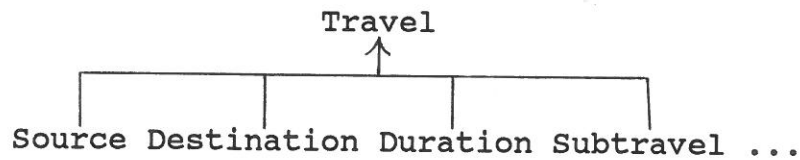
The concept "Travel" can be formed as an aggregation of the concepts: "Source", "Destination", "Duration", "Subtravels", etc.

To decompose is to focus on a component of a concept.

Decomposition is the inverse of aggregation.

When we talk about aggregation and decomposition, we focus on the relation between a concept and the concepts used to describe its components. We will illustrate this relation graphically as follows:

Chapter 2



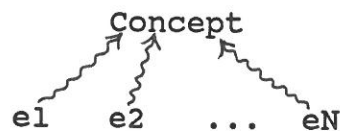
Note that properties of the components are not necessarily also properties of an aggregated phenomenon as a whole. That is, the aggregation relation between concepts does not imply inheritance of properties as opposed to the generalization relation.

2.2.4 Analogies to Mathematical Set Operations

There is a very close analogy between the abstraction functions discussed above and mathematical set operations, when we focus on the extensions of concepts:

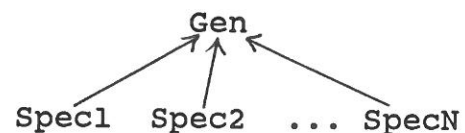
Classification corresponds to the basic grouping of elements into a set:

$\text{Concept} = \{e_1, e_2, \dots, e_N\};$



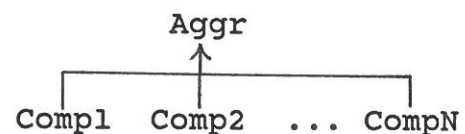
Generalization corresponds to creating the union of a number of concepts:

$\text{Gen} = \text{Spec1} \cup \text{Spec2} \cup \dots \cup \text{SpecN};$



Aggregation corresponds to creating the cross-product of a number of concepts:

$\text{Aggr} = \text{Comp1} \times \text{Comp2} \times \dots \times \text{CompN};$



However, to consider the abstraction functions only as set operations on extensions is not sufficient. To classify is not only grouping of elements - it is important that we focus on

similarities and describe common properties. To generalize is not only to create unions, since we may not always have special concepts whose extensions together span the whole extension of the generalized concept. Aggregation is not only creation of cross-products, since also relations between components may be important and some components may be only characteristic properties.

We have therefore defined the abstraction functions in terms of both extensions and intensions of concepts. However, the analogy to set operations can be used to convince us that the abstraction functions are really basically different. Moreover, the analogy can serve as an intuitive aid when dealing with concept structures.

2.2.5 Hierarchical Concept Structures

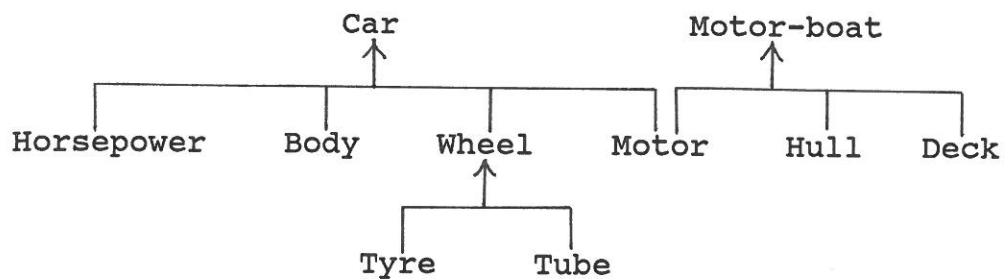
Concepts can be generalized and aggregated in many levels giving a complex hierarchical structure of interrelated concepts. A concept can be considered independently as a member of a generalization hierarchy and an aggregation hierarchy.

Within the field of knowledge representation in Artificial Intelligence, these two kinds of hierarchies are traditionally called "is-a" and "part-of" hierarchies, respectively. /Barr et al./

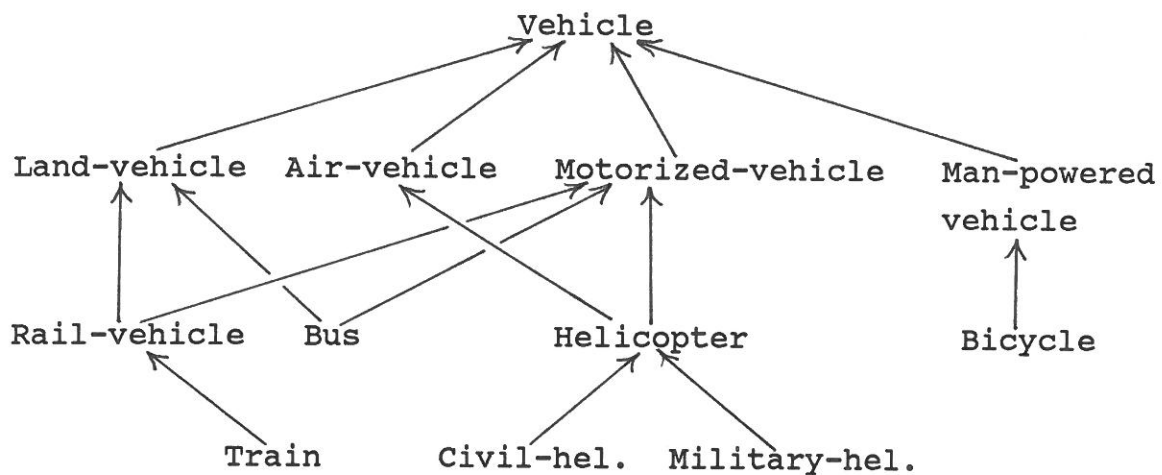
To give an impression of the complexity and possible structure of the concept hierarchies, the following page will contain a few examples where we separate the aggregation hierarchy from the generalization hierarchy.

Chapter 2

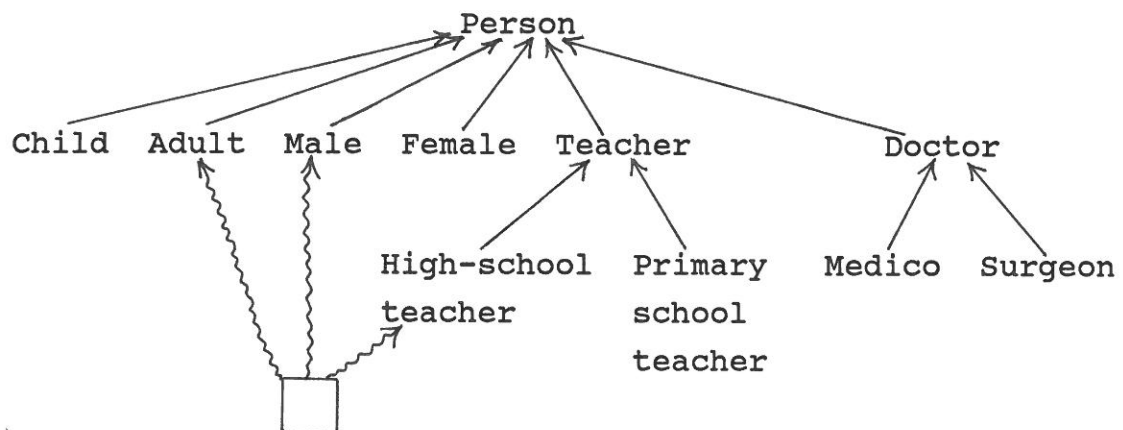
Example 1: Aggregation Hierarchy



Example 2: Generalization Hierarchy



Example 3: Generalization Hierarchy Including Classifications



2.2.6 The Abstraction Function, Reconsidered

This discussion of abstraction and concretion has resulted in the identification of three fundamental subfunctions of abstraction (and three fundamental subfunction of concretion).

We have discussed the subfunctions using an Aristotelian view in order to shorten the discussion. We believe that the same subfunctions can be identified using the fuzzy view although they would be less rigidly defined. In other words, we find that the subfunctions of abstraction and concretion that we have identified are of basic importance in the referent system.

This observation suggests that the same abstraction and concretion functions should be supported in the model system - that is, by the programming language. This will ease the modelling function considerably since the conceptual distance between the problem area and the programming language can be naturally bridged by means of the abstraction functions. Moreover, the structure of the realized concepts created as a result of the abstraction function in the model system will more directly mirror the concept structure in the referent system and thus make the program more comprehensible.

Several other advantages of supporting the abstraction functions in programming languages are discussed in chapter 4 - especially the advantages of supporting specialization/generalization.

CHAPTER 3

FUNDAMENTALS OF PROGRAMMING LANGUAGES

Basically, we consider a program execution as a number of processes that change the state of some substance in the computer. We use the term substance to cover all different kinds of storage in a computer - i.e. anything that can contain bit patterns. Substance is physically present in the computer and used in two ways: Firstly, substance can be used to represent values that processes can obtain by measuring upon the substance. Secondly, substance can be used to represent the state of processes (we say that the processes originate from the substance).

Besides this basic distinction between processes, substance and values in a program execution, we want to impose some further structure on our view of a program execution in order to ease the discussion of the relation between programming languages and program executions independently of physical computer architectures and specific language implementation techniques.

We consider a program execution as consisting of a dynamic collection of logical units, called entities and descriptors:

A descriptor is a pattern from which entities may be created and it gives the interpretation of the entities created from it. An entity created from a descriptor occupies some substance and is said to be an instance of the descriptor.

An entity behaves in accordance with its descriptor. The descriptor specifies which kinds of measurements (if any) that can be performed on the substance of the entity, and it specifies the behaviour of processes (if any) that can originate from the substance of the entity.

By means of the descriptor, the entity can be interpreted as a semantically meaningful entity at some level of abstraction instead of just a section of storage in the computer.

The relation between descriptors and entities is analogous to the model system relation between concepts and phenomena. We will therefore use the terms extension and intension for descriptors in much the same way as we do for concepts.

By considering a program execution as consisting of entities and descriptors, we can establish a direct relation to the programming language, since the programming language specifies the descriptors and entities relevant to the program execution.

Types and variables are examples of descriptors and entities, respectively. Similarly, procedure declarations and procedure activations are descriptors and entities, respectively, where the substance of a procedure activation is its activation record, and the procedure activation behaves according to the code given in the descriptor.

The rest of this chapter dedicates a section to further discussion of each of the important concepts: Descriptors, entities, values and processes. Moreover, a section discusses names, which play an important role both in programs and in program executions, and a final section discusses parameterization, which is a mechanism closely related to descriptors, entities and naming.

Chapter 3

3.1 Descriptors

The purpose of descriptors is twofold. Firstly, they act as patterns from which entities may be created, and secondly, they give interpretations of substance.

Types and procedure declarations are examples of descriptors in Pascal. Variables (entities) are created from types and the type determines the interpretation of the variable and the possible manipulation of the substance representing the variable.

In many languages the instantiation aspect and the interpretation aspect of a descriptor are very different. In for instance Ada, the package concept supports an explicit distinction between these two aspects: A package defining an abstract data structure gives rise to exactly one entity (which contains the datastructures and the operations defined in the package) and two descriptors. The first descriptor is the instantiation descriptor (the package body) and the other descriptor is the interpretation descriptor (the package specification). The rules for using these descriptors in Ada are that the instantiation descriptor is used for instantiation of the package, and for the instance's own interpretation of itself. The interpretation descriptor, on the other hand, is used by other entities when they want to interpret the package entity (or rather the substance of the package entity). In languages like Clu, Alphard, Concurrent Pascal, Edison, Modula-2, etc. there is a similar distinction, but not so explicitly expressed syntactically. In other languages - e.g. Pascal - this distinction is not made at all, implying that all details of the descriptor are visible when interpreting instances of it.

In our view of program executions, we choose to consider descriptors as occupying substance (e.g. a class in Smalltalk or the code for a procedure in most implementations of Pascal). We choose this view regardless of whether the actual implementation is able to eliminate the presence of the descriptors at runtime by extracting the needed information at compile time.

In some cases it is possible to consider the substance of a descriptor (or part of it) as representing an entity by interpreting it by means of another descriptor. E.g. in Lisp, a function definition is a descriptor but on the other hand it is possible to manipulate a function description by considering the definition as an ordinary Lisp list - that is, by using the Lisp list descriptor as an interpretation.

Another example is Smalltalk where classes are also objects. As opposed to Lisp, it is only those parts of the class that are not used for instantiation that can be manipulated. These parts may be used to represent properties of the extension of the descriptor as a whole, instead of properties of the individual entities - i.e. class-variables and class-methods in Smalltalk.

Both the Lisp and the Smalltalk examples show programming language support of the ability to consider a concept as a phenomenon, which was mentioned in the beginning of chapter 2.

3.2 Entities

As mentioned in the previous section, entities are created according to one (or more) descriptors. Entities are present in the computer as substance and the substance represents the state of the entity.

Examples of entities are variables in Pascal and objects in Smalltalk. Another example is a procedure activation (the activation-record and the corresponding procedure-execution).

One fundamental property of entities is that it may be possible to take a snapshot of the state of an entity. This is done by measuring on the substance of the entity in order to obtain a value, representing the state of the entity at the time of the measurement. Values will be discussed in section 3.3.

Another fundamental property of entities is that they may have processes originating from their substance. The only way in

Chapter 3

which processes can occur in a program execution is by originating from some entity. In other words, at least part of the substance of the entity is used to represent the state of the process and used to control the course of the process. Processes will be discussed in section 3.4.

In order for something to be regarded as an entity, it must have either the property of measurability, or of process(es) originating from its substance, or both.

An entity may be composed of other entities and/or descriptors. That is, the substance of the entity may be divided into parts that represent entities and/or descriptors that are components of the entity.

The above discussion might indicate that there is only one descriptor to an entity but this is not always true. In some cases it is for instance possible to use several different descriptors when interpreting the substance of an entity (e.g. classes at several levels of abstraction in Simula).

In most programming languages the concepts of data and process are handled as fundamentally different concepts. In order to ease discussion of such languages and in order to stress the differences of the two fundamental properties of entities already mentioned, we distinguish between two kinds of entities, namely data-entities and process-entities. In the following two subsections we will discuss these concepts and the interaction between them.

3.2.1 Data- and Process-entities

For both data- and process-entities we will give two definitions - a rigid one that defines the concepts very distinctly and a more flexible one that is adequate for most purposes.

As mentioned earlier there are two fundamental aspects of entities - measurability and originating processes. The rigid

definitions of data-entities and process-entities are based on having one and only one of these aspects:

An entity is a data-entity iff no processes originate from its substance and it is possible to take a snapshot of the state of the entity by means of some measurement on its substance.

An entity is a process-entity iff no values can be measured on its substance but one or more processes originate from (part of) its substance.

By this definition most ordinary datastructures from programming languages will be data-entities. E.g. the integer-variables in Pascal or the ordinary stack-variables in, say, Ada. A procedure invocation in Pascal and a generator instance in Icon are examples of process-entities according to this definition.

Entities in general are composed of data- and process-entities and are therefore hybrid forms. On the other hand, in many cases one can say that an entity is best characterized as a data- or a process-entity which is the same as saying that the entity gives rise to mainly data or process. At a higher level of abstraction it may be relevant to abstract away from the data- or process-aspect of an entity and consider it as a process-entity or data-entity although according to the strict definitions it is a hybrid form.

One such example is a heap with automatic garbage collection. One realization of this datastructure is to have a process local to the heap continuously doing "mark-and-scan" garbage collection. At some level of abstraction we would like to consider this kind of heap as a data-entity although it has a process originating from its substance.

Another example is a job-scheduler. One realization is as a process that schedules the different jobs but at the same time allows enquiries (measurements) about, say, the number of jobs of a certain category. In this case we may want to abstract away

Chapter 3

from the data-aspect - i.e. the measurability - and consider the job-scheduler as a process-entity.

As the examples show, we need more flexible definitions of the concepts: data-entity and process-entity, that reflect the possibility of considering entities at different levels of abstraction:

A data-entity is an entity that mainly gives rise to substance on which measurements can be performed.

A process-entity is an entity that mainly gives rise to substance from which process(es) originate.

3.2.2 Interaction of Data and Process

Data- and process-entities have no meaning if they are considered isolated from each other. Data-entities are not very interesting if they never change, and process-entities are not very interesting if they do not change the state of any data-entity. The interesting aspect of a program execution is the interaction between the two kinds of entities. The state of data-entities are changed by process-entities and the course of the process-entities may be influenced by the state of data-entities.

Some language constructs exist that support description of hybrid entities that are intended to capture this interaction between data and process. For instance the class-construct in Simula and the pattern construct in Beta.

In the definition of the entity concept we required that an entity must have either data-aspects or process-aspects or both. This requirement implies that program units that contain only descriptors but no data and process - e.g. a routine library - is not an entity according to our definition.

3.3 Values

One of the important aspects of the substance of an entity is that values can be obtained by measurement on the substance. In this section, we will discuss values and their relationship to programming languages.

Values are timeless abstractions that are unchangeable as opposed to entities that have state and can be created, copied, shared and updated. The value "3" cannot be changed whereas an entity with the state represented by "3" at some moment, may change and later on have state represented by some other value.

Values have some semantic meaning associated with them - e.g. values representing different colours or values representing names.

Values are represented by indivisible symbols, sequences of symbols, or tuples of symbols, or combinations hereof (e.g. sequences of tuples of symbols).

When we discuss values, we are in the world of mathematics where there is no state change but just values, value domains and pure functions.

We will briefly present the most important kinds of mathematical value domains together with the mathematical functions associated with these kinds of domains. Afterwards, we will discuss the relation between entities and values from a programming language perspective.

3.3.1 Value Domains

Scalar domains are the basic domains. They consist of symbols and a value from a scalar domain is as such indivisible. The only function common for all scalar domains is test for equality, but of course there may be other functions related to specific scalar domains - e.g. the arithmetic functions on the domain of Integral numbers.

Chapter 3

Union domains are unions of several other domains. The set of values in a union domain is the ordinary mathematical union of the involved domains. Associated with union domains in general is a domain-test function that tests whether a specific value belongs to a specific domain in the union.

Tuple domains are mathematical fixed length cross-products of some other domains. The values in a tuple domain are fixed length vectors of values from the component domains. Associated with tuple domains in general are a number of projection functions - one for each component domain. The *i*'th projection function projects on the *i*'th component domain - that is, it selects the *i*'th element of a vector value.

Sequence domains are mathematical variable length cross-products of some other domain. The values of a sequence domain are sequences of values from the component domain. Different values from the same sequence domain may have different lengths. No projection functions are associated with sequence domains, but six other functions are often considered applicable for sequence domains: "First", "trailer", "last", "leader", "length" and "concatenate". "First" returns the first element in a sequence value. "Trailer" returns a copy of a sequence value except for the first element. "Last" returns the last element. "Leader" returns a copy of a sequence value except for the last element. "Length" returns the number of elements in a sequence value, and "concatenate" takes two sequence values and returns a sequence value that are the concatenation of the two.

Examples:

Integer, Real, Boolean, Char and user defined enumeration types are Pascal examples of support of scalar domains, although the names of the types are not only the names of the associated value domains but also names of descriptors. The value domains are not named but anonymous properties of the descriptor.

Union domains can be found in for instance Algol68.

Arrays and records support (also anonymous) tuple domains as-

sociated with array types and record types. The projection functions are the indexing mechanism and the field selection mechanism, respectively.

Sequence domains are supported in a restricted version by the file concept in Pascal and by the list concept of Lisp.

3.3.2 Applicative Versus Imperative Programming

The term applicative programming is used for programming with values and functions whereas the term imperative programming is used for programming with entities and state.

Most programming languages take either the one or the other stand. On the other hand, most of them include both the value and the entity/state aspect. Most applicative languages include update-functions on values (e.g. assignment and variables in Lisp), and most imperative languages include the possibility of defining new value domains (e.g. the enumeration type constructor in Ada) and for specifying values from certain domains (e.g. integer, array and record values in Ada). But usually the integration of the two approaches is only halfhearted (e.g. Ada allows you to define values but includes no pure, side-effect free mathematical functions).

From a mathematical point of view, the applicative approach to programming is very appealing because the absence of state, state transformation and time makes it possible to reason and verify programs by means of well-defined mathematical principles. On the other hand, as we have argued in the first two chapters, state and time are important aspects of the referent system, and we therefore often need to be able to create models that include these aspects. Within many problem areas (e.g. data bases) usage of a purely applicative language will require that large values representing the state of the entire computation are continuously passed as parameters from one function to the next. This obscures the program and we find that it is far more elegant to ac-

Chapter 3

cept the notion of state and incorporate it in the programming language in a thorough way. That is, we prefer the imperative approach to programming. On the other hand, we do find that the mathematical elegance of the applicative approach is in itself a good argument in favor of treating values as an integral part of an imperative programming language. In /MacLennan/, a more detailed discussion of value vs. state is given.

The programming language ML (and to some extent APL) is an example of languages that seek to incorporate both imperative and applicative language elements. In ML, Values are treated just as thoroughly as variables, but unfortunately the imperative parts of the language may interfere with the applicative parts in such a way that reasoning using mathematical principles is hazardous without a very strict programming discipline.

We find that language design in the future should strive to find ways of unifying the two approaches into one language design where entities and values are distinct language elements of equal rights. Such a language will enable us to treat state in an adequate way, and to use mathematical reasoning within the applicative parts of a program.

We find that the descriptor-entity model presented in this paper is a sound basis for such a work since the concept of measurement on substance may act as the "missing link" between the imperative and the applicative approaches. The idea is to carry out a measurement on some entity in order to obtain a value. This value may then be passed to applicative functions in order to obtain other values that can be stored in some entities.

However, we must admit that our approach to programming language design is mainly imperative. We mainly focus on the imperative aspects and the possible improvements of the imperative aspects of languages, but we want to stress that a similar effort should be done to analyse the applicative aspects and the possible integration of the two approaches.

3.4 Processes

In the previous section we discussed the aspect of measurement on the substance of an entity within the framework of well-established mathematical concepts - that is, within a model which is commonly agreed upon.

Unfortunately, such a well-established model does not exist for processes. Many different models have been proposed but no common agreement has been reached concerning which model to choose. Among the most widely known models are the Petri-net model /Peterson/, the event-structure model /Nielsen et al./, Calculus for Communicating systems /Milner/, and Communicating Sequential Processes /Hoare/.

The various models tend to agree that processes can be modelled as collections of events with some relations between the events, where the relations normally expresses time dependencies. In some of the models (e.g. Communicating Sequential Processes) the events are atomic, indivisible symbols whose semantics are given by their written form (i.e. their written form should indicate the semantics that the programmer associate with them). In other models (e.g. Petri-nets) the events may in addition be abstractions of processes (i.e. events may have complex internal structure).

It is outside the scope of this paper to discuss the various models and their relative merits. In this paper we have used only the bare minimum with respect to a model for processes.

We have chosen to consider processes as a partially ordered set of state changing events where the ordering defines the ordering in time of the events. For the purpose of this paper this rudimental model has proved sufficient.

Our descriptor-entity model contributes to this model in the following way. A process-descriptor specifies the possible processes that may originate from the substance of an entity instantiated from the descriptor. The descriptor stresses the

Chapter 3

similarities of the patterns of events in all the processes. Within the descriptor the possible patterns of events are defined by the control structure, which is specified by means of language constructs for control (e.g. the alternative command, the repetitive command, and the parallel command in CSP). Many control structures express dependence between the state of data-entities and the course of the process. In this way, the actual course of a process is defined in a two-level manner: The descriptor for a process entity defines the possibilities, and the state of the computation defines the actual course of the process within the possibilities given by the descriptor.

3.5 Names

The only way in which the computer can effectively manipulate the entities and descriptors of a program is by giving each of them a name (e.g. an address). In a programming language, these names may be either hidden, computable or accessible as identifiers.

Hidden names cannot be specified by the programmer at all. Names can be hidden as a result of the scope rules of the language (see later) or because they are only relevant to the runtime system (e.g. the dope vector of an array).

Computable names are names that can be specified by means of expressions. They are usually used to refer to components of structures such as arrays, where the names of the components are not accessible as identifiers. Another use of computable names is relative naming such as "THIS Stack" in Simula.

The last category of names, the names that are accessible as identifiers, will be the subject of the rest of this section. Examples are names of static variables in Pascal.

For convenience, we say that entities and descriptors with identifier names are named, whereas entities and descriptors with hidden or computable names are said to be anonymous.

This enables us to give two useful definitions:

An entity is said to be category defined iff it is an instance of a named descriptor (e.g. a variable instantiated from a named type in Pascal).

and

An entity is said to be singularly defined iff it is an instance of an anonymous descriptor (e.g. a variable directly instantiated from a record-constructor in Pascal).

- oOo -

Identifiers are treated very differently in different programming languages and the differences are usually discussed in terms of the scope rules for bindings.

It is not the aim of this paper to give a thorough discussion of usage of identifiers in programming languages. We only give a short introduction to the subject and for a detailed discussion the reader is referred to chapter 15 in /Wulf et al./ and chapter 6 in /Tennent/.

An identifier occurrence is an occurrence of an identifier in a program text and is either a binding or an applied occurrence:

A binding occurrence serves the purpose of binding the identifier to a specific meaning - e.g. a value, an entity, or a descriptor. That is, binding occurrences establish bindings, which are associations between identifiers and meanings.

The associations between identifiers and meanings are not always required to be unique. Some programming languages allow overloading - i.e. several active bindings of the same identifier - and some programming languages allow aliasing - i.e. several identifiers bound to the same meaning.

Chapter 3

An applied occurrence is an occurrence where the identifier is used to refer to its meaning. An applied occurrence requires a corresponding binding to be found so that the meaning associated with the identifier can be used.

All declarations in Pascal contain examples of binding occurrences, whereas the identifiers in an assignment or in a procedure call are examples of applied occurrences.

Given an applied occurrence of an identifier, the scope rules of a language are the rules that determine how to find a binding occurrence of the same identifier. The corresponding binding is used as the one valid for the applied occurrence.

A great variety of different kinds of scope rules exist in various programming languages, but they can be divided into two major categories: Static scope rules and dynamic scope rules.

Static Scope Rules are rules that imply that the valid binding for a specific applied occurrence is determined by the lexical structure of the program.

Examples are the block-structured scope-rules of Pascal and Beta.

Dynamic Scope rules are rules that imply that the valid binding for a specific applied occurrence is determined by the dynamic structure of the program execution - i.e. the dynamic chain of program unit activations.

Lisp is an example of a language that uses dynamic scope rules. Ada is an example of a language that uses both static and dynamic scope rules. Static scope rules are mainly used in Ada, but a dynamic scope rule is used to find bindings of exception identifiers to exception handlers.

This formulation of the scope rules focus on the identifiers - that is, given an identifier, what is its binding. We could equally well have given a formulation of the scope rules that focus on the program unit - that is, given a program unit, which identifiers are usable (and with which bindings) within the given program unit.

Discussion of Scope Rules

The choice between static and dynamic scope rules is in some way a choice between security (in terms of compile time checking) and flexibility. The static scope rules enable the compiler to ensure that an applied occurrence of a name is bound to some meaning, and to some extent ensure that the name is used according to its binding. On the other hand, this security forces the programmer to be very explicit and rigorous with bindings. The dynamic scope rules allow a more flexible style of programming since an applied occurrence of a name does not need to be bound to the same meaning all the time. That is, it is the dynamic context of an applied occurrence of a name that determines its meaning. In some respect, use of dynamic scope rules is a shortcut to parameterization (see next section for a discussion of parameterization).

The major argument that can be given against dynamic scope rules is the complexity of name usage. In the static case, it is the lexical structure of the program that defines the meaning of a name. That is, one only has to examine a static structure in order to grasp the meaning of a name. In the dynamic case, one has to examine a dynamic structure - namely the dynamically changing structure of a program execution - in order to grasp the meaning of a name, and moreover, the meaning may change during the execution.

For a discussion of static versus dynamic scope rules in connection with exception handling, the reader is referred to /Knudsen a/.

3.6 Parameterization

We have chosen to discuss parameterization in a subsection of this chapter because it is a very fundamental mechanism and because it relates very closely to the discussion of descriptors, entities, and binding of names.

On the other hand, parameterization is usually considered as a

Chapter 3

fundamental abstraction mechanism, and this aspect of parameterization will be discussed in section 4.3.7.

From a very abstract view, parameterization is to specify a descriptor and leave out some parts of it in order to allow these parts to be specified either when the descriptor is used for specifying other descriptors or for instantiating entities.

When a descriptor is parameterized, it contains some formal parameters, also called formals, that are used in the descriptor to denote the partially specified parts. A parameterized descriptor cannot be used for instantiation of entities without first supplying actual parameters, also called actuals, for the formals.

Usually, a parameterized descriptor contains a formal parameter specification that specifies the name of the formal parameter. Moreover, the formal parameter specification may specify some constraints that the actual parameter must satisfy.

An actual parameter specification must complete the binding of the formal name by specifying the descriptor, the entity or the value to be bound to the name of the formal. Of course, the actual parameter specification must satisfy the constraints in the formal parameter specification, if any.

In many cases it is possible to specify default parameters together with the formal parameters. A default parameter specification is an actual parameter specification that becomes valid if no actual parameter specification is given later.

A formal parameter specification together with its corresponding actual parameter specification form a total binding of the name of the formal parameter.

The popularity of parameterization in programming languages has resulted in a great variety of parameter passing mechanisms such as call-by-value, call-by-reference, call-by-name, type-parameters and procedure-parameters.

In many programming languages the parameter passing mechanism to be used for a particular formal is specified either as an overall

rule of the language (a default rule) or as part of the constraints in the formal parameter specification. In such languages compile time checking of the parameterized descriptor is possible to some extent (we will later discuss some problems related to compile time checking of parameterized descriptors).

One typical constraint on the formal expresses whether the actual must be a descriptor, an entity, or a value. Other typical constraints are type constraints on entity parameters.

However, in some programming languages (typically typeless languages such as Icon) the constraints are very weak. This makes compile time checking very difficult - if not impossible - and leaves a great responsibility to the run time system. The run time system has to check the legality of all manipulations of the parameters.

Compile time checking of the parameterized descriptor and all usages of it has the advantage that it is unnecessary to check the legality of the manipulations of the parameters at run time. There are two major approaches to compile time checking.

The most restrictive approach to compile time checking is the above mentioned formal parameter specification method that enables the compiler to check that the formal within the descriptor is used in accordance with its specification. The formal parameter specification tells something about the intention of the programmer and the compiler checks whether the usage of the formal within the descriptor is in accordance with this intention. Furthermore, the compiler is able to check whether all actual parameter specifications are in accordance with the formal parameter specification. The parameter passing mechanisms of languages like for instance Algol60, Pascal and Ada are based on the formal parameter specification method.

The other approach is the formal parameter inference method. In this case, no formal parameter specification is given as part of the descriptor (except usually the names of the formals - either as user defined names or as language defined names). The compiler will then infer the specifications from the usages of the formals

Chapter 3

within the descriptor and then associate this specification with the formal. This inferred specification is then used to check the legality of the actual parameter specifications. During the inference process, the compiler usually checks the usages of the formal for consistency with respect to the inferred specification. As an example, the parameter passing mechanism of the programming language ML uses compile time inference of the formal parameter specification if no explicit specification is given.

In order to discuss further characteristics of parameterization, we divide the discussion into three: Parameterization by values, parameterization by entities, and parameterization by descriptors.

3.6.1 Parameterization by Values

The purpose of parameterizing a descriptor by a value is to allow different instances of the descriptor to use the name of the parameter to refer to different values.

Of course, parameterization by values is a major parameterization mechanism in the applicative programming languages. Most imperative programming languages, on the other hand, either do not support parameterization by values (e.g. Pascal, Algol60, Icon) or they support it the same way as the value parameter mechanism of Concurrent Pascal: Within an instance of the parameterized descriptor the name of the formal is bound to a value but the actual may be a value or it may be an entity.

3.6.2 Parameterization by Entities

The purpose of parameterizing a descriptor by an entity is to allow different instances of the descriptor to access different other entities.

When the formal is bound to an entity, it is interesting to study the relationship between the formals and the actuals:

The name of the formal may be bound to either an existing entity, or to a new entity.

- If the formal binds to the same entity as the actual, the actual parameter specification must specify an entity and the formal will be bound to this entity (e.g. the call-by-reference parameter passing mechanism).
- If the formal binds to a new entity, the actual parameter may be either an entity or a value. The purpose of the actual must be at least one of the following:
 - . The substance of the new entity bound to the formal is initially given a state represented by the value of the actual at the instantiation time of the parameterized descriptor (e.g. the call-by-value parameter passing mechanism).
 - . If the actual is an entity, there is also the possibility that immediately before the instance of the parameterized descriptor ceases to exist, the state of the entity bound to the formal is transferred to the state of the actual (e.g. the call-by-result parameter passing mechanism).

3.6.3 Parameterization by Descriptors

The purpose of parameterizing a descriptor by a descriptor is to allow different instances of the parameterized descriptor to use different actual descriptors.

Parameterization by descriptors appears in several programming languages. In Pascal, parameterization by procedures and functions is parameterization by descriptors. In Ada, generic parameterization by types is parameterization by descriptors. The parameter passing mechanisms call-by-name and call-by-need are also examples of parameterization by descriptors, since the

Chapter 3

parameters in these cases are expressions to be evaluated within the parameterized descriptor.

It is important to note that parameterization by descriptors introduces some new problems if we want a compiler to check the legality of the uses of the parameters.

When an entity is used as a parameter in a typed language its descriptor is usually known in advance (specified in the formal parameter specification). This makes it possible for a compiler to check the legality of the uses of the entity parameter.

When a descriptor is used as a parameter it is equally important to know at least some of the properties of the formal descriptor in order to ensure legal use. As most languages do not allow descriptors for descriptors, these properties must be specified in some other way, or inferred.

In the following, we will give three examples that illustrate compiler problems with descriptors as parameters. The first two illustrate that information about the formal descriptor parameter is needed when it is used to create and manipulate instances. The inference method is able to infer the needed specification in these cases, so our discussion will concentrate on how to enable compile time checks by means of specification. The third example illustrates a problem in connection with definition of new descriptors based on the formal descriptor. The inference method will not always be able to infer the specification in this case, so our discussion is relevant to both the inference and the specification approach to compile time checks.

Programming languages that accept extensive run time checks typically ignore the problems and leave legality checking to the run time system.

The Parameter Specification Problem

The parameter specification problem can be illustrated in connection with routines as parameters to routines.

If a parameterized routine calls its actual routine parameter, it

must specify the right number and types of actual parameters to this routine. If this is to be checked at compile time based on specification and not inference, it is necessary to specify the number and types of the formal parameters of the formal routine parameter in the declaration of the parameterized routine.

Pascal allows routines as parameters to routines, but the Pascal Report /Jensen et al./ does not require specification of the formal parameters of the routine parameter. This has lead to problems with run time checks, so most implementations of Pascal and the ISO Pascal Standard require specification of the formal parameters of the routine parameter.

This problem can be generalized to all descriptors with parameterized descriptor parameters. If an instance of the parameterized descriptor creates instances of the parameterized descriptor parameter, it must know the number and kinds of formal parameters to the parameterized descriptor parameter. That is, the specification of the formals must include enough specification to ensure legal use of the formals in the parameterized descriptor.

The Operation Problem

The operation problem can be illustrated in connection with types as parameters to routines.

If a routine with a type as parameter manipulates variables of the parameter type, it must manipulate these variables in accordance with their type. For example, if the routine assigns to a variable of the parameter type, it must be checked that assignment is a legal operation for all types used as actual parameters to the routine.

If this is to be checked at compile time, then it must be possible to ensure that the manipulations of the variables are legal for all possible actuals. In order to make this possible by means of the specification method, it must be specified in some way, which operations the routine requires the type parameter to have.

Chapter 3

For example, it should be possible to specify that the operations "assignment" and "test for equality" must be legal operations on variables of the parameter type.

In general, if a descriptor is parameterized by a descriptor, it must be specified how instances of the parameter can be manipulated. Again, this can be ensured by the formal parameter specification.

In Alphard, this is done by enumerating the legal operations as part of the formal parameter specification. In Ada, the operation problem is solved by elaborate rules stating which operations are legal by default, depending on the formal parameter specification. Additional operations must be given as explicit additional parameters. In section 4.3.1 we introduce the concept of qualification that solves the operation problem in an elegant way.

The Recursion Problem

This problem can also be illustrated in connection with types as parameters to routines.

Recursive call of a routine with a type parameter can lead to types with an indeterminable level of complexity. This makes it complicated for a compiler to know and check all types in the program.

Example:

```
Procedure Pip( Type t )  
  Type t' = Array [1 .. 10] of t;  
Begin  
  ...  
  Pip( t' );  
  ...  
End
```

In general, if a recursive descriptor uses a data-descriptor parameter to define new data-descriptors, then data-descriptors with an indeterminable level of complexity may be created. There

is no way of avoiding the problem unless dynamic instantiation of such parameterized descriptors is eliminated or compile time checking is dropped /Gehani/.

In Ada, the problem is solved by a binding "trick" that effectively eliminates the possibility of recursive instantiation. The trick is that outside the generic unit, the name of the generic unit is bound to the generic unit, whereas within the generic unit the name is bound to the actual instantiated unit. This makes it impossible to instantiate the generic unit recursively, since there is no way of referring to it within an instantiated unit.

CHAPTER 4

ABSTRACTION IN PROGRAMMING LANGUAGES

This chapter is intended as a synthesis of chapter 2 and 3. The abstraction functions identified in chapter 2 will be analysed in a programming language perspective by means of the terminology and descriptor-entity model introduced in chapter 3.

As mentioned in chapter 1 the conceptual distance between a programming language and a problem area is determined by two factors:

- the language defined realized concepts
- the support of abstraction

By support of abstraction we mean the ability to create a named descriptor that models a concept that we want to realize. This involves two steps: Firstly, the ability to describe the wanted concept in terms of the given language and secondly, the ability to encapsulate this description in a named descriptor. If only the first step is possible, we say that the concept can be simulated, but not obtained by abstraction by the programming language.

For instance, any programming language with a mechanism for asynchronous communication can simulate synchronous communication by means of a certain programming discipline, but in order to make it possible to realize synchronous communication by abstraction in a programming language it should be possible to name (and perhaps parameterize) a descriptor that describes the discipline for synchronous communication. When such a descriptor is used to create instances, the communication will automatically behave according to the discipline encapsulated in the descriptor, and thus enforce synchronous communication.

If a programming language contains language defined concepts that make simulation of the problem specific concepts very easy, this

can to some extent compensate for the lack of good abstraction mechanisms. However, since the optimal is to be able to abstract and since abstraction depends on the ability to simulate, we will concentrate on abstraction and not discuss simulation any further.

We will discuss abstraction in programming languages by focusing on the three abstraction functions: Classification, aggregation, and generalization and the inverse concretion functions.

4.1 Classification and Exemplification

In chapter 3 we discussed the descriptor-entity model and the instantiation relation between descriptors and entities. Furthermore, we pointed out that descriptors and entities in many ways are analogies in the model system of the referent system's concepts and phenomena, respectively. Then, looking at the discussion of the abstraction function in chapter 2, it is apparent that classification and instantiation are closely related. In the following we will discuss classification and exemplification in programming languages.

The way in which classification is supported in programming languages is by allowing creation of new descriptors. When creating a new descriptor, we classify all future entities, instantiated from or interpreted according to the descriptor.

Exemplification, on the other hand, occurs when an entity is instantiated from a particular descriptor, when an entity is selected as an instance of a particular descriptor, and when an entity is interpreted according to a particular descriptor.

From the above it can be seen that classification is very fundamental - it is impossible to specify a descriptor without involving classification (just as it is impossible to think of a concept without classifying).

Chapter 4

Examples:

Definition of a new type in Pascal and definition of a new generator in Icon are examples of classification. Declaration of a variable of the type and creation of an instance of the generator are examples of exemplification.

- oOo -

Programming languages usually contain some language defined descriptors (i.e. language defined classifications) and some language constructs for definition of new descriptors (i.e. user defined classifications). In the following, we will discuss some of the most common language constructs for simple classification of data and process, respectively.

4.1.1 Classification of Data

The oldest and most common approach to classification of data in programming languages is by means of a number of language defined descriptors, such as Integer, Character and Boolean in Pascal. Each of these descriptors models entities in a way inspired by the scalar domain of values. The approach is that the substance of the entities instantiated from the descriptors may have states that, when measured, correspond to values in the corresponding scalar domain. Moreover, the language defines some operations on the entities which make state changes possible.

As an example, the Boolean descriptor is based on the scalar domain (false,true) with the ordering: false<true, and several operations, such as "not", "and", "or", and "assignment". If we ignore the assignment, we could say that the Boolean descriptor defines a new scalar domain, boolean=(false,true), with associated functions "not", "and", "or" and "<". Moreover, the Boolean descriptor describes how to instantiate entities whose substance may describe boolean-values and furthermore defines the state changing operation "assignment". An equivalent discussion

could be given for any of the other descriptors mentioned above.

Another way in which programming languages support classification of data is by providing some language constructs for specifying new descriptors. The simplest such language construct is the enumeration type constructor^{*)}. The enumeration type constructor (e.g. in Pascal and Ada) is used for specifying a descriptor which, in addition to the normal purpose of descriptors, defines a new scalar domain consisting of the symbols enumerated in the constructor.

Common to the simple descriptors above is that they describe entities whose substance can represent values from a specific mathematical scalar domain (see section 3.3). Moreover, they may describe some additional mathematical functions and some state changing operations. In other words, the intension of a simple descriptor is first of all the property that measurements on the substance of entities instantiated from the descriptor result in values from the specified scalar domain, but in addition, the intension also specifies the existence of the additional mathematical functions and the state changing operations.

All of the above mentioned descriptors are usually called simple descriptors or enumeration types. These simple descriptors form the basis for definition of new descriptors by means of the other descriptor constructors such as the array-constructor, the record-constructor and the class-constructor, which all involve aggregation as well as classification.

4.1.2 Classification of Process

The most common approach to classification of process in a programming language is that the language provides some constructors that allow the programmer to specify simple and aggregated

^{*)} We use the term x-constructor instead of the phrase: the language construct which makes it possible to specify an x-descriptor.

process-descriptors. Examples of constructors for simple process-descriptors are the assignment-constructor in Pascal and the communication-constructs in CSP, which both result in anonymous simple process-descriptors when used by the programmer.

Examples of constructors for aggregated process-descriptors are the procedure-constructor in Pascal and the process-constructor in Concurrent Pascal, which both result in named process-descriptors. The block-constructor in Algol60 and the while-constructor in Pascal both result in anonymous aggregated process-descriptors.

Moreover, a language may provide some language defined simple named process-descriptors that can be used directly or as aggregation components in an aggregated descriptor. Examples are the "Read" and "Write" primitives of Pascal. For a further discussion of aggregated process-descriptors we refer to section 4.2.3.

- oOo -

A language may also support classification of hybrid entities by providing constructors that allow the programmer to define aggregated descriptors that describe both data and process components. The class-constructor in Simula is an example of this.

We will not discuss classification any further, but for a full understanding of classification the following sections on aggregation and generalization are also important, since classification is also involved when creating a new named descriptor by aggregation or specialization/generalization.

4.2 Aggregation and Decomposition

Simple descriptors were discussed in the previous section. When discussing composite descriptors we choose to distinguish between two different kinds of properties that can be used to specify the

intension of a descriptor in a programming language: Aggregation components and constraints on the components, which will be further discussed in section 4.2.1 and 4.2.2.

To aggregate is to create a descriptor by specifying some components that instances of the descriptor must have. Moreover, aggregation usually involves association of some constraints to the components - constraints that instances must satisfy.

Note that in general properties of the components are not inherited by the aggregated descriptor and its entities. For instance the values that can be measured on an aggregated entity as a whole need not be the same as the values that can be measured on its components.

Aggregation components may be named or anonymous. As examples, the components of a record in Pascal are named whereas the components of an array in Pascal are anonymous (in the absence of aliasing).

Most language constructs that support aggregation also support decomposition to some degree. Let us assume that we have an aggregated entity. Then decomposition may take different forms. The most common form is that the scope rules of the language (see section 3.5) allow access to the named aggregation components of the entity (e.g. the field selection of record entities in Pascal). Another very common form is that the descriptor constructor implicitly defines the access to the anonymous aggregation components (e.g. the subscription operation on array-entities). Finally, it may be possible to define the access in the descriptor for the aggregated entity by means of the so-called selector functions (see chapter 7 in /Tennent/). When invoked, a selector function returns a name of an aggregation component and thereby gives access to it. A selector function is usually defined in terms of process-descriptors.

It is important to note that it is by no means always the case that an aggregated entity can be fully decomposed (e.g. only the

Chapter 4

aggregation components marked "Entry" in a Concurrent Pascal class can be accessed). There is an analogy between decomposition and the projections functions of the tuple-domains, and the different functions of the sequence domain. However, in some cases decomposition allows changes to be made to the state of the data-components, whereas the domain functions return values only.

Examples:

The array and record constructs in Pascal support aggregation. There is one selector function for each projection function of the underlying tuple-domains, namely one for each component. The class construct of for instance Concurrent Pascal also supports aggregation. Here only the components marked "Entry" give rise to selector functions.

The procedure concept of many languages supports aggregation of a number of anonymous process components (statements) and a number of named data components (local variables).

- oOo -

In chapter 2, concept understanding and abstraction were discussed in terms of properties in general. The properties that are relevant when discussing descriptors and abstraction in programming languages are different kinds of aggregation components and constraints (e.g. local variables, local procedures/functions, and type constraints). Aggregation components and constraints will be discussed in the following two subsections. A third subsection contains a discussion particularly related to aggregation of process.

4.2.1 Different Kinds of Aggregation Components

We will discuss a number of different kinds of aggregation components that are relevant when describing properties in a programming language. They all have analogies to conceptually

different kinds of properties of phenomena in the referent system. However, we do not claim that our discussion is complete or that the kind of some property from the referent system or some component from an entity in the model system can always be uniquely characterized by one of our categories.

Dependent Entities

An aggregation component of an entity is called a dependent entity if it is an entity and its substance is part of the substance of the surrounding entity. That is, the dependent entity exists only as a part of the surrounding entity and cannot survive this entity.

Examples from programming languages are local variables in a procedure and components of a record variable.

To give referent system analogies we must either choose an abstract example - e.g. the voice, hearing and vision of a person - or assume a particular perspective and relative stability of the phenomena considered - then components like arms and legs of a person are examples of dependent components.

Dependent Descriptors

An aggregation component is called a dependent descriptor if it is a descriptor whose substance is part of the substance of the surrounding entity, and dependent on it as described for dependent entities.

Examples from programming languages are local procedures and local types of a procedure.

Analogies from the referent system are concepts that make sense only in a limited context - e.g. grammatical rules that make sense only in the context of a specific language.

Chapter 4

Relations to Independent Entities

An aggregation component may represent a named relation to another entity with an independent substance and existence.

An actual reference parameter of a Pascal procedure activation is a relation to an independent entity, since the existence of the variable does not depend on the procedure activation. Pointer structures in Pascal also represent relations to independent entities.

As a referent system analogy we can consider a person who may have a "has mother" relation to another independent person.

Relations to Independent Descriptors

An aggregation component may similarly represent a named relation to an independent descriptor.

A programming language example is the actual type parameter or routine parameter of a generic instantiation in Ada.

A referent system analogy could be the religion of a person - i.e. the named relation "Religion" to an independent concept representing a religion.

Measurable Properties

An aggregation component of an entity is said to represent a measurable property if it is a measuring method that delivers a value when it is activated. The value represents the result of the measurement on the substance of the entity, and the value need not be directly represented in the substance of the entity. The aggregation component representing the measurable property will typically be a dependent process-descriptor.

In programming languages, local functions, value-returning procedures and generators are examples of language constructs that can be used to describe measurable properties.

Analogies from the referent system are properties like height, weight and eyecolour of a person.

4.2.2 Constraints

A constraint can be associated with one or more aggregation components. It is specified in a descriptor and expresses some requirements that all instances of the descriptor must satisfy. That is, constraints express properties in the intension of descriptors.

The most common constraints known in programming languages are type constraints. A type constraint is associated with an aggregation component that is an entity, and expresses the requirement that the component must be an instance of a particular descriptor.

Example:

```
Type T = Record
      X : Integer;
      Y : Char;
      End
```

Other kinds of constraints on individual aggregation components can be found in existing programming languages. In some languages that allow descriptors as parameters to other descriptors the operation problem (see section 3.6.3) is solved by specifying explicitly the operations that the actual parameter must possess. The parameter is a relation to an independent descriptor, and the requirement of having certain operations is a constraint.

Example:

In Alphard, descriptors for abstract datatypes are called forms:

```
Form Stack( T : Form < ← >, N : Integer );
      .
      .
      .
      Endform
```

The type parameter T is required to possess an assignment operation, i.e. "←".

Chapter 4

Constraints can also be concerned with the relation between different aggregation components. This kind of constraint is not so common in programming languages.

Example:

In /Thinglab/, such constraints can be specified in a constraint-section of the class definition:

Class Line-With-Midpoint

Part-descriptions

Line : aLine
Midpoint : aPoint

Constraints

$(\text{Line Startpoint} + \text{Line Endpoint})/2 = \text{Midpoint}$
...

Other examples can be found in database specification languages /Astrahan et al./.

4.2.3 The Control Aspect of Process Aggregation

As mentioned in section 3.4, we consider processes as being composed of state changing 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.

Aggregation of process is concerned with formation of this partially ordered set of events. We aggregate processes by composing a number of component processes and define the partial ordering between the events of the component processes. The component processes may also be composite processes (i.e. aggregated processes) that define their own partial ordering of events. The aggregation then defines a new partial order of events based on the partial orders of the component processes. The composition of the component processes can be formed in several different ways, resulting in different partial orderings of the events in the ag-

gregated process. The partial ordering of the aggregated process is specified in the descriptor by means of language constructs for control (e.g. the alternative command, the repetitive command, and the parallel command in CSP).

We will in the following discuss some basic control mechanisms: Sequential composition, nested composition, co-sequential composition and concurrent composition:

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 come before all events from the other process.

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.

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 e_1 and e_2 from two different processes in a co-sequential composition are ordered either $(e_1 < e_2)$ or $(e_2 < e_1)$, but as opposed to sequential composition, not all such pairs need to be ordered the same way. Shift of execution from one process to another is made at the so-called changeover points (or control transfer points), which are explicitly marked in the process descriptor.

Chapter 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.

Discussion

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 (they share one thread of control) whereas co-sequential and concurrent processes are multi-sequential (the processes have individual threads of control).

We want to use the distinction between these composition forms to characterize different language constructs for aggregation of process. Therefore we want to be able to determine which composition form is supported by a particular language construct by considering only the semantics of the language and not the implementation. For instance, we will say that processes in Concurrent Pascal support 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 at the language level. We will therefore say that, conceptually, Concurrent Pascal only supports concurrent composition.

In general, we decide only to consider a language as supporting co-sequential composition of processes if the changeover points are explicitly specified in the process descriptor. An example of this is Simula's quasi-parallelism where resume and detach specify changeover points.

- oOo -

Within each composition form there are further interesting control aspects. We will not go into a detailed discussion of these aspects here but give a brief overview.

Control within uni-sequential processes is usually divided into three well-known main categories: Sequencing, selection and iteration.

Examples from programming languages are ";" for sequencing, "If-then-else", "Case" for selection, and "While" for iteration. A detailed discussion of uni-sequential control can be found in chapter 5 of /Tennent/.

Within multi-sequential processes in general, the important control aspects are synchronization and communication. The purpose of composing processes multi-sequentially is either that they must cooperate to do some task, or that they must share some resources, or both. Therefore, synchronization which enforces some mutual time control on the processes, and communication which enables exchange of information, are important. A further discussion of communication and synchronization mechanisms in programming languages can be found in /Thomsen et al./.

Moreover, some special control aspects concerning the changeover points of co-sequential processes are relevant. For a discussion of these aspects, the reader is referred to /Knudsen b/ and /Thomsen et al./.

Chapter 4

4.3 Specialization and Generalization

When we talk about the final hierarchy of descriptors obtained by specialization or generalization, it is both a specialization hierarchy and a generalization hierarchy depending on whether we choose to consider it top-down or bottom-up. During the programming process, however, it makes a difference whether the language supports specialization or generalization. Only very few language constructs exist for generalization, whereas a number of object-oriented languages support specialization. We will therefore only discuss specialization in depth.

Specialization means set-inclusion of the extensions, and implication between the defining parts of the intensions as described in section 2.2.2.

We talk about descriptors as describing aggregation components and constraints for the entities in their extension. That is, the defining properties in the intension of a descriptor are the properties of having some aggregation components and satisfying some constraints.

The way in which a programming language can support specialization is to allow definition of a specialized descriptor by mentioning the more general descriptor(s) and specifying the additional properties (aggregation components and constraints) in the intension of the new descriptor as compared to the general descriptor(s). The language should then let the properties from the general descriptor(s) be automatically inherited by the new specialized descriptor, so that the programmer need not specify the general properties more than once.

Examples:

Simula's prefix-mechanism supports definition of specialized descriptors called subclasses by adding new aggregation components:

```

Class Person
Begin
  Text Name;
  Text Address;

  Procedure New-address( A )
  Text A;
  Begin ... End;
End;

```

```

Person Class Student
Begin
  Text Subject;
  ...

  Procedure Examination;
  Begin ... End;
  ...
End;

```



Ada's subtypes support specialization by adding new constraints—e.g. a record subtype has an additional constraint concerning the value of the discriminant:

```

Type Vkind is ( Bus, Truck );

```

```

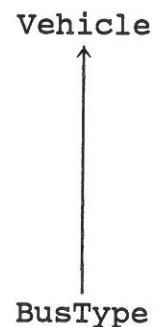
Type Vehicle( Kind : Vkind ) is
Record
  Registration-Number : Integer;
  Case Kind is
    When Bus => No-Of-Seats : Integer;
    When Truck => Max-Load : Integer;
  End Case;
End Record;

```

```

Subtype BusType is Vehicle( Bus );

```



- oOo -

It is generally accepted that aggregation is needed in programming languages, since otherwise no composite types or processes can be built. Specialization is not in the same way a basic necessity of a language, and many high level languages do not support it. Therefore we feel there is a need for a detailed discussion of the advantages obtained by supporting specialization in a programming language.

We will show that although specialization can be partially simulated by aggregation, we cannot obtain all the advantages of specialization by such a simulation.

After this discussion, we will return to a more detailed discussion of the interaction between aggregation and specialization due to the fact that specialization is often done with respect to

Chapter 4

the possession of certain aggregation components, as illustrated by the previous examples. Moreover, specialization of process needs a special analysis, as it was also the case for aggregation.

As the last aspect of specialization, we will discuss the possible structure of the resulting hierarchy.

We will finish the section by summing up the important aspects of specialization and analysing some existing language constructs.

4.3.1 Advantages of Specialization

The advantages of supporting specialization in a programming language are discussed in the following five subsections.

The first advantage is concerned with the ease of the modelling function. The next three are concerned with the support of structured programming methodology. Finally, the last, and perhaps the most important, is concerned with the expressive power of the language.

4.3.1.1 Conceptual Modelling

Since generalization/specialization are important mechanisms for structuring concepts in the referent system, a clear conceptual modelling can be obtained when specialization is supported in the programming language. This makes programming easier since the conceptual distance between the problem specific concepts and the realised concepts can be naturally bridged by means of the language mechanisms for specialization.

Moreover, it makes reading and understanding of programs - and thus program maintenance - easier, since the described concepts can be more easily recognized as models of problem specific concepts.

4.3.1.2 Stepwise Refinement

Specialization as a structuring mechanism is particularly well suited for applications where the complexity is due to a large amount of details, and where many related but different concepts are to be modelled.

Common properties of different concepts are described first, and then a stepwise refinement technique is used in connection with the development of a number of specialization hierarchies where additional details are described step by step down the hierarchy.

After the full program has been developed, the intermediate descriptors in the hierarchy are still part of the final program. This means that specialization, used as a stepwise refinement technique, directly supports that the intermediate descriptors can be used as documentation that is developed together with the program itself. Moreover, the intermediate descriptors can be used as the basis for reasoning about the properties of the final program, since all defining properties are invariantly satisfied all the way down the hierarchy.

Attempts to use specialization hierarchies as an aid in formal verification is described in /Wong/.

4.3.1.3 Factorization

Specialization supports that common properties of different concepts are factorized - that is, described only once and reused when needed. This results in greater modularity and makes complicated programs more comprehensible, since a lot of redundant description is avoided.

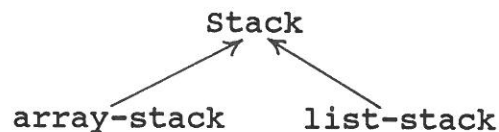
Intentional similarities between descriptors are made explicit so that they can be distinguished from accidental similarities. In this way it can be ensured that it is only possible to make use of the intentional similarities.

Chapter 4

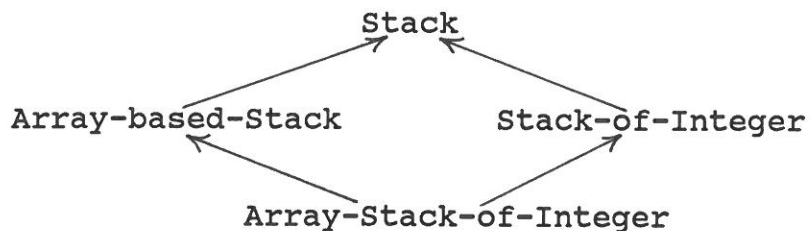
4.3.1.4 Separation of Concerns

When several more or less independent aspects of a concept are to be described, specialization hierarchies help separate the different concerns, so that the programmer (and the reader of a program) can focus attention on one aspect at a time.

For example, the abstract behaviour and the implementation of an abstract datatype can be separated by means of specialization. A general descriptor can abstract from the implementation and focus on specification, whereas one or more specialized descriptors can realize an implementation:



If two independent perspectives are relevant for a specific concept, the perspectives can be described in two different descriptors and inherited by a third:



4.3.1.5 Qualification: a Flexible Type-concept

With a specialization hierarchy of descriptors we can have qualifications instead of strict types when declaring entities, formal parameters, etc. in the following way: If an entity "e" is qualified by the descriptor "D", then "e" must be in the extension of "D", but it may also be in the extension of any specialization of "D". At this level of abstraction, however,

"e" can be manipulated only in accordance with the descriptor "D".

This kind of qualification mechanism is known for instance in Simula as qualified references. As an illustration of the flexibility and the expressive power gained by qualifications as compared to a traditional type-concept, we can consider the example of developing a program library that implements index sequential files. The library should be generally usable for index sequential files regardless of the kinds of elements to be stored in the file, assuming only the existence of, say, an integer "Key" in every element. The library could then contain a general descriptor called "Element" with only the defining property of having an integer component "Key". Besides, the library should contain the descriptor for index sequential files, whose elements are just qualified by "Element", and a number of procedures, whose formal parameters are qualified by "Element".

We illustrate this by means of a pseudo programming language:

```
Descriptor Element
  Key : Integer;
End Descriptor;
```

```
Descriptor Index-Sequential-File
  File : Array[ Range ] of Element;
  Position: Range;

  Function Index( Key : Integer ) : Range;
  Begin ... End;

  Function Get( Key : Integer ) : Element;
  Begin ... End;

  Procedure Put( E : Element );
  Begin ... End;
```

etc.

```
End Descriptor;
```

This library could for instance be used in a database application, where information about persons are to be stored with the civil registration number represented by the "Key" component.

Chapter 4

The application then defines a specialized descriptor from "Element" to model the concept "Person", and all entities that are instances of this descriptor will now be legal actual parameters to the procedures "Get" and "Put", since they are also instances of the qualifying descriptor for the formal parameters.

```
Descriptor Person
Is-specialization-of Element
(* with additional properties *)
  Name      : Text;
  Address   : Text;
  etc. ...
End Descriptor;

f : Index-Sequential-File;
k : Integer;
p1, p2 : Person;
...
f.Put( p1 );
p2 := f.Get( k );
```

Also entities belonging to the extensions of specializations of "Person" - if present - can be included in the file. If the language allows additional constraints to be added to inherited properties in a specialized descriptor, we could make a specialization of the descriptor "Index-Sequential-File", that allows only "Person" entities (and specializations) to be included in the file.

```
Descriptor Person-file
Is-specialization-of Index-Sequential-File
  Where Element Is-restricted-to Person;
End Descriptor;
```

Some of this flexibility can be obtained in languages that allow types as parameters, for instance by means of generic packages in Ada.

The index sequential file could then be a generic package with a generic parameter representing the element type and another generic parameter - a function - representing the requirement that a key-value must exist. The person-file could then be created as a generic instantiation with the relevant actual

parameters. However, such a person-file could contain persons only, not specializations of persons like for instance secretaries, truck-drivers and other kinds of employees. Of course, if all needed specializations of "Person" are foreseen when the descriptor is specified, "Person" can be defined as a discriminated record with a variant part, implying that the specializations described by this record can all be included in the database. However, it is impossible later on to make new specializations of "Person" such that instances of these specializations can also be included in the database. Thus, most of the flexibility of allowing the actual entities in the person-file to be heterogeneous is lost when choosing the Ada approach. On the other hand, it makes implementation easier.

A more important difference, however, between the specialization approach and the Ada approach can be illustrated by extending the example:

Suppose the general index sequential file is part of a standard library, supplied and owned by an independent software developer. Suppose moreover, that a customer who has bought this library often wants to dump different index sequential files on a dump-file. Obviously then, it would be desirable for the customer to be able to write a general dump-procedure that would accept any index sequential file as input parameter. This would be impossible in Ada without changing the generic package specification and thus violating the modularity of the system. Moreover, since the specification is the property of the software developer, it may not be available to the customer at all.

In a language based on specialization, this could be done very nicely by writing a procedure "Dump" with a formal parameter qualified by "Index-Sequential-File".

Flexibility of the kind just demonstrated is very valuable. So valuable that some language designers have chosen to abandon types completely - e.g. Smalltalk and Lisp - to avoid the

Chapter 4

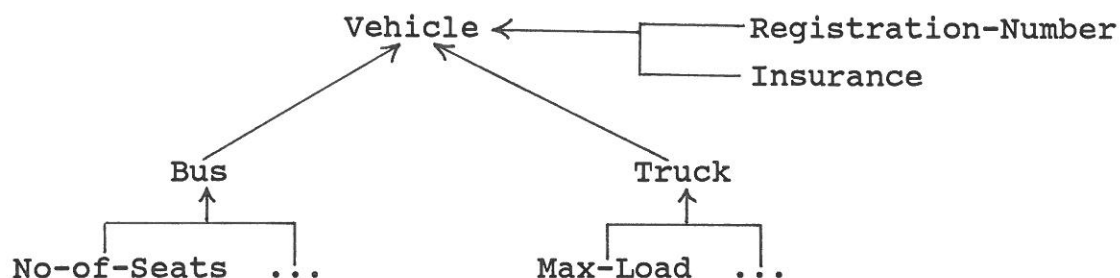
rigidity usually inherent in traditional type systems. However, when designing a type-less programming language, the security that types can give, is also renounced, and great responsibility is placed on the runtime system.

With qualifications based on a specialization hierarchy instead of a traditional type concept, we have gained additional flexibility, but we have preserved most of the security, since a compiler can still check that an entity is manipulated only in accordance with its qualifying descriptor. The specialization hierarchy guarantees that all defining properties of the qualifying descriptor are possessed by the entity also if the entity later on turns out to be an instance of more specialized descriptors. The defining properties of a qualifying descriptor represent invariant knowledge of any entity with this qualification, so that any manipulation in accordance with the qualifying descriptor will be valid for the actual entity.

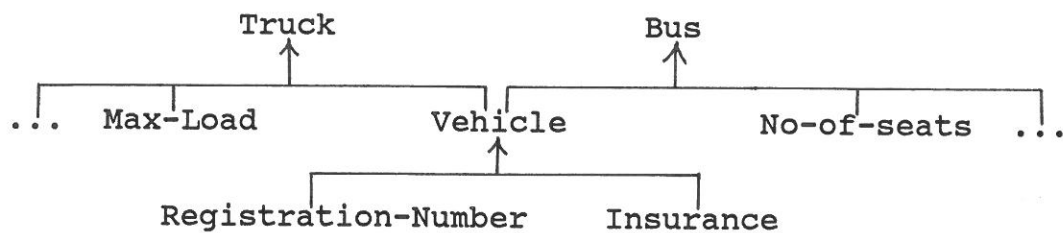
4.3.2 Simulation of Generalization by Aggregation

As already mentioned, specialization and aggregation are both conceptually basic, but from a technical viewpoint, aggregation is more fundamental than specialization. In the previous section, however, we presented the advantages of also supporting specialization. As a continuation, we will now show that these advantages cannot be obtained by aggregation alone, although specialization can be partially simulated by aggregation.

Example:



can be partially simulated by



The simulation is done by letting both the "Truck" and the "Bus" descriptors describe an aggregation component which is an instance of "Vehicle".

However, this simulation is not a very good one. The relations between the concepts have become confused, since it now seems that the relationship between "Registration-Number" and "Vehicle" is the same as the relationship between "Vehicle" and "Bus". Conceptually, these relationships are very different since busses are a subset of vehicles, whereas vehicles are certainly not a subset of registration-numbers.

It is impossible to obtain a qualification mechanism that is only valid for those part of the hierarchy that simulate specialization - e.g. the Vehicle-Truck-Bus part in the example above. Either we must abandon qualification totally or we must accept a qualification mechanism that covers the whole hierarchy. The latter forces us to allow also a "Vehicle" as actual parameter to a procedure where the formal is specified as "Registration-number", although this may not be what we want.

As shown above, a qualification mechanism which only follows the specialization hierarchy cannot be obtained if specialization is simulated by means of aggregation. Since such a qualification mechanism is one of the strongest arguments in favor of specialization, simulation by aggregation is insufficient.

Chapter 4

4.3.3 Use of Aggregation in Specialization

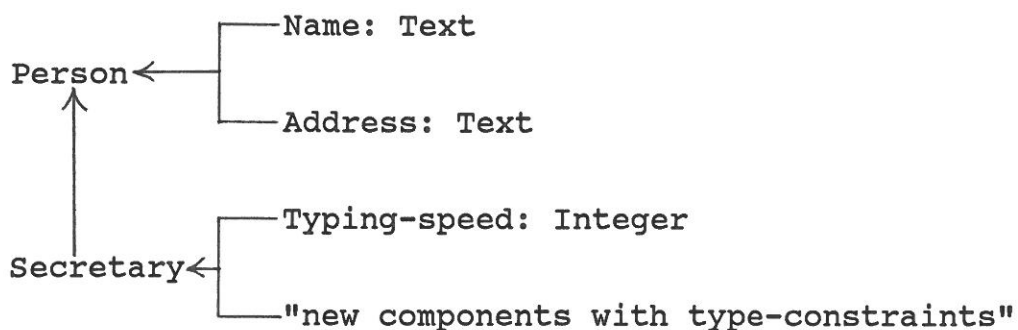
As already mentioned, specialization heavily depends on aggregation since specialization is done by adding new aggregation components and new constraints. In this section we will have a closer look at this interaction between aggregation and specialization. First we will discuss constraints added to aggregation components, and afterwards we will discuss the role of defining and characteristic properties.

4.3.3.1 Constraints on the Aggregation Components

A general discussion of the kinds of constraints that it could be relevant to specify on aggregation components was given in section 4.2.1. Here we will concentrate on constraints added in a specialized descriptor.

It is obviously relevant to be able to add constraints together with the addition of new aggregation components in a specialized descriptor. This is the basic possibility of the prefix mechanism in Simula.

Example:



It is also relevant to be able to add new (stronger) constraints on the old - the inherited - aggregation components.

The person-file example in section 4.3.1 illustrates the usefulness of this possibility, since "Person-file" is obtained as a specialization of "Index-Sequential-File" by adding further con-

straints to all the inherited "Element" components.

The person-file example shows that the possibility to add constraints to inherited components increases the expressive power of a specialization mechanism. There is also a maintenance argument in favor of this possibility as illustrated in the following:

It is important to notice that a concept is often specialized with respect to a specific property or a number of properties. For instance, the concept "Person" can be specialized with respect to "Job" into different categories like "Secretary", "Truck-driver", etc. The property "Job" of a person can be represented directly as an aggregation component of the person or indirectly by the person's membership of the extensions of more specialized descriptors.

In a language like Simula, where addition of constraints to inherited components is not allowed, a choice between these two representations would have to be made, and the criterion for the decision must be whether additional properties have to be modeled for persons with different jobs. If this is the case, two subclasses should be made and no aggregation component should represent the job of a person. If no additional properties are needed, subclasses are unnecessary and the job can be represented by a simple aggregation component.

If the simple representation is chosen and the application later on changes so that different additional information is to be stored for secretaries (e.g. typing speed and language knowledge), and for truck-drivers (e.g. type of licence), then this change will be very difficult to incorporate since besides defining subclasses to represent secretaries and truck-drivers, the "Job" component of class "Person" should be deleted.

Deletion of the "Job" component would have consequences also for all applications that operate on the abstraction level where the additional details of secretaries and truck-drivers are uninteresting - that is, applications that use the qualification

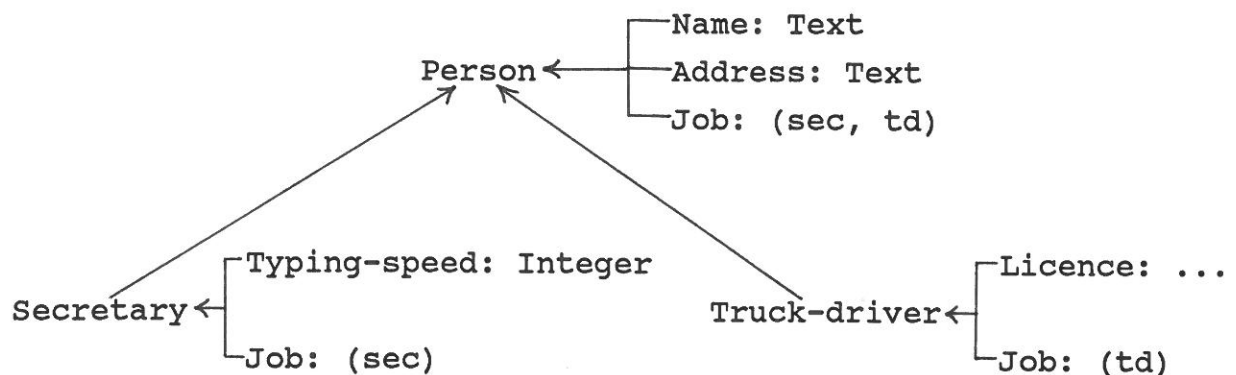
Chapter 4

"Person" and not the qualifications "Secretary" and "Truck-driver".

If the "Job" component is not deleted, a programming discipline must ensure that the value of the "Job" component of an entity is consistent with the class-membership of that entity.

If constraints can be added to inherited components, specialization with respect to a specific aggregation component can be made by adding constraints to this component and adding the relevant new aggregation components, without having to change from an explicit to an implicit representation of the component that causes the specialization.

Example:



With this model, applications that use only the qualification "Person" need not worry about the additional descriptors, since all persons still have all the original aggregation components.

4.3.3.2 Defining and Characteristic Properties

When we have been talking about specialization by adding new aggregation components and new constraints, we have mainly discussed defining properties. As the conceptual discussion of specialization in chapter 2 showed, the characteristic properties may also play a role in connection with specialization. If the programming language supports description of both defining and characteristic properties, it is relevant to be able to

specialize by promoting a characteristic property to a defining property or by introducing a defining property saying that entities in the extension of the specialized concept do not have the previous characteristic property.

An example is that we could be interested in considering the descriptor "Stack" as a specialization of the descriptor "Sequence", where some of the "Sequence" operations are only characteristic properties of "Sequence", and excluded from the specialized descriptor "Stack".

However, a mechanism for specializing with respect to a characteristic property is only useful if the language clearly distinguishes between defining and characteristic properties. If all properties are considered only characteristic and all properties may be excluded at lower levels in the specialization hierarchy, then the knowledge that a specific entity at least belongs to the extension of a specific descriptor, is useless since it does not allow us to assume anything about the properties of the entity. Thus the knowledge will not allow us to reason about the entity. To qualify an entity by a certain descriptor will give no security because the compiler is unable to deduce anything from this qualification about what is legal and what is illegal manipulation of the entity.

In summary, it is crucial to have defining properties in order for a specialization mechanism to be useful at all. In fact, most languages with specialization do support that some properties are defining.

Moreover, other interesting specializations are possible if characteristic properties are also supported.

4.3.4 Specialization of Process-descriptors

The relation between specialization and the defining and characteristic properties of a descriptor causes problems when considering process descriptors.

Chapter 4

There are two important aspects of a process:

- 1) The functional aspect - i.e. the effect of executing the process, measured in terms of the change of state of the involved entities.
- 2) The procedural aspect - i.e. how this effect is obtained by means of a sequence of state changing steps.

In most programming languages only the procedural aspect is explicitly described, whereas the functional aspect is implicitly given as the net effect of the performed steps. Conceptually, however, the defining property of a process concept is often its functional aspect and not its procedural aspect. For instance, the defining property of the stack operation "Push" will often be considered to be its effect of adding an element to the stack, not the concrete algorithmic sequence of steps used to obtain this effect.

Conceptually then, the most obvious way to define what we mean by specialization of a process would be in terms of the functional aspect. That is to require a specialized process descriptor to describe at least the same effect as a more general descriptor.

In a language where the functional aspect is explicitly described and not just implicitly as a result of the procedural aspect, such a notion of specialization could probably be realized - see for instance /Taxis/ and /Borgida/.

Most attempts to including specialization of process in programming languages have been made in languages that focus on the procedural aspects of processes. This means that the languages have chosen to let the procedural aspects be the defining properties of a process descriptor. This is by far the simplest in imperative languages, and when carefully used, it can still correspond to functional specialization and be a useful structuring mechanism for processes.

Example:

```

Procedure Access-shared-data
Begin
    Wait( Semaphore )
    Inner (* activates specialized body, if present *)
    Signal( Semaphore )
End

Procedure Update( Param )
Is-specialization-of Access-shared-data
Begin
    Data := ....
End

```

Procedure "Update" is a specialization of procedure "Access-shared-data" (both considering the functional and the procedural aspects) since "Update" always reserves the right to use the shared data, then updates the data and finishes by releasing the shared data.

However, a process descriptor that is a specialization of another with respect to its procedural aspects does not need to be a specialization with respect to the functional aspects. For instance, specializing a process descriptor by adding a procedural aggregation component in the form of a statement, does not necessarily result in a specialization from a functional viewpoint, since the additional statement may undo some of the effect of the original descriptor instead of adding new effect.

Specialization of process descriptors mainly with respect to the procedural aspect is described by /Vaucher/ and also realized in /Beta/. In /Thomsen b/ a more general mechanism for procedural specialization of process is introduced.

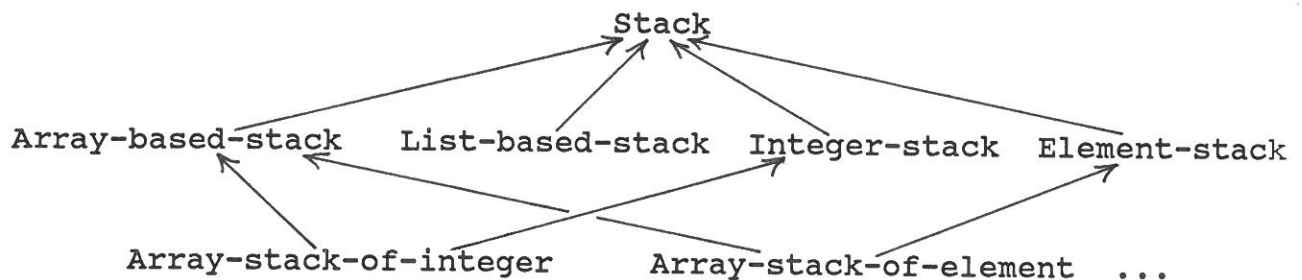
In summary, we would like that a language mechanism for specialization of process ensures specialization with respect to both the functional and the procedural aspect. However, it is our impression that further research is needed in order to find elegant solutions to this problem.

4.3.5 Single Versus Multiple Inheritance

If specialization hierarchies are restricted to be tree-structured we talk about single inheritance, since a descriptor can only inherit properties from a single more general descriptor (when we ignore the even more general descriptors). If the hierarchy can be an acyclic directed graph, we talk about multiple inheritance.

For a detailed discussion of the advantages of multiple inheritance and the different possible realizations of it, the reader is referred to /Thomsen a/.

The following figure illustrates a useful application of multiple inheritance:



The following procedures show the advantages of such a hierarchy, since procedures can be described in general for the whole range of stacks on which they make sense.

```
Procedure Bottom-up( St: Stack )
    (* reverses a general stack *)
```

```
Procedure Sum-of-elements( Ist: Integer-stack ): Integer
    (* returns the sum of the values on a general Integer-stack *)
```

```
Procedure Full( Ast: Array-based-stack ): Boolean
    (* returns true if the array limit is reached *)
```


4.3.6 Summary Concerning Specialization

As a summary, we will briefly mention the aspects of specialization that have been discussed in the preceding sections. The summary is intended to provide a basis for the understanding of the kind and quality of specialization mechanisms in different languages.

- 1) What kinds of defining properties can be added when specializing.
 - new aggregation components
 - new constraints
 - associated with new components
 - associated with inherited components
- 2) Does the specialization mechanism provide qualification instead of strong (or no) typing.
- 3) Is multiple or only single inheritance supported.
- 4) Does the language distinguish between defining and characteristic properties.
- 5) Is specialization of process (if possible) done with respect to the functional or the procedural aspect of processes.
- 6) Is the number of possible levels of specialization hierarchies limited or not. This aspect has not been discussed above, but it is obviously interesting whether the number of levels can be arbitrary large or not.

4.3.7 Analysis of Some Language Constructs

In this section we will analyse some language constructs that support specialization. We will characterize them in terms of the aspects summarized in the preceding section.

Chapter 4

4.3.7.1 Parameterization

When we have a parameterized descriptor, we can consider the specification of the formals as constraints on the parameters - e.g. type-constraints. From the preceding discussion of specialization it is then obvious that specialized descriptors could be obtained by adding further/stronger constraints to the parameters. Only few languages support this kind of specialization, but in the following we will show the advantages of such a possibility using a language invented for the purpose.

The strongest constraint that can be added to a formal parameter is to supply the actual parameter:

```
Descriptor Stack( Descriptor Elementtype; Size : Integer )
```

```
Descriptor Int-Stack-100  
Is-specialization-of Stack( Elementtype = Integer;  
                             Size = 100 )
```

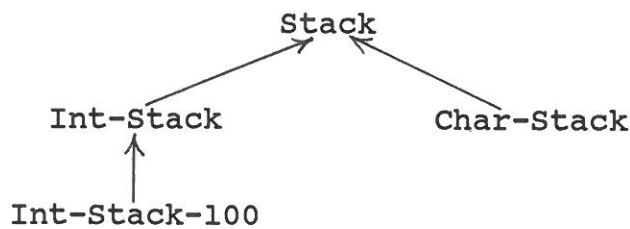
This possibility exists in Ada by the generic facility. In an Ada generic, all generic parameters are bound simultaneously resulting in a two-level specialization hierarchy. If selective binding was allowed, several levels could be obtained:

```
Descriptor Int-Stack( Size : Integer )  
Is-specialization-of Stack( Elementtype = Integer;  
                             Size : Integer )
```

```
Descriptor Char-Stack( Size : Integer )  
Is-specialization-of Stack( Elementtype = Char;  
                             Size : Integer )
```

```
Descriptor Int-Stack-100  
Is-specialization-of Int-Stack( Size = 100 )
```

This gives the following hierarchy:

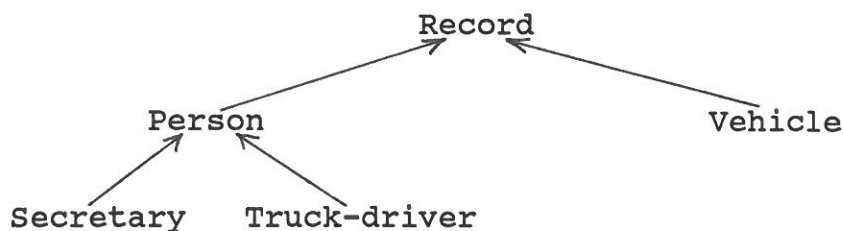


The concept of curried functions /Stoy/ is an example of this specialization technique used for functions.

If the programming language already includes a specialization mechanism for descriptors, the formals of a parameterized descriptor can be specified by means of qualification as described in section 4.3.1. A specialization of the parameterized descriptor can then be obtained by adding a constraint that specializes the qualification of a formal instead of binding it completely to an actual parameter.

Note that qualification can be used also to specify formal parameters of kind descriptor and thus solve the operation problem described in section 3.6.3: Qualifying a formal descriptor by a descriptor *Q* means that the actual parameter must be either *Q* or a more specialized descriptor. Thus the parameterized descriptor can assume that its formal parameters has all the properties of *Q* and manipulate it accordingly.

Now we will illustrate the specialization mechanism that can be supported by parameterization. Assume that we have the following hierarchy of descriptors where "Record" just describe the property of having a key-value.



and a parameterized descriptor

Chapter 4

Descriptor Queue(Descriptor Element : Record)

Then the following specializations can be made:

Descriptor Person-Queue(Descriptor Element)
Is-specialization-of Queue(Descriptor Element : Person)

Descriptor Vehicle-Queue(Descriptor Element : Vehicle)
Is-specialization-of Queue(Descriptor Element : Vehicle)

If there are several parameters, they can be used to obtain several levels in the specialization hierarchy by focusing on one parameter at a time. Moreover, the same parameter can be gradually specialized down the hierarchy:

Descriptor Secretary-Queue(Descriptor Element : Secretary)
Is-specialization-of
 Person-Queue(Descriptor Element : Secretary)

The virtual pattern mechanism in Beta is an example of a parameterization mechanism that supports this kind of specialization.

4.3.7.2 The Prefix Mechanism in Simula

A class in Simula describes aggregation components in the form of data attributes, procedures and an action part. Constraints are expressed as qualifications on data attributes and association of procedure bodies to procedure names.

In a specialized class - a subclass - new aggregation components can be added - i.e. new data attributes, procedures and action. The specialization thus obtained on the process part of the class is specialization with respect to the procedural aspect of the action, not the functional.

In subclasses, new qualification constraints are added to new data attributes and new body constraints are added to new

procedures. New qualification constraints cannot be added to inherited data attributes, but new body constraints can be given for a special kind of inherited procedures - the virtual procedures. In fact the new body constraint on a virtual procedure need not be stronger than the old one but just overwrites the old one. This means that in reality the body constraint on a virtual procedure is not a defining property but just a characteristic property of the class, whereas of course the property of having a procedure with the given name is a defining property.

For ordinary procedures - i.e. non-virtual procedures - both the existence of the name and the associated body constraint are defining properties, and further constraints cannot be added later on. This also means that for ordinary procedures both the procedural and the functional aspects are defining properties, whereas for virtual procedures none of the aspects are defining - only the name. Of course, this does not prevent the programmer from writing virtual procedures where the functional aspect is in fact invariant down through the specialization hierarchy - it is just not guaranteed by the language construct.

As a summary, the defining properties of a class in Simula - i.e. the properties that can be assumed valid when reasoning about all instances of the class independently of whether they are also instances of subclasses, are:

- The possession of certain named data attributes with some qualification constraints associated with them.
- The possession of certain named procedures with body constraints that define both the procedural and the functional aspects of the procedures.
- The possession of certain other named procedures (the virtual procedures for which only the names are defining properties).
- The possession of an action part with a certain procedural behaviour.

Chapter 4

4.3.7.3 The Subclass Mechanism in Smalltalk

A class in Smalltalk describes data attributes and procedures only, and no constraints are associated with the data attributes, whereas a body constraint is associated with procedure names.

In subclasses, new data attributes and new procedures can be added, and old body constraints can be replaced with new ones like for virtual procedures in Simula.

The only defining properties of a class are thus the possession of some specific named data attributes and procedures. No constraints are defining properties at all.

4.3.7.4 The Prefix Mechanism in Beta

Instead of an exhaustive description of the prefix mechanism in Beta, we will point out the most important differences from Simula and Smalltalk:

A pattern (a descriptor similar to a class or a procedure) describes local entities (for instance data attributes), local patterns and an action.

Local entities are qualified like in Simula, and the qualification is a defining property, but as opposed to Simula, additional qualification constraints can be added in a specialized pattern (a subpattern). That is, the qualification constraints on existing data attributes can be strengthened but never cancelled in subpatterns. Similarly, the body constraints on procedures can be strengthened to more specialized bodies but never cancelled. Thus also the body constraints are defining properties, but as in Simula, specialization of process considers only the procedural aspect, not the functional.

In summary, Beta has obtained that all aggregation components and all constraints are defining properties, and at the same time allows new constraints to be added also to inherited aggregation components in specialized descriptors. This is obtained by means of the virtual pattern mechanism in Beta.

4.3.7.5 Subtypes and Generics in Ada

In Ada there is only very little support of specialization. However, we will mention subtypes and generics and show why these concepts only weakly support specialization.

In section 4.3 and section 4.3.1 we discussed specialization in Ada but let us here give a short summary:

In section 4.3 an example of a record-subtype was given, but only a two-level specialization hierarchy was obtained. Array-subtypes can also result in two-level specialization hierarchies only. Only for enumeration types, hierarchies of greater depth can be obtained.

In section 4.3.1 some of the limitations of the Ada generic concept were illustrated, namely the fact that although a generic instantiation can be considered a specialization of the generic unit, it is impossible to use qualification - for instance to write a procedure that will accept any generic instantiation of a particular generic unit as an actual parameter. Moreover, generics only support two-level specialization hierarchies.

In /Wegner/, a more detailed criticism of the Ada generic concept is given.

CHAPTER 5

CONCLUSION

We have shown that modelling and abstraction are basic activities in a programming process. Modelling causes some serious problems due to the conflict between the fuzzyness of many concepts in the problem area and the Aristotelian nature of concepts in traditional programming languages. We have made no attempts to solve these problems but have concentrated on how to judge and improve the quality of programming languages with respect to conceptual modelling and abstraction within an Aristotelian view of concepts.

As a basis for the discussion, we have introduced a model of program executions that makes it easy to relate program executions to programming languages and conceptual modelling. This model makes a basic distinction between descriptors and entities - a distinction that is usable for all programming languages.

We have shown that traditional topics like data, values, processes, names and parameterization can be adequately discussed within this model.

A thorough analysis of abstraction has been given. First as general mechanisms for concept structuring, and later as mechanisms in programming languages. The discussion of abstraction in programming languages is based on our descriptor-entity model and showed the importance of supporting all of the functions: Classification/exemplification, aggregation/decomposition and generalization/specialization.

Additional expressive power in the programming language is obtained when all the abstraction functions are supported. Moreover, by supporting all the functions, the conceptual distan-

ce between the problem area and the programming language can be more easily bridged, and the concept structure in the resulting program will in a nice way mirror the concept structure in the problem area, thus making the program more comprehensible.

It is our hope that our framework can serve as an aid in gaining a coherent understanding of the programming language area, and be a source of inspiration in the design of new programming languages. In particular, we hope that we have succeeded in arguing that specialization deserves to be better supported in programming languages than it is at present.

Acknowledgements

We want to express our gratitude to our supervisor Ole Lehrmann Madsen for many inspiring discussions concerning our conceptual framework. Thanks are also due to Peter Mosses who acted as our supervisor during Ole Lehrmann Madsen's sabbatical, and to Brian Mayoh who made us clarify our analysis of specialization. Finally, we have benefited from comments from graduate students who read an earlier version of this paper on a graduate course.

CHAPTER 6

REFERENCES

- /Ada/ Reference Manual for the Ada Programming Language, United States Department of Defence, July 1982.
- /Algol60/ P. Naur: "Revised Report on the Algorithmic Language Algol60", Communications of the ACM, Vol.6, No.1, January 1963.
- /Algol68/ A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisher (Eds.): "Revised Report on the Algorithmic Language Algol68", Acta Informatica, Vol.5, No. 1-3, 1975.
- /Alphard/ W.A. Wulf, R.L. London, M. Shaw: "An Introduction to the Construction and Verification of Alphard Programs", IEEE Transactions on Software Engineering, Vol.2, No.4, December 1976.
- /APL/ K. Iverson: "A Programming Language", John Wiley & Sons, Inc., London, 1962.
- /Astrahan et al./ M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, J.L. Traiger, B.W. Wade, V. Watson: "System R: A Relational Approach to Data Base Management", ACM Transactions on Database Systems, Vol.1, No.2, June 1976.

- /Barr et al./ A. Barr, A. Feigenbaum (Eds.): "The Handbook of Artificial Intelligence", Vol.1, Pitman Books Limited, London, 1982.
- /Beta/ B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: "Abstraction Mechanisms in the Beta Programming Language", Proceedings of the 10'th ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983.
- /Borgida/ A. Borgida: "On the Definition of Specialization Hierarchies for Procedures", 7'th International Joint Conference of Artificial Intelligence, Vancouver, Canada, August 1981.
- /Clu/ B. Liskov, A. Snyder, R. Atkinson, G. Schaffert: "Abstraction Mechanisms in CLU", Communications of the ACM, Vol.20, No.8, August 1977.
- /Concurrent Pascal/ P. Brinch Hansen: "The Architecture of Concurrent Programs", Chap.8, Prentice-Hall, 1977.
- /CSP/ C.A.R. Hoare: "Communicating Sequential Processes", Communications of the ACM, Vol.21, No.8, August 1978.
- /Edison/ P. Brinch Hansen: "Edison - A Multiprocessor Language", Software-Practice & Experience, Vol.11, No.4, April 1981.
- /Fortran/ American National Standard Fortran (ANS X3.9-1966), American National Standards Institute, New York.
- /Fuzzy/ R. LeFaivre: "Fuzzy Reference Manual" Computer Science Department, Rutgers University, March 1977, Revised June 1978.
- /Gehani/ N. Gehani: "Generic Procedures, An Implementation and an Undecidability Result", Computer Languages, Vol.5, 1980.
- /Hanssen et al./ E. Holbaek-Hanssen, P. Haandlykken, K. Nygaard: "System Description and the

Chapter 6

- DELTA Language", publication no. 523, Norwegian Computing Center, February 1977.
- /Hoare/ C.A.R. Hoare: "Notes on Communicating Sequential Processes", Technical Monograph PRG-33, Oxford University Computing Laboratory, August 1983.
- /Icon/ R.E. Griswold, M.T. Griswold: "The Icon Programming Language", Prentice-Hall, 1983.
- /Jensen et al./ K. Jensen, N. Wirth: "Pascal User Manual and Report", second edition, Springer-Verlag, 1975.
- /Jørgensen et al./ T.M. Jørgensen, J. Kammersgaard: "Et begrebsapparat til karakteristik af programmeringsprocesser" (In Danish), DAIMI IR-38, Computer Science Department, Aarhus University, August 1982.
- /Knudsen a/ J.L. Knudsen: "Exception Handling - A Static Approach", Software-Practice & Experience, Vol.14, No.5, May 1984.
- /Knudsen b/ J.L. Knudsen: "Control Abstraction in Programming Languages", in preparation.
- /Larsen/ S.F. Larsen: "Egocentrisk tale, begrebs-strukturer og semantisk udvikling" (In Danish), Nordisk Psykologi, Vol.32, No.1, January 1980.
- /Lisp/ J. McCarty et al.: "Lisp 1.5 Programmer's Manual", 2nd. edition, M.I.T. Press, Cambridge, Mass.
- /MacLennan/ B.J. MacLennan: "Values and Objects in Programming Languages", ACM Sigplan Notices, Vol.17, No.12, December 1982.
- /Madsen et al./ O.L. Madsen, B. Møller-Pedersen, K. Nygaard: "From Simula-67 to Beta", Proceedings of the Eleventh Simula-67 User's Conference, Paris, Norwegian Computing Centre, 1983.
- /Mathiassen/ L. Mathiassen: "Systemudvikling og system-udviklingsmetode" (In Danish), DAIMI PB-136, Computer Science Department, Aarhus University, September 1981.

- /Milner/ R. Milner: "A Calculus for Communicating Systems", Lecture Notes in Computer Science, Vol.92, Springer-Verlag, 1980.
- /ML/ M. Gordon, R. Milner, C. Wadsworth: "Edinburgh LCF", Lecture Notes on Computer Science, Vol.78, Springer-Verlag, 1979.
- /Modula-2/ N. Wirth: "Modula-2", Berichte des Instituts fuer Informatik, Nr.36, ETH, 1980.
- /Nielsen et al./ M. Nielsen, G. Plotkin, G. Winskel: "Petri nets, event structures and domains", Proceedings of Conference on Semantics of Concurrent Computation, Evian, Lecture Notes in Computer Science, Vol.70, Springer-Verlag, 1979.
- /Peterson/ J.L. Peterson: "Petri Nets", Computing Surveys, Vol.9, No.3, September 1977.
- /Pascal/ BS6192: 1982 Specification for Computer Programming Language Pascal, ISO 7185. In I.R. Wilson, A.M. Addyman: "A Practical Introduction to Pascal - with BS 6192", Macmillan, 1982 (second edition).
- /Simula/ O.-J. Dahl, B. Myhrhaug, K. Nygaard: "Simula 67, Common Base Language", Norwegian Computing Center, 1970.
- /Smalltalk/ A. Goldberg, D. Robson: "Smalltalk-80: The Language and its Implementation", Addison-Wesley Publishing Company, 1983.
- /Smith et al./ J.M. Smith, D.C.P. Smith: "Database Abstractions: Aggregation and Generalization", ACM Transactions on Database Systems, Vol.2, No.2, June 1977.
- /Stoy/ J.E. Stoy: "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", MIT Press, 1977.
- /Taxis/ J. Mylopoulos, M.K.T. Wong: "Some Features of the Taxis Data Model", 6'th International Conference on Very Large Data Bases, October 1980.

Chapter 6

- /Tennent/ R.D. Tennent: "Principles of Programming Languages", Prentice-Hall International, London, 1981.
- /Thinglab/ A. Borning: "Thinglab -- A Constraint-Oriented Simulation Laboratory", SSL-79-3, Xerox PARC, July 1979.
- /Thomsen a/ K.S. Thomsen: "Multiple Inheritance in Object Oriented Languages", preliminary version, Computer Science Department, Aarhus University, June 1984.
- /Thomsen b/ K.S. Thomsen: "Specialization Hierarchies and Distributed Termination Detection", in preparation.
- /Thomsen et al./ K.S. Thomsen, J.L. Knudsen: "A Taxonomy for Programming Languages with Multi-Sequential Processes", DAIMI PB-175, Computer Science Department, Aarhus University, May 1984.
- /Vaucher/ J.G. Vaucher: "Prefixed Procedures: A Structuring Concept for Operations", INFOR, Vol.13, No.3, October 1975.
- /Wegner/ P. Wegner: "On the Unification of Data and Program Abstraction in Ada", Tenth International Conference on Principles of Programming Languages, January 1983.
- /Wong/ M.K.T. Wong: "Design and Verification of Interactive Information Systems Using TAXIS", TR CSRG-129, University of Toronto, April 1981.
- /Wulf et al./ W.A. Wulf, M. Shaw, P.N. Hilfinger, L. Flon: "Fundamental Structures of Computer Science", Addison-Wesley Publishing Company, Philippines, 1981.