Notes on the lecture

Fundamentals of Software Engineering I

Prof. Dr.-Ing. Axel Hunger

UNIVERSITÄT
D U I S B U R G
E S S E N

WARNUNG
preliminary lecture notes

Dieser Text beschreibt in komprimierter Form die Inhalte der Vorlesung „Grundlagen der Programmentwurfstechnik 1“ wie sie regelmäßig an der Universität Duisburg-Essen für Studierende in den Bachelor Studiengängen der Abteilung Elektrotechnik- und Informationstechnik gelesen wird.

Die überwiegend ausländischen Studierenden haben immer wieder den Wunsch nach englischsprachigen Vorlesungsunterlagen geäußert. Aus diesem Grund wird seit 1999 der vorliegende Text den Studierenden zur Verfügung gestellt. Es wird dringend zur kritischen Verwendung des Textes geraten, da die abschließende Korrektur noch aussteht und einige Bilder noch nicht ins Englische übersetzt sind. Außerdem sind inhaltliche Anpassungen des Stoffes in den letzten Jahres erfolgt, die sich noch nicht im Text wiederfinden. Daher soll der Text in erster Linie vorlesungsbegleitend eingesetzt werden und die eigenen Vorlesungsmitschriften ergänzen, diese aber auf keinen Fall ersetzen.

Duisburg, im April 2009

Table of Contents

1 The Life Cycle of Software

Cost Development of Computer Systems

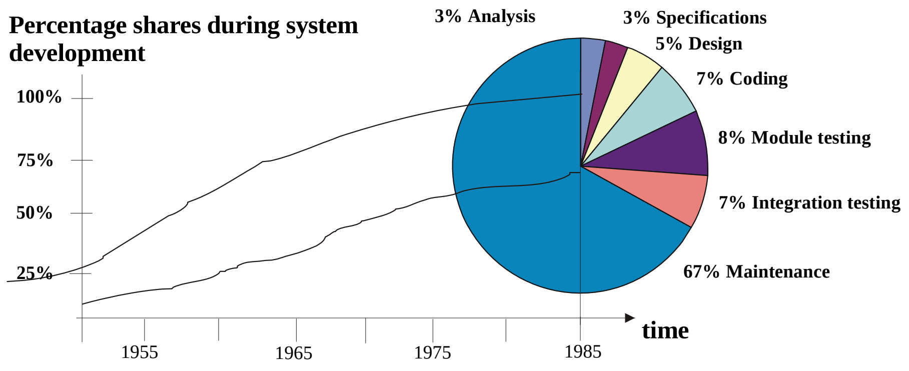


Fig. 1.1: Proportion of Development Costs in Different Project Phases

The most important statements of this analysis can be summarised as follows:

Cost development	Hardware	Software
1955	80%	20%
1985	20%	80%!

Especially for software it can be said that

- one third of the costs are development and
- two thirds maintenance !

Development costs consist of

3% analysis
3% specification
5% design
7% coding!
8% module testing
7% integration testing
67% maintenance

It can be seen that not only the preparatory phases (from analysis to design) but also the testing phases each amount to double the costs of coding. If on the other

hand you add up the expenses incurred after coding, , they amount to **82%** of the total costs.

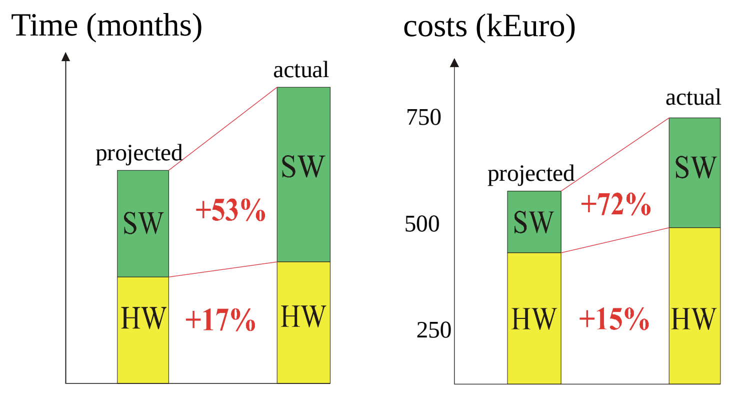


Fig. 1.2: Relation between planned and actual time required and between planned and actual costs

When analysing differences in cost development, it has to be taken into account that software and hardware differ greatly in their development processes. In addition, different frameworks influence the quality of these processes:

Hardware:	Software:
industrial development	*creative development*
(construction) parts	personnel
manufacture	design methods
quality control	compiler

It has to be taken into account that the industrial manufacturing process has a tradition of approximately one hundred years and have thus benefited from continuous improvements and automation. Software development on the other hand is still in its infancy and it is still difficult to prove economic advantages of certain procedures. Above all, until today it is often felt that the personnel's creative potential and their ability to come up with special "tricks", i.e. to achieve outstanding results by using obscure methods and languages, are critical for the quality of the product. This mistaken belief is in fact counter-productive.

Hopes:	Reality:
SW factories	SW freaks
Program generators	CASE-tools, but programming languages are - not standardised - complex systems are gaining ground (RISC for HW, for human beings ???) Programming styles are - (structured, functional, object-oriented) - extremely dependent on trends - claiming to have *the* solution to every-thing
portable SW	
the "thinking machine“	

To gain a systematic view of software engineering, let's take an idealised model of the software life cycle (life cycle model, cf fig. 1.3) as a starting point.

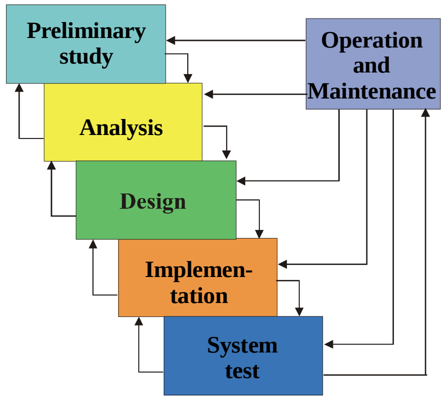


Fig 1.3: Model of the Software Life Cycle (Waterfall Model NEM 90)
The separate phases can be described as follows [NEM 90]:

Preliminary study:
Project suggestions (possibly with outlines, including left-out alternatives) are part of the preliminary studies. Objectives and requirements have to be clearly defined. Additionally alternatives have to be considered from a technical, as well as from an economic point of view. Finally, one project is selected.
Requirement Analysis:

Technical specifications are laid down. It is absolutely vital that the end user takes an active role in this process so that his or her needs can be taken into account.

Design:
The system, i.e. hardware and subsystems, i.e. the software, have to be designed in detail.

Implementation:
Implementation is carried out top-down or bottom-up.

Top-down:
• Downward levelling, starting from the highest level of abstraction
• Interfaces are tested at a lower level
• Only after this can implementation of the individual components begin. (This is psychologically positive, but can get very complex with larger systems.)
• Important representatives of this method: structured analysis after DE MARCO and design after WARD/MELLOR, especially useful for real-time systems.

Or

Bottom-up:
• Component implementation and testing occurs at the lowest level
• This is followed by incremental construction (simple procedure, "estranged", design errors may only be discovered later in the development process)
• Object-oriented programming: The goal is to create reusable code: Small classes are defined which have a certain functionality. New classes inherit the properties of the smaller classes and extend them. Classes are thus developed successively based on the principle of heredity. Routines which work with the smaller classes will also work with the extended ones.

System testing:
Usually this is an incremental test carried out in a real environment.

Operation and maintenance:
Operation and maintenance consist of updating on the one hand, and repairs,on the other. Updating includes the adjusting to new environments or an extension of functions respectively, while repairs are limited to the rectifying of errors.

1.1 Common mistakes made during Software Development

1. Coding is started immediately

Reasons:

• Deadlines:
This reasoning is hardly valid, since resulting errors have to be corrected later on anyway and thus lead to an increase in the work effort. This approach is only justified when there is a necessity to produce a partial solution quickly.

Consequences:
• Software requirements have not been properly analysed and established. The software being developed will most likely not meet the demands of the end-user.
• There is next to no structure and modulisation present in the software. To modify the software later on according to demands, extra program parts will have to be added which will influence the structure negatively. In addition, the adding of program parts takes up more time and effort than an appropriate modular design would have done in the first place.
• Without properly establishing the requirements a goal-oriented examination of the software is unfeasible.

Errors can be avoided if:

• Consequences in terms of time and cost for follow-up work are pointed out right away of software that has been coded.
• Procedural models (Vorgehensmodell) are binding and clearly determined.
• Unrealistic deadlines are avoided.
• Software that has been coded right away, without going through the phases of analysis, system design and program design, is only used as an "island solution" for specific needs. It is not integrated into the overall system, but will be replaced by "normally" developed software later on.

2. Testing is unsystematic and/or insufficient.

Reasons:
• Deadlines
• The importance of proper testing is not realised.
• The testing schedule has not been developed during the beginning phases of development. Thus, the time required for testing is very high.

Consequences:

• Tests that have not been carried out systematically cannot be reproduced, i.e. it is impossible to do regression testing.
• Improvements in testing are impossible since errors occurring later on cannot be correlated to certain (insufficient) testing phases.
• Errors which have not been discovered during testing will appear in later phases of the process or are only discovered by the end-user (customer).
  - Depending on the type of software these errors can lead to serious damage or even to serious or fatal injuries (with all its consequences) and/or
  - cause error removal costs that will be higher than the costs of systematic testing would have immediately after software creation.

Errors can be avoided if:
• Possible consequences of insufficient testing are pointed out.
• Check list for the development process is binding and clearly determined.
• Time is reserved for testing.
• Unrealistic deadlines are avoided.

3. Requirements and quality features have not been determined

Reasons:
• User/customer does not want to determine anything (yet)
• In the first phases of development the software to be developed is also used to find out what the final system can achieve. So demands and limits of the system are determined by using only the provisional software system.

Consequences:
• The software which is being developed does not meet the customer's expectations (which have not been fixed in written form).
• The software has to be re-written requiring extra effort.
• Criteria for examinations and testing are missing, so testing cannot be carried out effectively.

Errors can be avoided if:
• Temporary development goals are determined.

4. Standards and regulations are ignored

Reasons:
• There are too many standards and regulations. Useful standards are "hidden" by the large amount of superfluous regulations.
• Standards are not practical and thus, not accepted.
• The benefit of standards and their advantages are not understood by the developers.

Consequences:
• Re-use of software is complicated.
• Testing by other employees is complicated.
• Mechanisms and measures for error avoidance which are normally covered by standards and regulations cannot take effect.

Errors can be avoided if:
• Developers are informed about the purposes and aims of standards and regulations.
• Standards are critically evaluated.

5. Missing, out-dated, insufficient or inadequate Documentation.

Reasons:
• Tiresome, dull routine
• Deadlines
• There is no appreciation of the difficulties.

Consequences:
• If development decisions of earlier phases have not been documented properly (including the reasons that led to the decisions), they have to be re-developed. There is also a danger of new decisions being wrong since the reasons for making the former decisions have not been documented and are thus ignored.
• Modifications and maintenance work are extremely difficult; in extreme cases it can be better to develop new software.

Errors can be avoided if:
• Documentation of development processes can be largely automated by using the appropriate tools.
• Regular documentation check-ups are carried out.
• Time is reserved for documentation procedures.
• Realistic time schedules are set up.
• Developers are informed about the negative consequences of missing and/or insufficient documentation.

6. A procedural model is missing or is kept to

Reasons:
• The supervisors in charge do not acknowledge the necessity for developing a procedural model.
• The procedural model is ignored as it is not suitable for the particular development process at hand.

Consequences:
• Sequence, range (of examinations, testing and documentation) and realisation (methods) of development vary from project to project and are largely dependent on the particular supervisors in charge.
• Co-operation with team colleagues from other projects becomes difficult (e.g. the realisation of testing in the form of reviews).

Errors can be avoided if:
• The positive effect is pointed out that a well thought-out written and binding procedural model can have on the quality of a software product.
• A suitable procedural model is developed.

7. Phase results are not inspected

Reasons:
• A suitable procedural model is unavailable.
• Distinct phase ends are not recognisable due to frequent iterations in the course of development.

Consequences:
• Errors are only discovered in later phases or even by customers (see No. 2).
• Project progress cannot be assessed properly as statements on the quality of a product based on testing are unavailable.

Errors can be avoided if:
• Deadlines and mandatory requirements for the inspection of phase results in the procedural model have been clearly defined.

8. Poor name assignment, e.g. file names, class names, method names and variable names

Reasons:
• There are no useful and binding rules for the assignment of names.
• The programming language employed restricts the length of identifiers.

Consequences:
• Names that seem informative and expressive to the inventor have no immediately recognisable informational value for team colleagues.

Errors can be avoided if:

• For the adequate characterisation of types, constants, variables, procedures, etc. choose names that express the function or task of the identifier in question. Avoid names that are irrespective of the problem at hand or describe technical aspects, e.g. of representation.
• In the case of programming languages restricting the length of identifiers, add a comment which contains the full name of the identifier.
• Standards and guidelines for name assignment are defined.

9. The system architecture cannot be extended or extension is complicated (no information hiding, no modularity)

Reasons:
• No plan of the later software architecture has been established in the first phases of software development.
• A plan of the later software architecture has been established, but it has not been critically reviewed.

Consequences:
• System architecture becomes more and more confusing with every change and amendment.
• Since the impact of changes cannot be limited to specific modules, the software is susceptible to errors.
• Time and effort are wasted on error search and testing.

Errors can be avoided if:
• a plan of the system architecture is created that is binding and clearly determined and it is to made sure that this plan is reviewed.

10. The training of software developers and software users is neglected or is not considered necessary

Reasons:
- It is wrongly assumed that working with a software tool properly and/or applying a developing method correctly can be learned on the job.
- Supposedly there is no time for training.
- Trying to cut costs by omitting supposedly unnecessary expenditures, such as training.

Consequences:

• Methods to be used have not been fully understood and are thus wrongly applied. This also leads to an enormous waste of time.
• Software tools and application programs are used correctly, but a lot of time is wasted since the users do not know the program well enough and also will not have time to get to know the program better in the future due to heavy work load.
• The time that has been initially saved by omitting training for software tools and development methods will later have to be made up for, since tools and methods are not applied properly.
• The time that has been initially saved by omitting training for the end-users has be reinvested several times because the end-users will contact the developers individually and ask them the same questions over and over again.

Errors can be avoided if:
• Financial loss and counter-productivity of omitting training are high-lighted.
• Time is reserved for training.

11. Unrealistic Deadlines

Reasons:
• Developers have no experience with the time needed for certain developmental tasks, since previous time allotments have not been documented in other projects.
• Deadlines that have been arranged with customers have not been confirmed with developers.
• The time needed for check-ups and testing as well as for documentation has been disregarded in scheduling.

Consequences:

• The "development" of software is reduced to the development of code without considering the phases *analysis* and *design* sufficiently. Tests are only conducted as long as time is abundant. There is no time for appropriate documentation.

Errors can be avoided if:
• The time needed for development tasks (incl. all necessary procedures) is documented and evaluated.

12. Terms have not been defined

Reasons:
• Everybody knows what is intended when using a certain term. It is assumed that team colleagues share the same definitions.

Consequences:
• Discussions about certain issues are rendered useless since the participants in the discussion have different views about the issue at hand.

Errors can be avoided if:
• A dictionary of terms is established during development which lists all technical terms which may not commonly known or used with varying meanings.

13. The selection of tools and methods has not been properly prepared

Reasons:

• Prior to the application of new tools and methods, an analysis of the developmental process has not been carried out, i.e. actual demand has not been determined.

Consequences:
• The tools and methods employed might not be the ones most suitable for the task at hand (loss of productivity).
• Developers have to deal with inappropriate tools and methods. They will try to avoid them or not to use them at all (loss of motivation and productivity).

Errors can be avoided if:
• The actual situation is analysed.
• A catalogue of criteria containing general as well as company-specific standards is set up.

1.2 The Impact of Errors on the Production Process

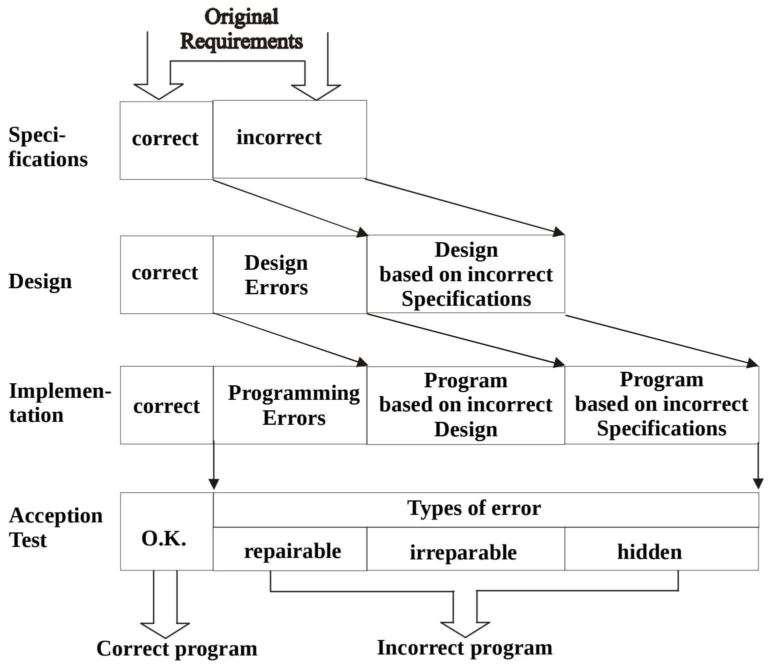


Fig. 1.4: Origin of different error types

A first attempt at error diagnosis (tracing back the origin of the error) can be derived from figure 1.4. These considerations justify a structured procedure in program development. Another reason lies in the law of product liability which forces all manufacturers to continually document the production process to avoid liability claims involving their product.

According to DE MARCO [DeMA 78] reasons for software errors and the resulting costs can be described as follows:

Software error	**Origin in**	**Resulting costs**
Requirements	56%	82%
Design	27%	13%
Implementation	7%	1%

1.2 Commercial Aspects

The following influences on contract arrangements have to be considered:
1. Legal department: ownership laws, right of use, copyright, etc.
2. Sales department: prices, deadlines, commodity sales, guarantee, etc.
3. Engineers: technical requirements, estimate of expenditure, etc.

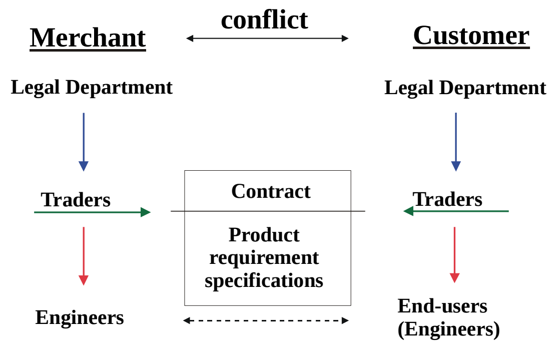


Fig. 1.5: Commercial Aspects

The contract results from supply and order on the basis of product requirement specifications. However, an iterative communication process between supplier and customer often leads to continuous changes of the original product specification requirements. It is crucial that consensus is eventually reached on common goals.

Product requirement specifications should include:

• introduction
• order
• problem analysis
• feasibility (cost/effect)
• scope of product requirement specifications
• project organisation
• phase schedule
• flow control
• agreements

A problem in particular is that the supplier usually wants to invest as little as possible prior to the completion of the order (usually a design is not carried out). This, however, extremely complicates the allotment of labour and time as well as specification.

General Model Construction

2.1 Functional Models

The first requirement-analysis methods were developed in the 1950s and were later extended to serve as the basis for program design. This additional application followed from the fact that the result of any formal analysis can also be expressed as a program-definition algorithm. This is especially true for the basic formulations detailed below.
Decision rules can provide the basis for formal algorithms. Fig. 2.1 illustrates a way of representing decisions in table form.

Table name	**e) Rules**						
	1	**2**	**3**	**..**	**..**		**n**
a) Conditional part (Conditions)	**c) Conditions display part (possible states)**						
b) Action part (Actions)	**d) Action display part (Assignment to conditions)**						

Fig. 2.1: Basic Form of a Decision Table (cf [NEM 90])

A decision table consists of five elements which perform the following functions:

a) **Condition part:** defines all conditions relevant to the respective problem.
b) **Action part:** lists all possible actions.
c) **Condition display:** assigns individual conditions to the rules.
d) **Action display:** assigns specific actions to the rules.
e) **Rules:** every column defines the individual conditions needed to comply with a rule (condition display) and which action is to be taken should the rule be observed (action display).

The conditions of a rule can be:
• simple "yes/no" decision,
• complex "yes/no/--" decision (--: irrelevant or don't care).

IF/THEN/ELSE structures can then be directly derived.
Every complex rule can also be expressed by two simple rules. (the use of "--" symbols therefore only serves to densify the table).

Extended (or customised) decision tables can be obtained by ensuring that rule or action definitions are drawn into the condition and action displays wherever possible. Such tables are more compact and suitable for human use, but are less appropriate for program use. Alternatively, a CASE structure can also be used to create tables (see Fig. y). In addition, these tables allow the formal verification or identification of
• completeness,
• redundancy or
• contradictions, etc..

```
case <expression> is
  when <choice 1> =>  <statements 1>
  when <choice 2> =>  <statements 2>
...
when others >= <statements>
```

Fig. 2.2: General Syntax of a CASE Statement [DU 93]

Decision trees contain identical information and are therefore no more than a graphic representation of the decisions and their specific sequence, as are tables in different form.

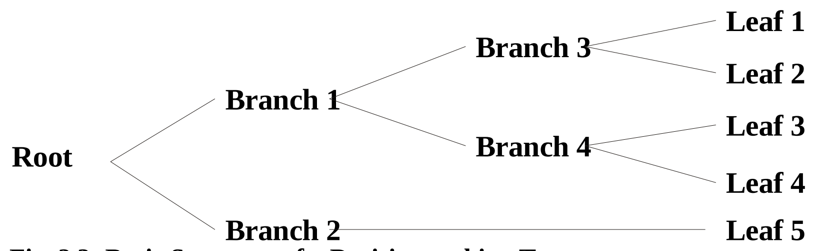


Fig. 2.3: Basic Structure of a Decision making Tree

2.1.1 Corresponding Machine Model

A corresponding formalisation can be undertaken in the form of a machine, for which the following is defined:

- the number of states Z,
- the number of inputs X,
- the state-transition function F,
- an initial state and
- a number of final states.

The interpretation of HW machine definitions deviates from those of program design.

Hardware Machine	Program Design
X: Input Signal and Clock	Event or Message
Y: Output Signal	Action upon State transition: Action, procedure, etc.
Z: (Coded) State	System state

Fig. 2.4: Use of Machine Models for Hardware and Software Design

(cf also fig. 3.14)

Finite machines can be divided into two groups:
- combinational machines
- sequential machines

They only differ in their state transition functions.

Combinational Machines
With combinational machines the new state is independent of the current state during input.

For example, the locking mechanism of a combination lock is basically, like the name implies, a combinational function.
Output Y = f(x) (≠ f(z)!)
Such problems can be defined by using the above-mentioned decision tables.

Example 2.2

Sequential Machines
Contrary to combinational machines the new state is dependent on the current state. i.e. on prior input.

In practice most systems can be described by sequential machines. They can be represented by:

- state diagrams,
- tables and
- matrices.

2.1.1.1 State Diagrams

As shown in fig. 2.5 there are various symbols for describing sequential behaviour in machine form. These definitions provide the basis for various design methods mentioned later. They are especially suitable for defining user interfaces.

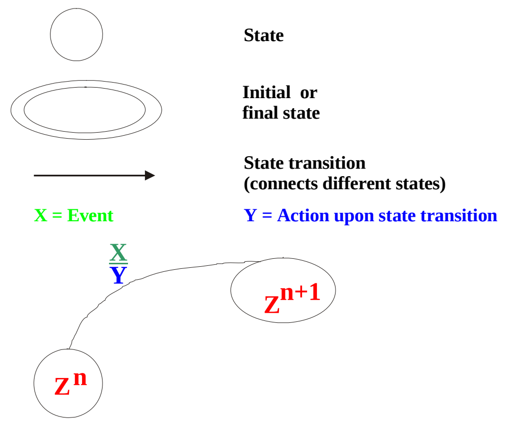
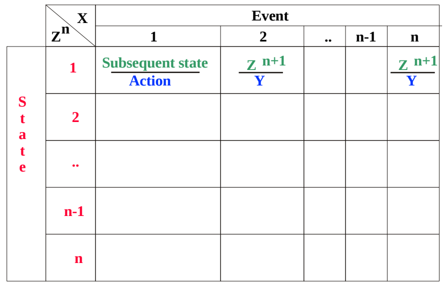

State
Initial or final state
State transition (connects different states)
X = Event
Y = Action upon state transition
X
Y
Z^n+1
Z^n

Fig. 2.5: Symbols for Describing Sequential Behaviour in Machines

2.1.1.2 State Matrix

The state matrix is also called machine table. Lines correspond with states and columns with events. As can be seen in fig. 2.6 we distinguish between Mealy- and Moore-Type matrices. With the Mealy machine a next state and an action are named for all matrix elements and all actions are executed dependent on the respective state transitions.
In contrast to that with the Moore machine actions are merely functions of states. The same action is executed for all transitions starting from a specific state.



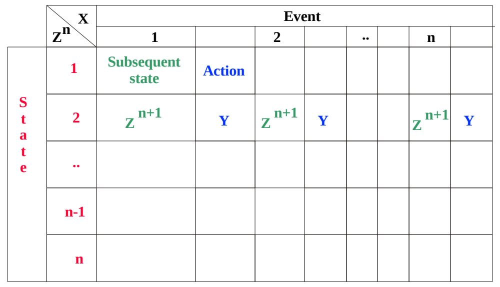


Fig 2.6: State Matrix
Mealy-Type: Action is Assigned to a State Transition
Moore-Type: Action is Assigned to a State

2.1.1.3 State Tables

State tables roughly correspond to the decision trees mentioned earlier.

System State	Event/ Message	Procedure / Process	New System State
Present State 1 □	Event□	Prompt Action	Next state □
Present State 2	Event 1	Action	Next state
	□..	□..	□..
	Event n	..	..
..	..	..	..

Fig. 2.7: Basic State Table Structure

These tables constitute a complete list of all case decisions that define the transitions:
- of each present state (first column),
- on the basis of all possible results (all sub-columns per state),
- for all defined next states (final column).

This definition matches Mealy-circuit structure, for which the following applies:

action = f (state transition).

State tables thus constitute nothing more than a compilation of all decision trees (system states) that a machine could run through (or take on) at any one time.

Example 2.3

2.2 Data Models

2.2.1 Layer Architecture

Prior to producing models of certain types of data it is necessary to view the entire 'data-modelling' process. This is the only way to initiate the modelling process and to differentiate between the individual modelling tasks.

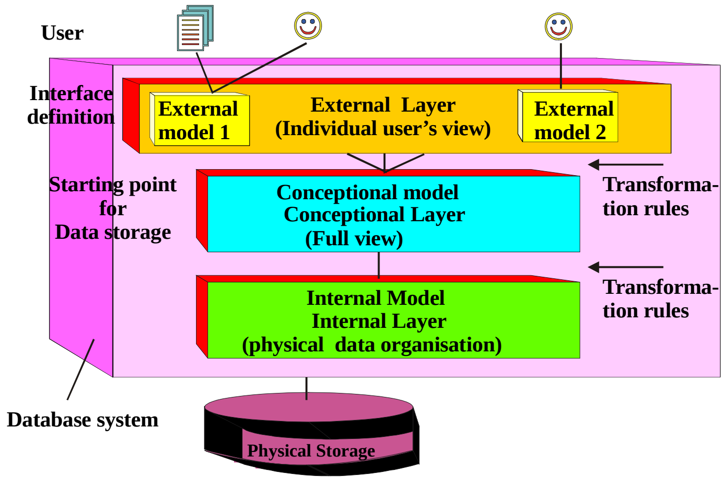


Fig. 2.8: Layered Architecture for Data Modelling (cf [NEM 90])

Explanantions for the various models can be found under figure 2.9. Let's only add that the transformation rules represent the connections between the models of the various different levels. “They determine how model objects are constructed from objects of a lower-level model. The rules are either implicit in the database system (e.g. data conversions upon the transition from internal to conceptual models) or have to be explicitly defined. A transformation rule, for example, could determine for an external/conceptual model how the data subset relevant to the external model is formed. “ [NEM 90]

This schema is mainly important because of the fact that the user has no direct control of the actual data storage. The following advantages result:

- the user is not involved in internal data organisation,
- the user remains unaffected by any possible data reorganisation,

- data integrity is assured (protecting data from the user with regard to confidentiality or destruction as well as ensuring the consistency of compound data sets).

A data model (information model) should be:

- compact,
- complete,
- simple.

There are various ways of achieving this:

hierarchical, father-son model,
network model (data allocation to several levels),
object-oriented model,
entity relationship model,
relational model,
. . .

2.2.2 Basics of Data Modelling

Information flow has constantly increased within society in recent years. The volume of data to be transmitted or stored has thus also grown. A formalised and uniform method of representation is therefore a prerequisite for storing and transmitting such data between the individual components of an information system. This is why standardised descriptions as well as the concise graphical representation of data plays such a crucial role.

This section will describe how to turn structured information into a data model. We will now explain the whole point of data-modelling definition languages. A practical example from the field of electrical engineering will demonstrate the hierarchical structure of a data model and its graphical representation.

A key feature of a database is the separation of the actual data storage from the users. Data organisation is performed by a central data administration unit. This method provides the benefits that have already been listed above:
- the user is not involved in the internal organisation of the database,
- the user remains unaffected by any possible internal reorganisation of data structures,
- data integrity is not endangered by incorrect operation.

This enables the database to be structured and at the same time avoids unstructured, multiple storage (redundancy). In addition, the flexibility of the database is guaranteed, thus ensuring that new user requirements can be integrated.

The structure of a database has to include all possible data and can be divided into three different schemas according to certain criteria. These form the conceptual framework of a database. [SCL 93].

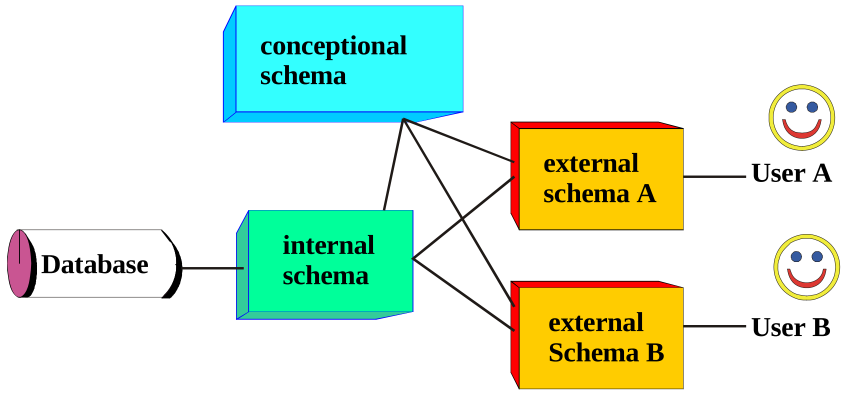


Fig. 2.9: The 3-Schema Model of Database Definition

With **conceptual** models, the data to be administered is defined from a purely logical point of view, wholly independent of any data-processing aspects. This constitutes a way of representing the real world in a data model, whereby the information to be stored is abstracted and separated so as to enable systematic representation. For this reason, various data and relationship types are defined and their attributes and values are fixed. The description of a conceptual model leads to the schema of the same name. Producing the conceptual schema is a fundamental task in creating a database. The choice of modelling method, which will be given fuller treatment in the further course of this chapter, also plays an important part.

The logical contents of the **external** model can be fully derived from the conceptual model. The external schema resulting from this model is important for every database user. As it would not be sensible to grant every user access to the entire database, this schema defines which data each respective user will be able to access.

Database queries are usually worded using a DML (data manipulation language), whose functions can be derived from the external model.

Physical data organisation is finally defined in the **internal** schema. In order to develop such an abstract organisation form, one needs to know what was defined in the conceptual schema. Furthermore, the format in which data are to be stored in the physical storage medium is also fixed.

Example 2.4

Only the conceptual and the external schema will be discussed in more detail below, as the internal schema only defines the physical data organisation in accordance with the data-format convention, which is mainly important for the technical construction of the mass storage unit.

A data model serves the uniform definition of all data contained within a closed system. Such a model should fulfil the following three criteria:

- **Compact Data Storage**
  Data storage should be as efficient as possible since many applications are very data intensive. This means that mechanisms need to be implemented to try and save space, for instance, including reference mechanisms with which to avoid multiple definitions of identical information (redundancy).

- **Completeness**
  The model should be able to represent all information relevant to the system. Although this constitutes a practical impossibility, the volume of undefined information should be kept to a minimum.

- **Simplicity**
  Data structure should be kept simple. Most users will only be working with a subset of the entire data volume. Choosing a problem-orientated structure is therefore particularly important. A hierarchical structure is usually most suitable.

A number of model approaches were developed in recent years with the aim of putting the basic considerations of conceptual schemas into practice, the most important of which are

- the hierarchical model,
- the network model,
- the relational model and
- the ER (Entity Relationship) model

The terms 1, c or m association enable the relationship between two sets of data, A and B, to be represented within these models.

Association type (A,B)		Data type from B which is assigned to every data type from A
1:	**(simple association)**	**exactly one**
c:	**(conditional association)**	**none or one (c=0|1)**
m:	**(multiple association)**	**one or more (m>=1)**
mc:	**(multiple-conditional association)**	**none, one or more (mc>=0)**

Fig. 2.10: The Four Types of Association

In this case, an association determines how many data types from A can be assigned to a data type from B [ZEH 87]. The data described with the following models can be assigned to one of the association types from fig. 2.10.

Hierarchical models are created using a tree structure in which every piece of data is ordered within a hierarchical structure ('father-son' relationship). The advantage of these models lies in their sequential data access. However, complex relationships like (m:m) associations cannot be represented.

Network models can also include data types that have to be assigned to several hierarchies. However, the advantage provided by this data model of being able to represent the sort of multiple associations found in the real world comes at the expense of no longer being able to access data in a strictly sequential fashion and of having to exert greater effort in creating a data-manipulation language. (m:m) associations are broken down into two (1:m) associations.

Relational models summarise several data types in tables, whereby each line of these tables is called a tuple. Tuples are characterised by their attributes, which are in turn defined by the columns of the tables. Associations within a table that contain redundant information are divided into several smaller associations until all redundant information is removed. This process is called normalisation. However, data-type structure is lost using this method.

This disadvantage was removed by the introduction of **entity relationship models** (ERM) [CHE 76]. These are really no more than extended relational

models, these now have major importance in the field of database technology. ERMs allow entities ("clearly identifiable 'things' existing in the real world " [CHE 76]) and their interdependencies to be represented.

Example 2.5

Finally, object-orientated data models combine the properties of conventional data models with the advantages of object-orientated programming. The definition of an entity can be made more concrete by introducing the term "object".

- A complex matter is broken down into individual objects. These and possibly other objects arising from further subdivision as well as the respective associations all serve to structure the available facts.

2.2.3 Modelling Languages for Data and Information

Data-modelling languages have recently emerged with which formal, machine-compatible data-model definitions can be produced. These languages provide the following advantages:-

- Extensive models defined in such a language can be interpreted by both users and computers. It is thus possible to test these models for consistency in automated form.

- The user is shown a simple way which helps understanding and assists in consistent modifications of an existing data model.

EXPRESS [SCH 90] is a data-modelling language aimed at producing standard information models. It was developed in recent years to make data definition independent of the implementation language. EXPRESS defines neither the data's nor the database's structure. EXPRESS was much rather developed as a way of standardising product data and was supported by the STEP initiative (Standard for Exchange of Product Definition Data Model).

It must again be mentioned that EXPRESS is not a database development language. Nevertheless, it does have a number of basic features in common with the conceptual database model introduced in Chapter 2.2.2.

Example 2.6

EXPRESS-G [SCH 90] was developed to enable graphic representation of EXPRESS, whereby considerations were taken of the following factors:

- model diagrams had to be easy to understand,
- diagrams had to illustrate the different levels of model abstraction,
- it had to be possible to spread diagrams logically across several pages,
- the automatic transfer of text-based EXPRESS definitions into graphic EXPRESS-G form had to be possible,
- diagrams were to contain no special graphic symbols in order to keep demands on computer performance to a minimum.

2.2.3.1 Practical Applications of Data Models in Electrical Engineering

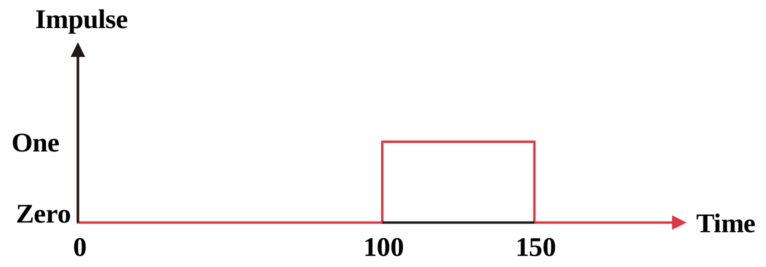
The field of electrical engineering has also been beset with increasing problems caused by incompatible system components. For instance, several megabytes worth of data can be involved in the design and testing of just one digital circuit. Numerous data conversions were hitherto necessary in order to ensure smooth data transfer between the system's design and testing tools. This problem resulted in the development of the Test Specification Format (TSF) [VER 1].

Data-modelling technology was used during the development stage of this data format. TSF is too complex for it to be described using just one of the data models introduced above. It combines features of the object-orientated approach with the advantages of the entity relationship model, which in itself is a re-development of an already existing data model. Associations are not explicitly included in the data model in accordance with TSF definition. These associations are much rather transformed into independent entities [BRA 90].

Fig. 2.11 and 2.12 show the relationship between data to be defined and the respective EXPRESS term. A square impulse is fully defined in EXPRESS by its name, time sequence and values.

As a complete TSF data model consists of a great number of such simple 'problems', EXPRESS-G was developed in order to give EXPRESS models graphic form and thus to increase model clarity. Despite the complexity of TSF, this graphic representation provides an overview of all the entities and their inter-relationships. An excerpt from an EXPRESS data model plus its corresponding EXPRESS-G representation can be seen in Fig. 2.13. EXPRESS-

G graphics are created automatically from the respective EXPRESS text notation with the help of a conversion program [DIN 90].



```
(frame IMPULSE drive
  (List_of_edge
    (edge (time_absolute 0) (logic_constant Zero))
    (edge (time_absolute 100) (logic_constant Zero))
    (edge (time_absolute 150) (logic_constant Zero))
  )
)
```

Fig. 2.11: A Square Pulse

```
ENTITY frame;
  name : name_def;
  drive_or_sense: direction;
  edges: OPTIONAL VISIBLE LIST [1:#] of edges;
UNIQUE
  name:
END_ENTITY;

ENTITY edge;
  name: OPTIONAL NAME_def;
  placement: time_point;
  state: logic_state;
UNIQUE
  name: -- if it exists
END_ENTITY;

ENTITY time_point
  SUPERTYPE OF (time_absolute XOR
                time_relative XOR
                time_percent);
END_ENTITY;
```

Fig. 2.12: Definition of a square pulse in EXPRESS

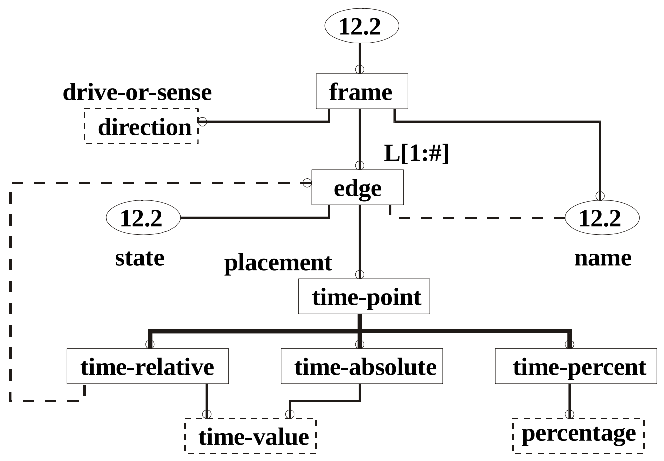


Fig. 2.13: Representation of an EXPRESS Statement Block using the graphical EXPRESS-G Form

The specifications of an EXPRESS-G representation enable the presentation of either a single schema (conceptual schema) or a multi-schema concept, as required by the task at hand [SCH 90]. The EXPRESS-G graphic representation relevant to the TSF constitutes a conceptual schema.

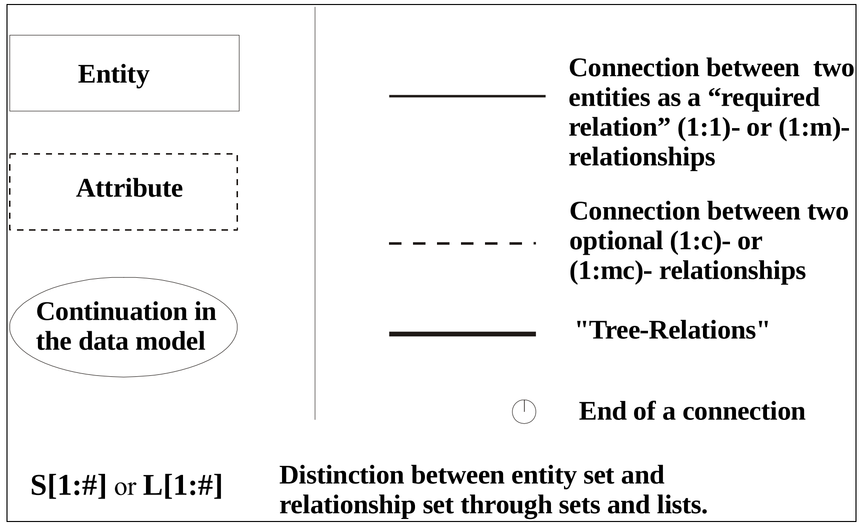

Entity
Attribute
Continuation in the data model
Connection between two entities as a “required relation” (1:1)- or (1:m)- relationships
Connection between two optional (1:c)- or (1:mc)- relationships
"Tree-Relations"
End of a connection
S[1:#] or L[1:#]
Distinction between entity set and relationship set through sets and lists.

Fig. 2.14: Basic Graphic Symbols of the EXPRESS-G Graphs

• The key elements , the entities, are symbolised by rectangles.

• An attribute of an entity is shown by a dotted line around the attribute's name. Such an attribute is called a "user-defined symbol" in the nomenclature of EXPRESS-G syntax. However, this says nothing about its value range.

• Ovals refer to continuation in the data model.

• A continuous line linking two entities denotes a required relation, which means either a (1:1) or (1:m) relationship.

• Optional (1:c) or (1:mc) relationships, however, are shown by a dotted line.

• "Tree relationships" are shown by bold-type continuous lines.

• Small circles denote the end of a connecting line.

• By introducing sentences and lists (S[1:#], L[1:#]), it is then possible to differentiate between entity and relationship sets.

2.2.3.2 The Entity Relationship Model (ERM)

In the following, the ERM will serve to introduce the basics of data modelling. The ERM belongs to the class of semantic data models used for conceptual design. Entities, entity sets and relationship sets form the basic concepts of ERMs, which also function as basic data models for numerous CASE tools.

Entities und Entity Sets

Entities can be any physical thing or abstract notion existing in the real world, e.g. a book in a library. Each entity can have any number of specific attributes. For a book, these could include title, publisher, ID code, etc. A range of permissible values can then be defined for each attribute, the so-called value range. Several entities of the same type are then grouped together to form an entity set. Staying with the above example, this could be given the name BOOK. Both the name of an entity set as well as its attributes generally remain constant over time.

Relationships

Differing relationships exist between the entities of different entity sets. These relationships are described in relationship sets. A relationship can involve several entity sets and can have its own attributes. In the case of a library, a possible relationship could be BORROWED BY, which would link the two entity sets BOOK INVENTORY and USER, thus identifying who has borrowed which book(s). A full description of a relationship also includes its specific value. This determines how many entities of one set can be linked to how many of the others. The entity sets are linked by using a minimally identifying attribute combination (key). For our library example, this would be either the user's or the book's ID code.

IS-A und Hierarchy Relationships

Further differentiation or hierarchical representation can be achieved with the help of IS-A and hierarchy relationships. In either case, though, these relationships may only involve two entity sets. Given our library example, magazines (as a special form of book) could constitute an IS-A relationship. This relationship would then contain all the attributes of the entity set BOOK, but also other attributes solely applicable to magazines, such as publication intervals. A father-son association is a classic example of a hierarchy relationship.

2.2.3.3 The Entity Relationship Diagram

One of the ERM's major advantages is its simple graphic representation. Fig. 2.17 provides an overview of the symbols for the basic concepts.

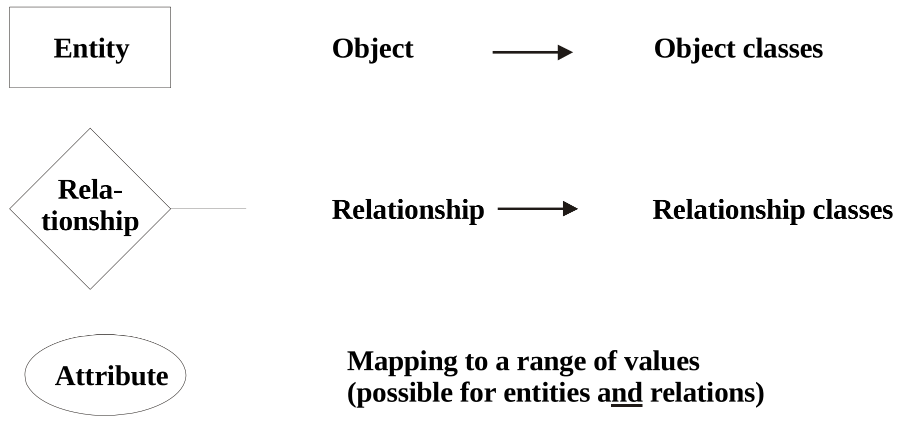


Fig. 2.15: Basic Graphic Symbols of the Entity Relationship Model

These symbols enable the following graphic representation:

- **Entity Set**
  by a rectangle containing its name.
- **Relationship Set**
  by a rhombus containing its name. The entity sets involved are connected to the rhombus at the undirected edges. The value of the relationship is noted above the edge. Relationship attributes are connected to the rhombus via undirected edges.
- **Attribute**
  by a circle containing its name. Primary key attributes should be underlined. The attributes of an entity set are connected to it by undirected edges.

Value ranges cannot be represented by ERDs. Various entity relationship models are shown in fig. 2.16:

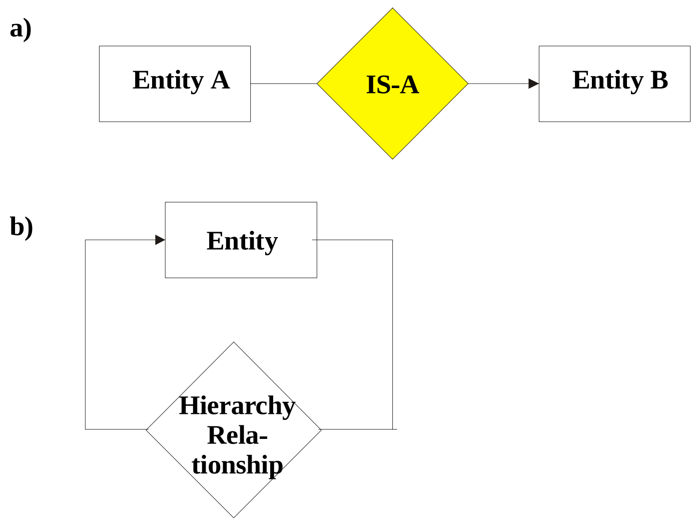


Fig. 2.16: Entity Relationship Models
a) IS-A Relationship
b) Hierarchy Relationship

- **IS-A Relationship**
  by a rhombus labelled IS-A. The two respective entity sets are interlinked by a directed edge, whereby the edge points towards the general entity set. The notation of specific relationship attributes is permissible and takes the same form as for relationship sets.

- **Hierarchy Relationship**
  by a rhombus whose start and end points rest on the same entity set. Hierarchy relationships are always 1:n associations.

Example 2.7

Example 2.8

2.2.3.4 Implementing an ERM in PASCAL

ERMs can be implemented in various different ways. For our library example, representation using linked lists would be most suitable as this represents a dynamic data structure which does not limit the number of possible entries. Such a representation translates both entity sets and relationship sets into their own respective lists. IS-A sets are represented by assigning specific attributes to the general entity set BOOK.

```
TYPE  bookPointer = ^book;
      book = RECORD
               bookID          : STRING[7];
               title           : STRING[35];
               publisher       : STRING[35];
               publ_year       : INTEGER;
               publ_interval   : STRING[15]
               next            : bookPointer;
             END;
TYPE userPointer = ^user;
     user = RECORD
              userID           : INTEGER;
              surname          : STRING[20];
              firstname        : STRING[20];
              address          : STRING[50];
              memb_Since       : STRING[9];
              next             : userPointer;
            END;

TYPE borrowed_byPointer = ^borrowed_by;
     borrowedby = RECORD
                    bookID    : STRING[7];
                    userID    : INTEGER;
                    next      : borrowed_byPointer;
                  END;
```

Fig. 2.17: Data Administration in a Library Using Linked Lists

The individual data sets are accessed or selected via the list element, whose respective attribute was defined as a KEY in the ERM.

3 Structured Design

3.1 Structured Analysis (SA)

This method represents a quasi-standard for top-down design. It was developed in the early 1970s by DEMARCO [DeMA 78].

This method offers a syntax which is powerful yet simple. It can be used for the
• collection
• structuring and
• conveying
of demands and ideas about existing or future systems.

First of all, the syntax defines data flow diagrams that illustrate
• system activities,
• their interfaces with each other and
• with the environment

For notation the following symbols are used:

→	**dataflow**	**Arrows showing direction of flow**
○	**process**	**circles**
=	**file**	**horizontal pair of lines**
□	**data-source, sink**	**rectangular box**

Fig. 3.1 Symbols used in the DEMARCO Notation of Data Flow Diagrams

Data flows
As expressed by the term "data flow" (as well as the symbol *directed flowlines*) data flows mark paths on which information about
• processes
• processes + files or
• processes + sources and sinks
are transported. Information is sent either uni- or bidirectionally where the direction is always determined. Data flows are given unique and unambiguous names that clearly describe their contents. Data that flows to and from files are not labelled, since the data type is already identified by the file name. Only if needed are data flows procedurally connected with
* : AND, conjunction
+ : OR, disjunction
? : exclusive OR
It is important to be careful when using procedural connections, since resulting diagrams tend to become complex rather quickly and thus interpretation can become complicated.

Processes
Processes carry out the transformation of incoming data flows into outgoing data flows. This is why they are labelled explicitly, explaining the on-going procedures. Additionally, they are often numbered (cf fig. 3.4) to render the schema non-ambiguous.

Files
Files offer storage space for data. They are abstract, delay-free and infinite. Even a database can be regarded as a file. The direction of the data flow, either towards or away from the file, indicates whether the file can be read or written into. If necessary, a double arrow can be used.

Data sources and sink (Terminators)
A data source represents a person, organisation or a complete system that sends data to the system as explained above. A data sink receives data from the system. Data sources and sinks are also called terminators since they always *terminate* the data flow of the system.

Example 3.1

The structuring of diagrams should follow criteria that have been recognised in practice and are ergonomically sound, i.e.
• An individual is usually able to differentiate between approximately seven objects in a diagram.
• The paper format A4 is the recommended size for the description of one diagram unit.

Example 3.2

This automatically leads to a formation of levels by successive downward levelling. Starting point of all downward levelling is the context diagram. Characteristically it only contains one process that represents the model of the whole system. All relations and data flows of this system are represented as arrows pointing to the terminators. All of the terminators together comprehensively describe the environment in which the system is supposed to function.

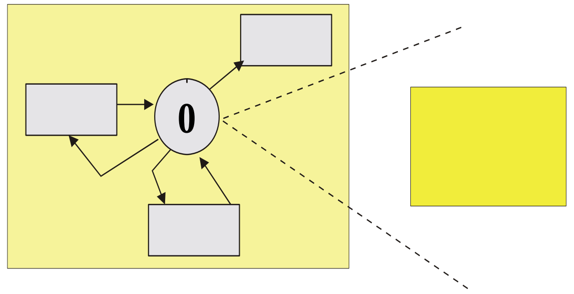


Fig. 3.2 Schema of a Context Diagram [RAA93]

Fig. 3.3 shows the first transformation of the design, which is also called the zero-level (Level 0). Level 0 is based on the context diagram of fig. 3.2.

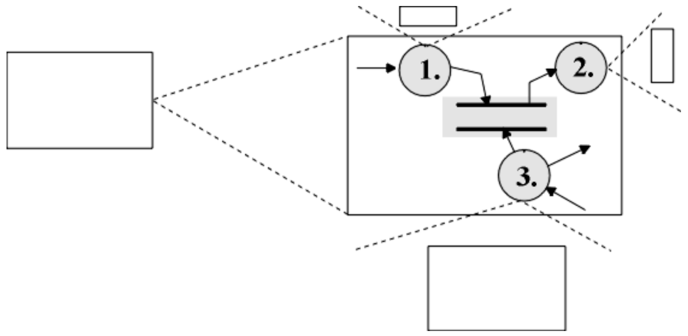


Fig. 3.3: Schema of the First Transformation - Level 0 [RAA93]

• Functions and
• data flows
are analysed and successively transformed.

The transformation numbering together with the number of the diagram forms a clear naming scheme for the hierarchical transformations. Figure 3.4 shows the downward levelling of transformation No. 3 in fig. 3.3 in three transformation steps 3.1 to 3.3. The number of the illustration always refers to the corresponding number of the transformation at the next-higher level.

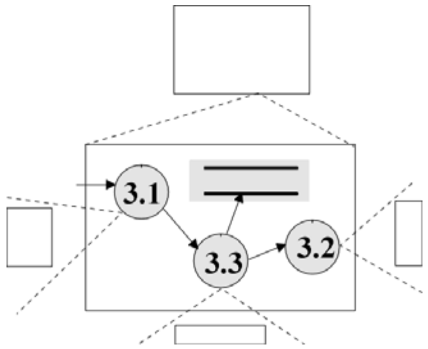


Fig. 3.4 : Schema of the next Transformation - Level 1 [RAA93]

Downward levelling is carried on until only primitive basic functions are left.

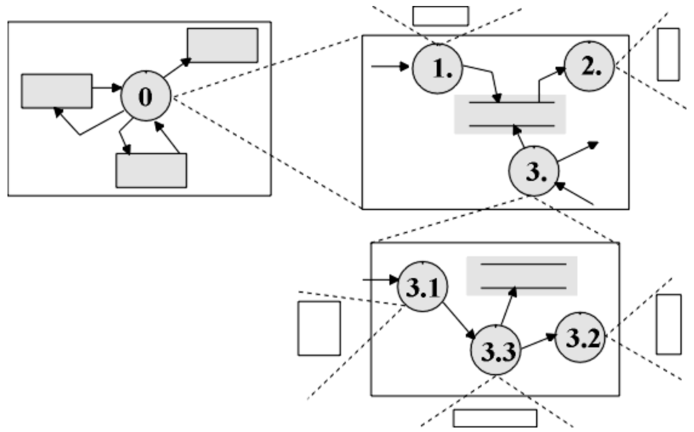


Fig. 3.5: Complete Model Hierarchy [RAA93]

Example 3.3

As a further example, let's apply data transformation to the making of carrot soup. The corresponding transformation diagram is shown in fig. 3.6

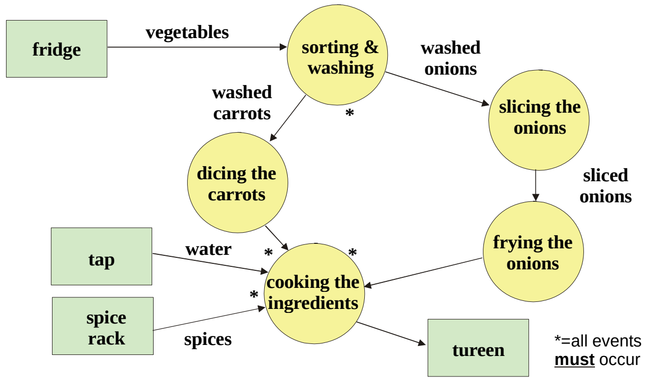


Fig. 3.6: Transformation Diagram describing the Preparation of a Carrot Soup

The diagram in fig. 3.6 is an example of non-hierarchical downward levelling. It contains the terminators as well as a first downward level of the system "making carrot soup". The procedural character of the process is represented by the

conjunctive connection of several data flows. The single steps of modelling result from the following procedures:

• Ingredients: vegetables (carrots, onions)
• get vegetables, wash and prepare them
• fry onions
• cook ingredients in water adding spices
• serve

3.2 Structured Design (SD)

Following the description of the basic methods, this chapter deals with structured design as a procedure in its entirety. As a first example, consider the system "bank":

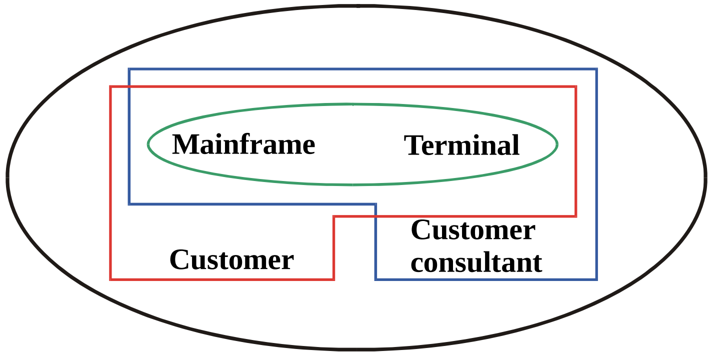


Fig. 3.7: Description of the System "Bank"

By setting up system boundaries
• the range of functions to be planned and
• the necessary language agreements at interfaces
are determined.

The crucial question is: Who is the software aimed at?

In our example the first level is the mainframe and its communication with the different terminals in the subsidiaries. It is exclusively controlled by an expert computer operator.

The second level includes the customer representative who can access the mainframe from his/her PC terminal.

At the third level the customer can access the system as well, e.g. with home banking.

When designing software a classic dilemma lies in the decision between a implementation-dominated approach and a problem-oriented one. Implementation is dominant when new technologies are employed. This implies, of course, that problems and boundaries also determine the advantages hoped for, during design proceedings. If, however, already existing and mature technology is used for implementation, a problem-oriented approach is dominant. It is supposed that the technology chosen is capable of sufficiently solving all design aspects. The design itself can then focus on the optimal solution to the problem.

3.1 Separating Essence from Implementation

In practice engineers should opt for a problem-oriented approach to work, i.e. they are supposed to solve a given problem optimally from the users point of view.
The ever decreasing length of innovation cycles in software technology however leads to a dominance of the implementation-dominated approach. This in turn may– quite unnecessarily - lead to less than optimal problem solutions in spite of or – probably even because of – the use of the latest technology. A problem-oriented approach - which has proved successful with structured analysis - is shown in the following figure:

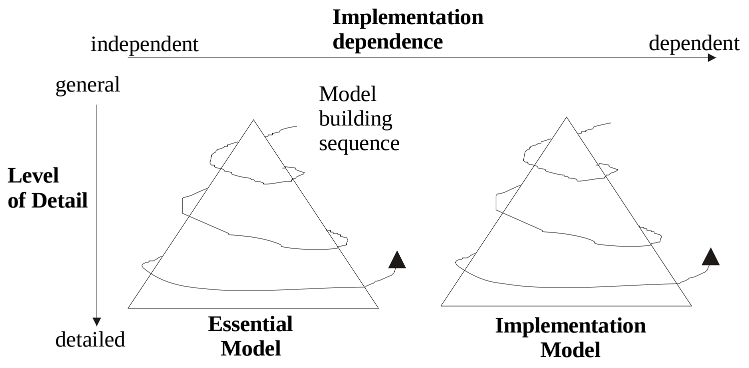


Fig. 3.8: Relationship between the Essential Model and Implementation Model

Essential:	problem-oriented, subject-oriented, compiled/developed and checked by an expert solely for this problem regardless of details and techniques of implementation =>smallest, complete model
Implementation:	Mapping of the essential model onto a technological environment ⇨ two consistent models

3.2 Characteristic Features of Modelling

Let us now look at certain specific features of modelling which draw a clear dividing line between modelling and the classic computer model. These specific features serve to model real problem cases as realistically as possible.

3.2.1 Structure of Modelling Phases according to WARD/MELLOR

1. Essential Model

1.1 Environmental Model
- Context Diagram
- Event, Control and Data List
- Information Model
- Data Dictionary

1.2 Behavioural Model
- Transformation Diagrams
- State Transition Diagrams
- Extension of Data Dictionaries
- Data Models

2. Implementation Model

- Processor Model
- Task Model
- Module Model

3.2.2 Differences between WARD/MELLOR's and DEMARCO'S Procedure

1. The design is separated into two models, so that
   - essential features and
   - system features arising through implementation

   are listed separately.

2. Separation of data flows into
   - Data flows and
   - Control flows

3. Replacement of operators
   (*, +,...) with lines

3.2.3 Breaking Away from the Von Neumann Structure

The classic model of the Von Neumann computer was developed in the beginnings of computer technology. It has served as the architectural model for various computer systems. Fig. 3.9 a) shows the hardware structure of such a computer. The basic components, i.e. CPU, memory as well as input and output are connected through a common bus. As can be seen in fig. 3.9 b), the memory contains both commands and data. This means that the interpretation of individual memory locations takes place during program execution and thus, all programs have to be run sequentially.

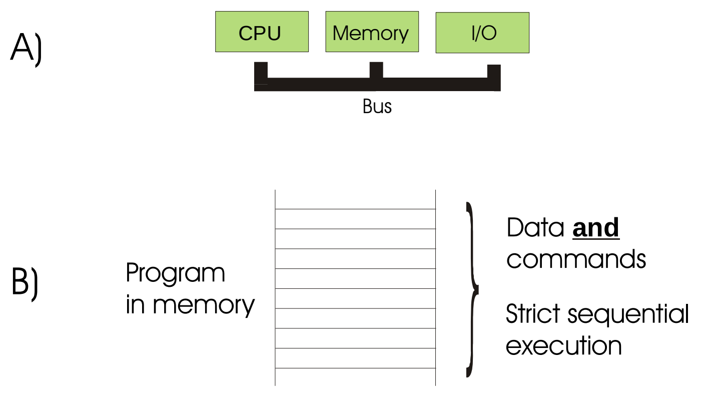


Fig. 3.9: The Von Neumann Computer
a) Hardware Structure
b) Program Structure

This concept was developed with the goal of minimising the amount of hardware required (which used to be extremely expensive) and making the best use of memory capacity (which also used to be expensive).
Due to enormous improvements in hardware with prices decreasing at the same time, various alternatives as well as partial optimisations of hardware have been developed in the last years, such as processors with multiple pipelines, vector processors and array processors.
The function of the computer, however, is crucial for the software used, the virtual processor. There have not been any radical changes in programming languages and methods used so far, at least not in the industrial sector.
But since all common programming languages, such as FORTRAN, PASCAL and C, already hold the Von Neumann concept as a language agreement and since the development of alternative languages and programming paradigms

shows little success despite huge research effort, the Von Neumann concept will continue to determine the possibilities and limits of computer systems in the future.

In practice real parallelism is perfectly feasible and in many areas of physics and engineering real parallelism is the norm.

• A simple example is vector addition. In order to calculate the sum of two vectors the addition of x- and y-components of two vectors are completely independent of each other and can be carried out in a parallel fashion. Every (Von Neumann) sequentialisation is arbitrary.

• The surveillance of a chemical reactor is a more complex system which synthesises macromolecular products from simpler structured elements. The result of a reaction is determined by the mixing ratio as well as pressure and temperature. The regulation of this process is also influenced by interdependencies between these three parameters. Due to its sequential functioning a classic Von Neumann computer is not capable of measuring all parameters at the same time simply because of it's sequential nature. Parallel processing would allow the computer to determine the three parameters *in parallel* and to thus intervene in the technical process directly. Even though most technical processes tolerate a certain time delay of individual computer actions, this basic problem has to be noted.

3.4.4. The Separation of Data and Control

Historically, the first improvement of the Von Neumann computer was the separation of processing and control functions within the CPU (arithmetic logic unit and control unit). Even though commands are divided in the same way with high-level languages (e.g. processing with +, -, *, SQRT, etc. and controlling with IF, THEN ELSE, WHILE, REPEAT, UNTIL, etc.) most programming and design procedures do not take this allocation of tasks into account. The isolation of data flow and control flow from a global task, as well as a consistently separated modelling and implementation, as it has been introduced especially by WARD and MELLOR, facilitates the control of real-time procedures, already in the design phase. This eventually leads to functionally clearly separated program modules and immensely facilitates implementation as well as maintenance.
The proportions of processing to controlling in a specific design depend on the actual task at hand. A rough division can be made according to:

• **Real-time Systems**:	Data for processing and events for controlling
• **Transaction Systems**:	Data flow suffices for modelling

All programs that primarily process data about specific transactions in reality - with or without (at least liberal) real-time conditions - are referred to as transaction systems. Such tasks mostly come up in the sales sector (orders and accountancy, accounting for insurances, banks, etc.)

3.4.5. Continuity and Discontinuity

Data flow and event flow can be further distinguished according to the chronological occurrence of their information content:
Permanent information flows are called continuous, e.g. sensor readings. Flows with information only at specific points in time are called discontinuous or discrete, e.g. course data of a ship when crossing the Atlantic Ocean. Even though differentiating between the two definitions can sometimes be difficult in practice, a detailed discussion of this problem can be very useful when calculating necessary resources and real-time behaviour.

3.5 Modelling and Downward Levelling

The basic principle of modelling and downward levelling is the goal of maximum independence of the respective transformation or modules being created. Specific variations of this principle are:

• organisation for maximum independence
• organisation of functionality:
Objects belonging together are united and objects not belonging together are separated.
• organisation of data to be exchanged:
In addition, partitioning is to be carried out according to the principle of minimal interfacing. The amount of agreed data sets between partitions (parameter list) as well as the frequency of their exchange (communicative effort) are to be minimised.
• organisation of local data:
The principle of *information hiding* (aka data encapsulation) makes data that is only used locally unrecognisable outside the boundaries of this module. This increases system clarity and helps to avoid incorrect modifications of data.

3.6. Storage and Transformations

Storage is not part of the context diagram, but part of data transformation. They have to be derived from the transformation during downward levelling. Fig. 3.10 a) shows a transformation that combines a storage function (state of the air space) and transformation (add aircraft). Consequently, resulting data flows are unnecessarily complex and the actual storage location is unknown. The separation of both functions (cf fig. 3.10 b) clarifies the contents of the data flows as well as the allocation of transaction and storage. In addition, fig. 3.10 b) shows a better modelling flexibility: further functions on the state of the air space - with minimal interfacing regarding the common storage - can be realised without any problem.

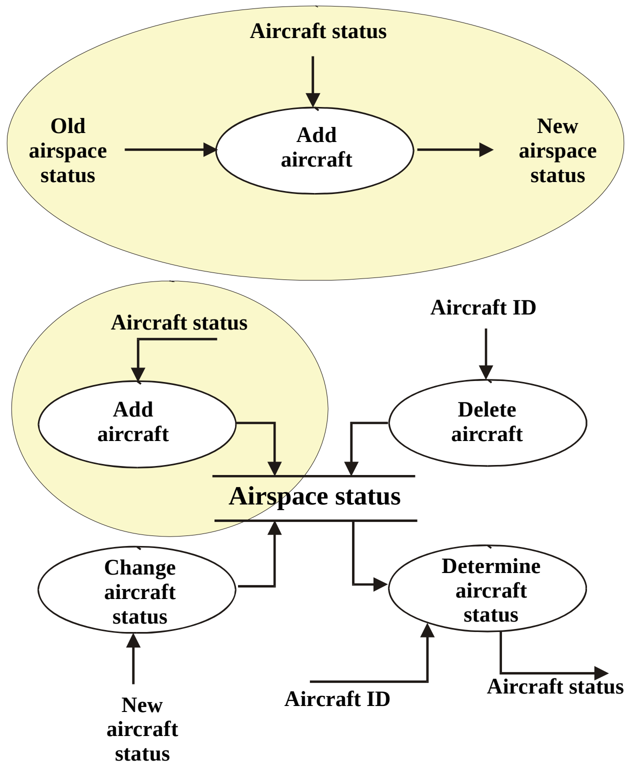


Fig. 3.10: Storage and Transformation [WAR 91]
a) Combined Transformation
b) Separation of Storage and Transformation

This separation makes it necessary to mark the data flow between transaction and storage. Fig. 3.11 shows the customary agreements, with a) indicating availability only and b) indicating the selective use of file parts and the direction of the resulting flow.
The example shown in fig. 3.11 can be regarded as an exception to the rule (especially with the description of pure availability), since the data flow does not need a specification of its own. This is due to the fact that the file as a whole has to be specified and specification of the transformation possibly determines the necessary part of the file.

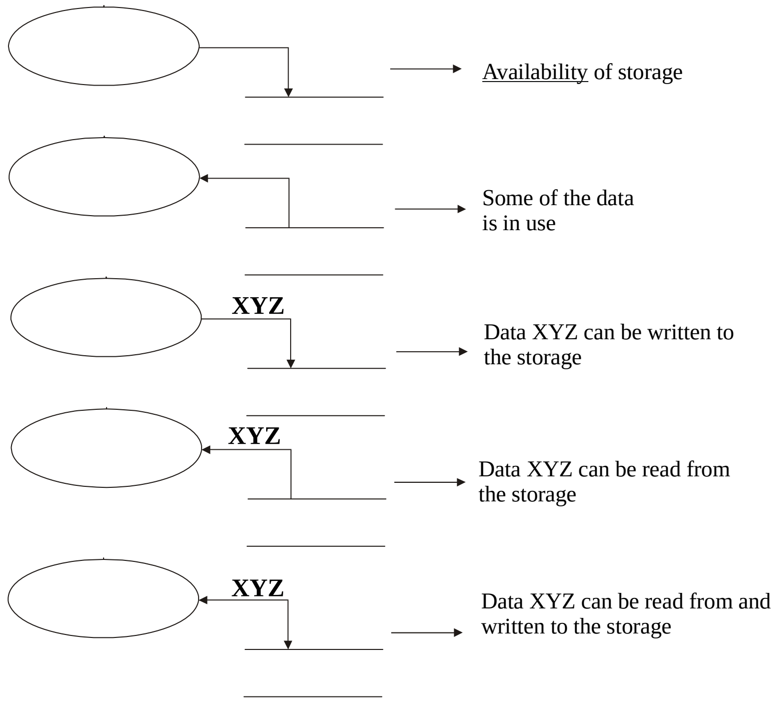

Availability of storage
Some of the data is in use
XYZ
Data XYZ can be written to the storage
XYZ
Data XYZ can be read from the storage
XYZ
Data XYZ can be read from and written to the storage

Fig. 3.11: Possibilities of Linking Transaction and Storage
a) Marking of Availability
b) Use of File Parts

Apart from the static data flows in fig. 3.10 there are also dynamic data flows on the basis of continuous data (cf Fig. 3.12). While the position of the aircraft is continuously[1]recorded, data of an approaching plane is only stored when the plane has identified itself (by a discrete data flow).

1= Referring to the operation mode of a radar the term *continuous* might be questionable.

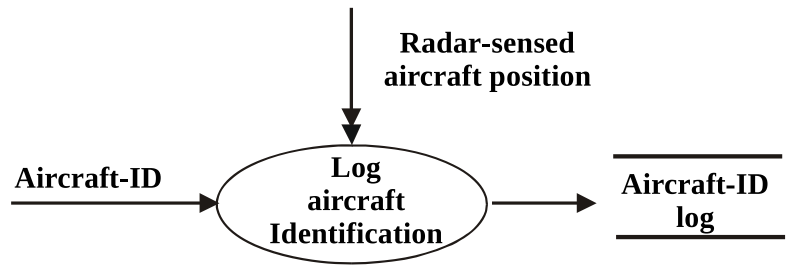


Fig. 3.12: The Filtering of a Continuous Data Flow (cf [WAR 91])

Every data transformation can be switched on or off by one or more control transformations (cf fig. 3.13). The signals *Enable* and *Disable* describe start and shut down of the system, e.g. at the beginning and end of flight operations. These control signals are events rather than data flows; events that merely represent the arrival of the meaning agreed upon. With downward levelling of a transformation schema these events are also called *prompts* or *triggers*.

It should be noted that control transformations cannot accept data flows as inputs under any circumstances.

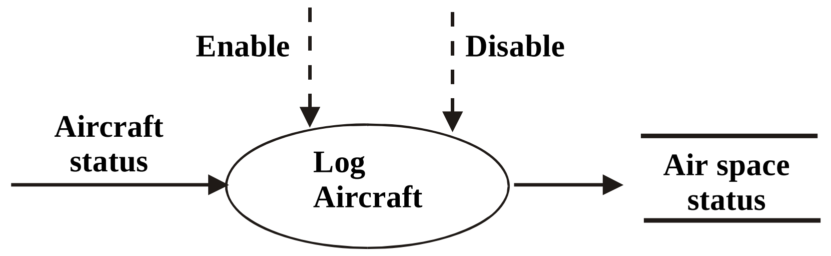


Fig. 3.13: Controlling of Data Transformation (cf [WAR 91])

Control transformations pool all control functions of software.
For this purpose it
• records input events/triggers
• issues events/triggers and
• comprises the system state in the form of a machine model

As an example fig. 3.14 shows the modelling of a lamp with a pull cord. This is a mere control transformation which generates the two triggers "turn lamp on" and "turn lamp off" from the event "pull" (cf fig. 3.14a).
Whether the pull cord switches the lamp on or off cannot be understood from the transformation alone. Additionally, an associated machine model (state transition diagram) is always needed. This state transition diagram then gives us information about the function, e.g. for the pull cord (cf fig. 3.14b).
The use of the memory symbol for expressing the system state is naturally impossible since the memory cannot provide us with instructions (Y) in dependence of input (X) and states (Z). In the case of a machine model, however, this is very simple:

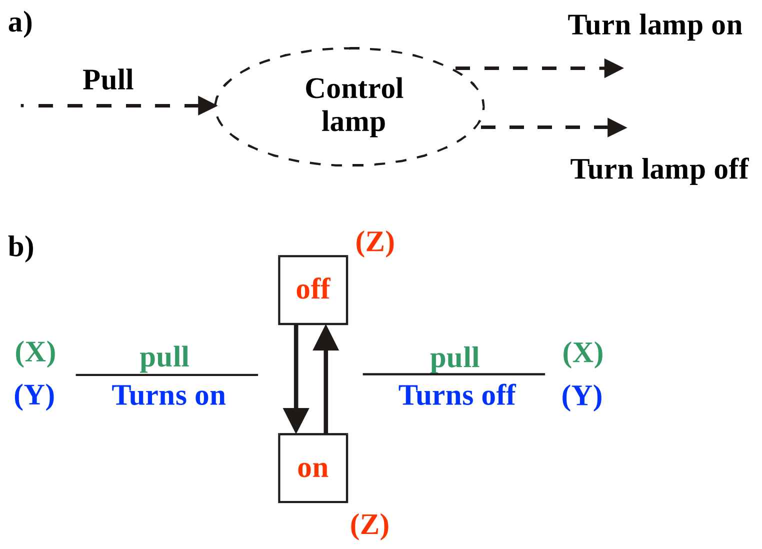


Fig. 3.14: Modelling of a Lamp with Pull Cord (cf [WAR 91])
a.) Control transformation
b.) corresponding Mealy-machine

A rather complex example of a control transformation is the signal control for a tunnel. Because of its length the tunnel can maximally hold four trains simultaneously. The entry signal is controlled depending on the number of trains in the tunnel.

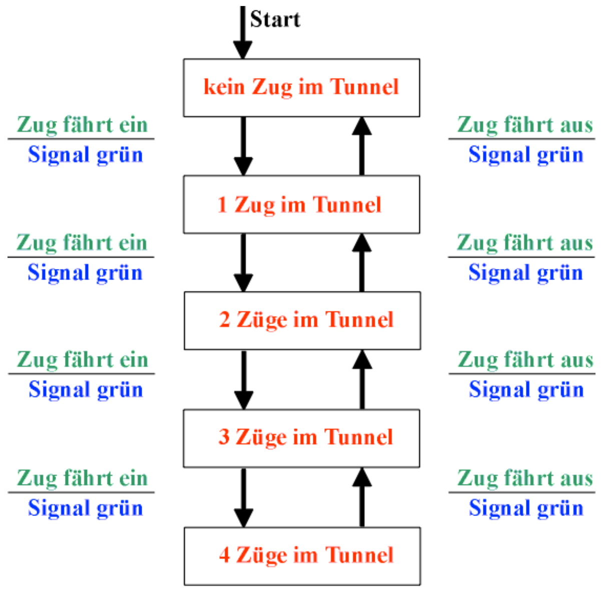


Fig. 3.15: Signal Control in a Tunnel with multiple similar states (cf [WAR 91])

Fig. 3.16 shows a compact representation of the machine model for fig. 3.15. Since the system state merely contains a counting function going forwards and backwards, a single state with variable contents is given for that. In the case of rather complex systems this surely is the most sensible solution for representation. It has to be noted, however, that this now presents a Moore-type machine with the output solely depending on the state Z (cf fig. 2.7).

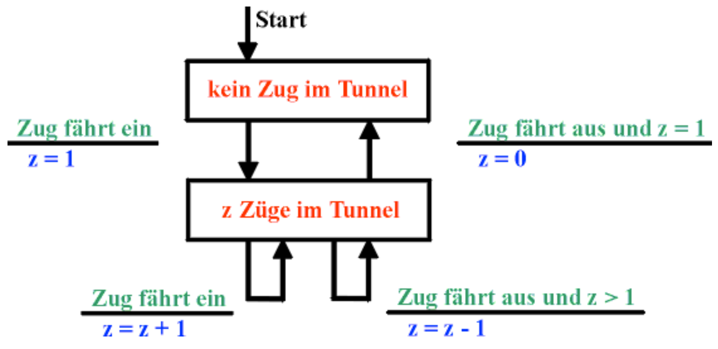


Fig. 3.16 Signal Control in a tunnel using a compact representation for multiple similar states (cf [WAR 91])

The interlocking of two (or more) machine models produces a further variation of the control transformation. The example given deals with a tunnel again; this time, however, it is a single-track tunnel within the course of an otherwise double-track route that can be used in both directions.

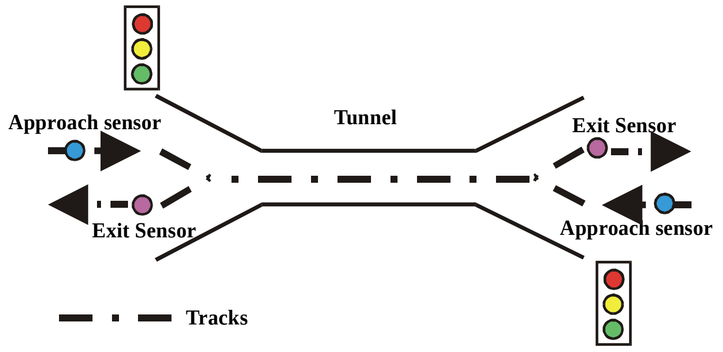


Fig. 3.17: Model of the Signalling System for a Tunnel Used in Both Directions (cf [WAR 91])

As trains approach the tunnel a sensor announces their arrival. If the light is *red*, the train will wait until it changes to *green*. Once traffic has started in one direction, the signal for this direction remains *green* and further trains can pass through the tunnel. The signal turns *red* when the last train has left the tunnel. In the meantime a train approaching from the opposite direction has to wait until the oncoming traffic has ended.

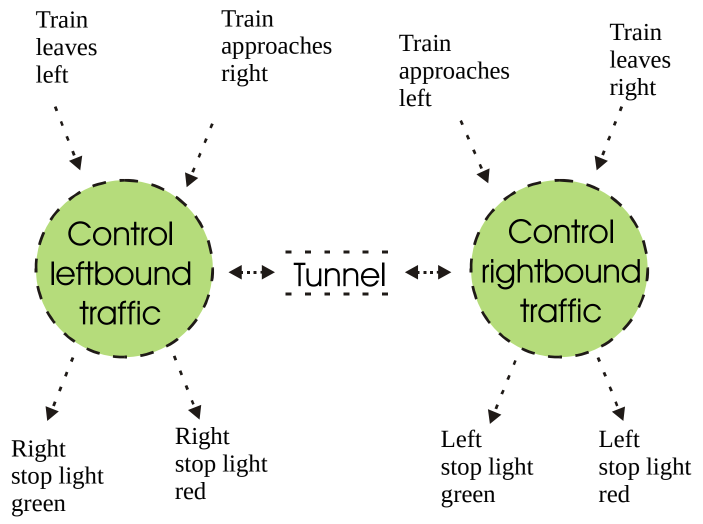


Fig. 3.18: Model of the Signalling System of a Tunnel - Control Transformation (cf [WAR 91])

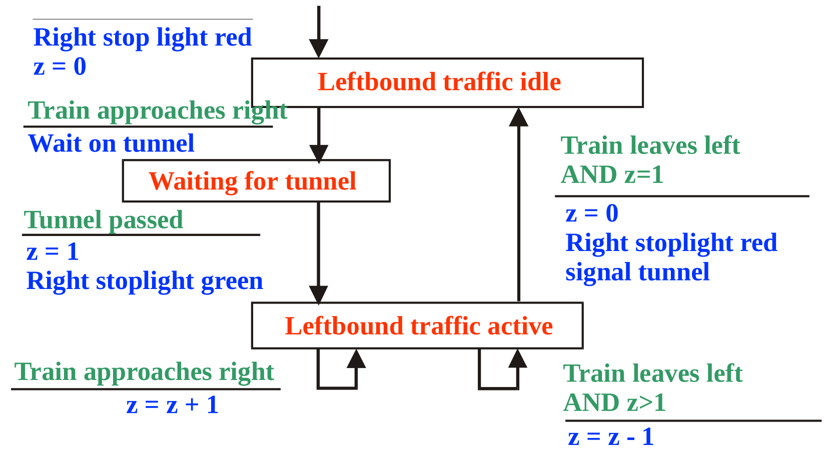


Fig. 3.19: Model of the Signalling System of a Tunnel - State Transition Diagram for Fig. 3.17 - Left Side of the Tunnel (cf [WAR 91])

Example 3.4

3.7 Task Summary for Constructing the Essential Model

1. Context diagram
• Definition of interfaces for the environment,
• Definition of a system,
• Definition of several terminators – sensibly united -
• List of all data flows and event flows

2. Downward levelling
• Minimal interfacing
• Separation of data transformation and control transformation
• Control transformations only have event flows for input/output.
• Data transformations only have data flows for input/output. Event and triggers can only serve as input.
• Both data transformations and control transformations *Enable* and *Disable* are triggers (prompts), not transformation input in the sense of data flows.
• Only control transformations can generate triggers for other transformations.

3. Resolution of data transformations
• Data dictionary (---> data format synt./sem.)
• Transformation rules (---> procedure) with pre- and post-conditions

4. Resolution of control transformations
• State transition diagram

4 Implementation model

After the downward levelling of the essential model has been completed, we enter the second phase of modelling, which has to take all resources into account. This second phase of modelling is based upon data flow and control flow of the behavioural model. Its aim is the mapping of these flows onto available hardware as well as onto the structure of the software to be designed.

As with the essential model, the implementation model is developed top-down, i.e. based upon an abstract general overview. Model design is divided into three successive phases: the processing model, the task model and the module model (cf fig. 4.3).

4.1 Processes and Process Administration

Definitions:

Processor:	Hardware used for the processing of data
Process:	program being executed or virtual processor. This includes programs, data, program counter, stack (pointer), status, register, etc.

Processes are described via context and are registered with the process control block (PCB). The PCB often consists of two parts:

a. HW-PCB (internal, operating system)
• description of current process
• or (if suspended): status of all frozen variables

b. SW-PCB (external, programmer)
• identification
• state
• priority
• etc.

Administrative tools:

Running of a single process on a single processor is simple (e.g. the standard personal computer), but if several processes have to be run on a single processor or maybe even distributed across several processors the above-stated definitions are of critical importance. The tasks of the usual administrative tools which carry out these processes are illustrated in fig. 4.1. The scheduler assigns up-

coming processes to available processors. During this procedure functional dependencies of processes and/or a fictive, superordinate time concept are of prime importance. The dispatcher, on the other hand, carries out the allocation of processors to certain processes, based on interrupts (possibly with priorities) and real-time conditions.

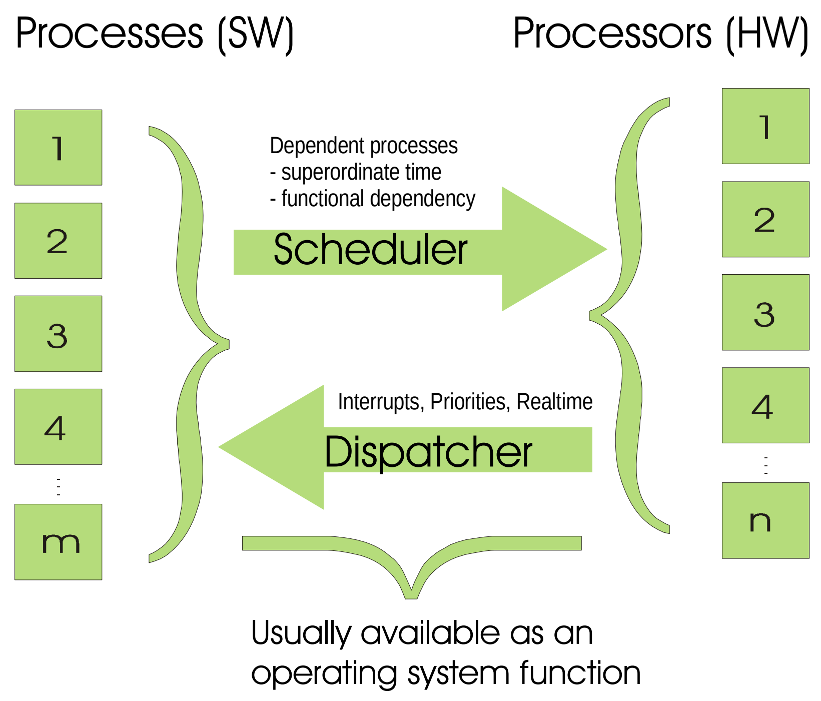
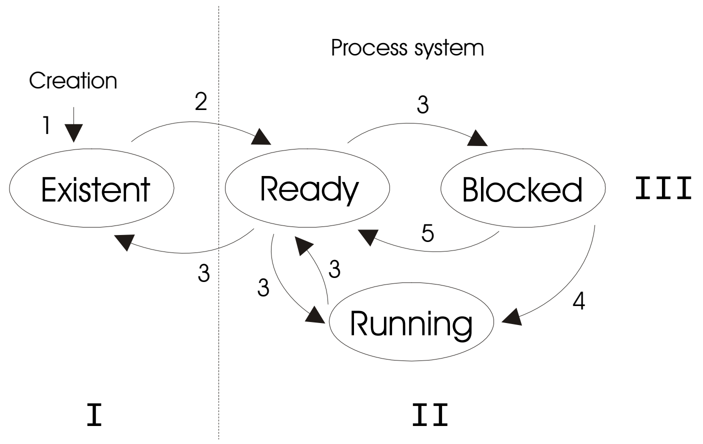


Fig. 4.1: Process Administration with Scheduler and Dispatcher

Several possible process states are defined for regular process administration. Taking also conditions and triggers for the transition between states into account, the following machine diagram (fig. 4.2) can be set up*:
(*Only parts I and II are relevant for the practical training.)



1.) By a running program
2.) Integration into the process system
3.) Interruption prompted by either dispatcher or scheduler for priority/time reasons
4.) Process is waiting for an input
5.) Input has arrived

I: Multi-process initialisation
II: Multi process operation (user controlled)
III: Additional effect caused by quasi-parallelism without user control

Fig. 4.2: States and Administration of Processes

A process computer
• allows the execution of several data manipulation processes (quasi-parallelism)
• allows real-time measuring, controlling or regulation of technical processes

From now on the terms *process* (in the sense of data processing) and *task* will be used synonymously; the former originating from data processing and computer science respectively, the latter stemming from the user's point of view. As far as process computers are concerned, the terms are identical. However, the terms *task* and *user* have to be clearly distinguished. At least one task (or the solution of which) will be assigned to every user. However, a user can request several tasks without any problem, and similarly, several tasks can be assigned to no user at all, since they simply co-operate with a technical process. Thus, a process computer is originally a multi-tasking computer which can be used as a single-user computer as well as a multi-user computer.

A multi-tasking operating system usually allocates constantly upcoming tasks (usually to the capacities of a single processor), as can be seen in fig. 4.3:

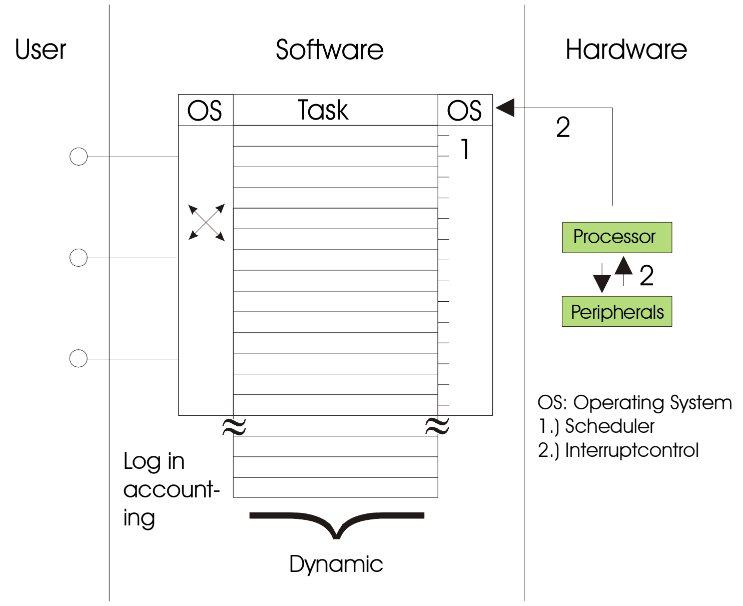


Fig. 4.3: Main Functions of a Multi-Tasking Operating System

For the sake of simplicity the following explanation of administrative mechanisms will begin with the structure of a multi-tasking administration, then moving on to the co-operation between computer and technical processes. In practice both mechanisms are intertwined.

4.2 The Structure of Multi-Tasking Operating Systems

As new users log into an operating system, access rights are monitored (log in) and fixed contingents of computer time, storage space and further resources are assigned. Furthermore, an account is opened by the operating system to keep track of the computer time used and possibly of the executed functions. Accounting allows the control of all computer operations during use as well as afterwards. In addition, an actual bill for computer services used can be produced. For every user his/her tasks are set up in a dynamic administrative chart. Apart from that, there are a number of general administrative tasks, e.g. *log in* and *account*, as mentioned above, as well as the usual terminal- I/O functions, print, etc. The scheduler assigns the processor to work on the different tasks in a time-sharing operation mode. (cf fig. 4.4)

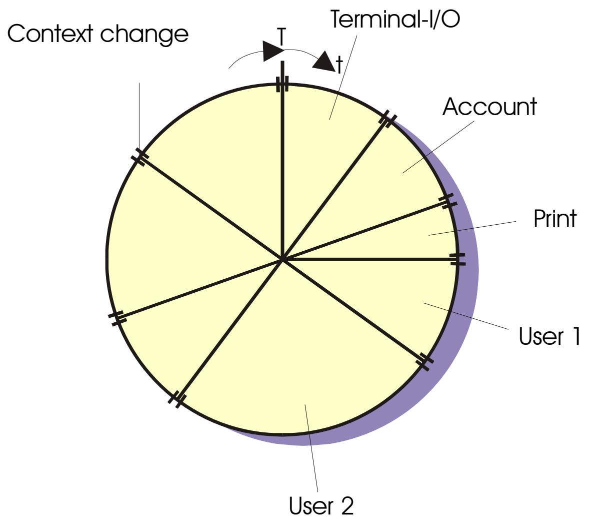


Fig. 4.4: Schematic Representation of Time-Sharing Operation by the Time Wheel Model

The time wheel model is based upon the notion that one begins with the first task at a time t_0 (e.g. the terminal-I/O in fig. 4.4) At t_1 the scheduler interrupts this task, changes context, and, as a result, moves on to the second task (e.g. account in fig. 4.4). This sequential allotment of time slices and changes of contexts continues until the time slice of the last task has been completed at a time T and a new turn of the time wheel starts with the renewed processing of terminal-I/O. Context changes are used to reload process-specific registers of the processor, i.e. to switch from one virtual processor to the next. (For a more detailed description of this process see below).

The second main task of a multi-tasking operating system is the synchronisation of running programmes with technical processes. The interlocking of data-technical processes with value entry or user input, for example, can be executed

a) by cyclic flow control (polling). i.e. program-controlled

b) time-controlled in certain given intervals with a real-time clock

c) by request (interrupt) via the technical process event-controlled

Process requests (triggered by the technical process):

• can be announced at any time

• have high priority (importance) by
  - priority in execution and/or
  - blocking other requests

• also lead to context change

An example of this is shown in fig. 4.5. A background process (this can also be the time-sharing operation) is interrupted by two interrupts.

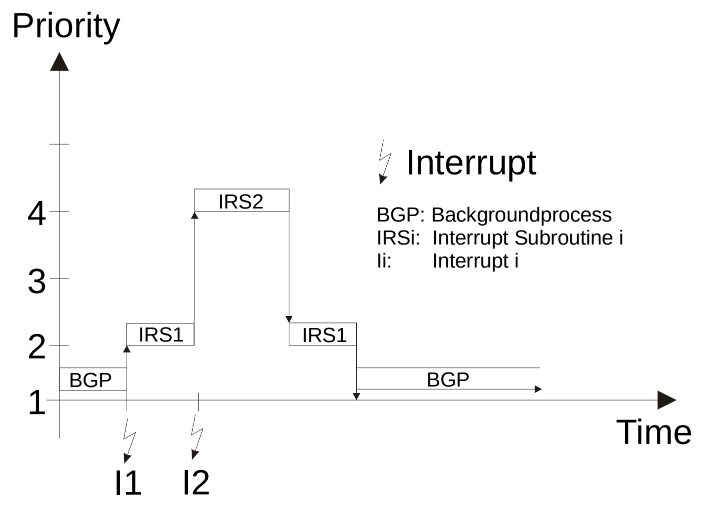


Fig. 4.5: Interrupt Control

Here the priority of the background process is at its lowest at 1, i.e. every interrupt is able to interrupt this process, as can be seen with I1, for instance.

The priority of IRS 1 is still set relatively low at 2, so that the process can easily be interrupted by I2 with its higher priority IRS 2. In this example IRS 2 can complete its task without any further interruptions before IRS 1 can complete its remaining tasks. Finally, the background process is resumed.

As mentioned above, time-sharing operations and interrupt control usually overlap.

4.3 Modelling of the Implementation

We will now describe the systematic development of the implementation model according to WARD and MELLOR. The position of the implementation model within the overall development process is illustrated in fig. 4.6. Fig. 4.7 describes the three partial models that are based on each other.

Activity	Commentary
Build essential model	Describe required system behaviour
Build implementation model	**Describe automated technology organisation that embodies required behaviour**
Build system	Embody implemen-tation model in hardware and software

Fig. 4.6: Evolution of a System

Processor Model	Description of the chosen allocation to processors and their interfaces	Transformation schemata and data schemata with their specifics	Description of transformation and stored data allocated to processors, tasks and their interfaces
Task Model	Description of the chosen allocation to tasks and their interfaces		
Module Model	Description of the chosen allocation to modules and their interfaces	Structure chart with their specifics	Description of the hierarchical organisation of modules in a program

Fig. 4.7: Implementation Model Layout

In the processor model HW units are defined with the goal of minimising communication. This is realised by adopting data flows and control flows from the context diagram, possibly also special additions. Downward levelling then leads to the design of task models per processor.

Here subtasks are defined and especially

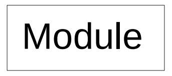
• their interfaces and
• the necessary cache memory is determined.

Hereby the conventional syntax of data transformations is used with

• labelling conditions for activation per subtask and
• choosing a form of realisation for events/triggers.

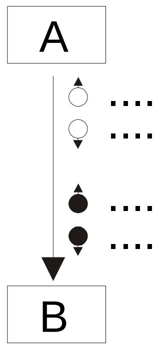
A new syntax is introduced for the concluding module diagram (cf fig. 4.8).

a) Module



: Function or procedure or a collection of these if they form a single functional unit.
(Practical tip: One has to be able to describe the function of a module in a single sentance)

b) Call (module A calls module B) with parameters



c) Definition of call parameters

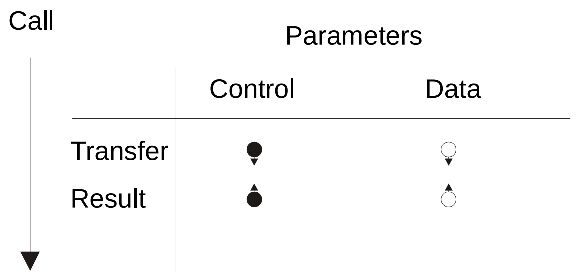


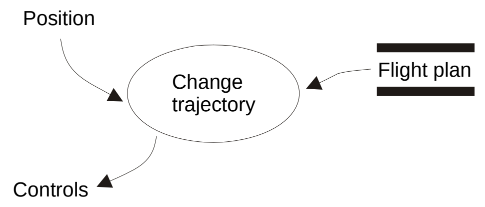
Fig. 4.8: Symbolism for Module Modulisation

4.3.1 Constructing the Implementation Model (IM)

1. General Heuristics

1.1 Minimal Deviations from the Essential Model (EM)

(This is both, the optimal and the minimal model!) In fig. 4.9 satellite control serves as an example of different ways for setting up the processor model.



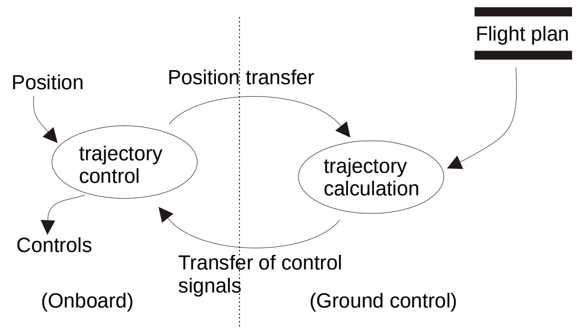


Fig. 4.9: Processor Model of a Satellite Control
a) Essential Model and identical processor model at the same time
b) Modelling of the essential model from a) as a system with two separate processors

The solution a) in fig. 4.9 is simple (identical mapping), but hardly meets the demands of the aimed at realisation: The function "change trajectory" is difficult to interpret with the given data flows. In addition, it cannot be derived from a) where the planned hardware is to be installed. Solution b), however, provides more detailed specifications for both problems at hand: spacial allocation, functions to be performed and resulting data flows can easily be interpreted in the two-processor system described.

1.2 Recognising Idealisations in the Essential Model (EM) and Checking whether the Respective Function in the Implementation Model (IM) is Fulfilling the Task

Examples of critical aspects can be found in the mechanics: tolerances, down time and, in data processing, computer time and storage capacity.

1.3 Downward Levelling - Top-down Approach

Processor: usually hardware (an individual can also take over this part)

Hardware: computer system (CPU, memory, operating system, ...everything the programmer uses)

Processor net: real parallelism

Task: Program part that is controlled by the operating system. Only quasi-parallelism, if at all, is feasible.

Module: Program part that fulfils subtasks of a task and is controlled by the inner logic of the task.

1.4. Available Resources

Considering various resources, the feasibility of the essence is checked (basis: aimed at implementation and its boundary conditions)

1.5 Concentration of Data

The essential model is checked for technically coherent data flows and data transformations (basis: essential model)

Limitations through implementation

2. Implementation

2.1 Real-Time

As an example let's have a look at three processes that have to handle three signals within a minimal distance (fig. 4.10). Measurement is to be simultaneous.

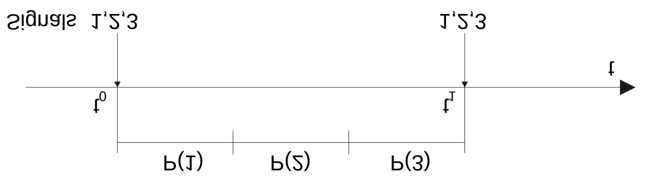


Fig. 4.10: Real-time Test of Three Quasi-Parallel Processes P(1) to P (3) Handling Measurements

The three signals 1 to 3 occur simultaneously at the times t_0 and t_1. Depending on the number of processors that are simultaneously available it has to be checked whether the necessary processing of signals can be achieved within the time period $t_1 - t_0$.

Real-time condition ($T_{P(i)}$: computing time of process 1) is:

- for one processor: $T_{P(1)} + T_{P(2)} + T_{P(3)} \leq t_1 - t_0$

- for three processors: $T_{P(1)} \leq t_1 - t_0$
$T_{P(2)} \leq t_1 - t_0$
$T_{P(3)} \leq t_1 - t_0$.

2.2. Storage Space and Computer Performance

Usually, there is a set gradation for these resources. It has to be checked whether processor, task and module can fulfil their tasks according to a specification. If this is not the case,

• the processor model can be further levelled down or
• back and the processor model can be extended.

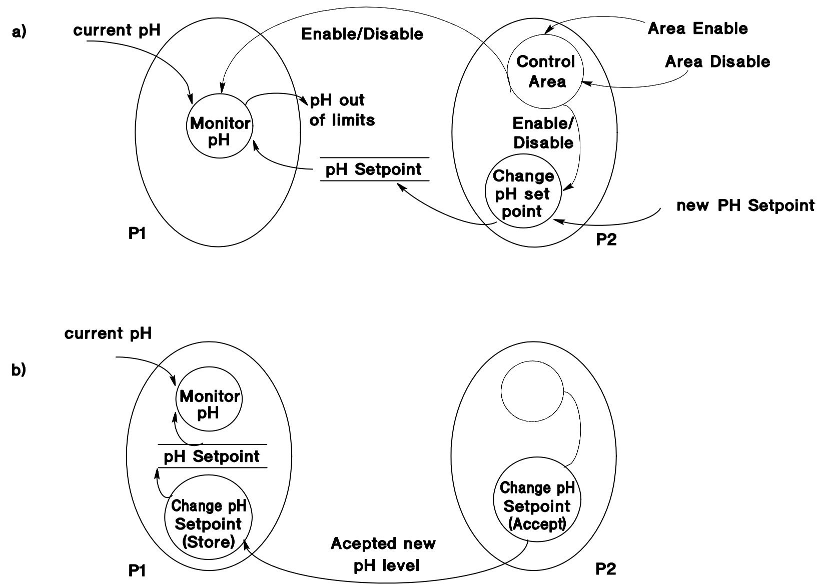
Limitations can be derived from

• data flow
 - discrete: width (bit-parallel), either medium or top rate, storage time, etc.
 continuous: amplitude, noise

• the data model
 - Size and relation of data packages
 (e.g. estimates of records and possibly of other, connected records
 [at least represented by pointer])
 - Maximum size of dynamic structures
 (e.g. Airspace with x planes with y approaches and z periods of stay)

4.3.2 Construction of the Processor Model

Due to the heuristics stated above it is necessary to map the essential model to one or more processors. It is especially important to observe which consequences this allocation has for the resulting communication (cf fig. 4.11).



The ovals P_1 and P_2 denote processors, the other symbols denote assigned tasks

Fig. 4.11: Two possibilities of Task Assignment for a Two-Computer System
a) simple solution: only P_2 has to be synchronized with the input
b) complex solution: division of one event processing between two processors, both processors have to be synchronized

When dividing a transformation among several processors it has to be noted for the mapping of:

Data:
- Data flows are assigned to processors,
- Transformation = global description of the code,
- Storage: assigned to processors

Control:
- The state diagram is duplicated,
- unnecessary actions are deleted respectively,
- resulting free transitions are redirected/deleted.

Task modelling

The model is mapped to the internal organization of each processor. Here the operating system is part of each processor because features for multi-tasking operation can only be derived if processor and operating system are seen as a single system.

Module Level

This is the transition from the task model as a collection of transformations to a structure diagram representing the hierarchical, sequential call mechanism of subroutine, as is eventually necessary for coding.

Within this mechanism any kind of communication between modules is linked to the call.

An alternative communication mechanism can be implemented with shared data (cf fig. 4.12).

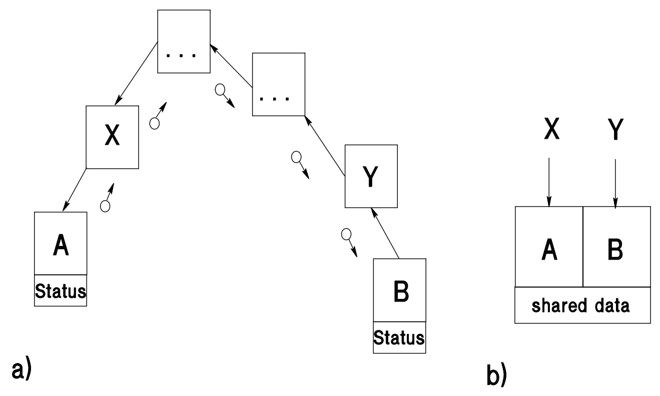


Fig. 4.12: Communication Mechanisms with Shared Data
a) Communication is Only used at Call
b) Definition of Shared Data

Shared data can be defined for several or all modules (global data). (For advantages and disadvantages of both possibilities consult the following chapter.) The reason for introducing shared data is shown in fig. 4.12: Modules A and B need the identical set of data "status". Thus, if B has to make decisions on the basis of the contents of "status", it does not suffice to check the data that have been assigned to B itself. There should be an additional concatenated, thus time-intensive inquiry to A, checking whether there have been any changes in status in the meantime. In turn, this construction leads to several further problems, e.g. in maintenance and reuse.

Fig. 4.12 b) offers an alternative with modules A and B remaining independent, but based on shared data. Most likely, reuse would also tend to be obstructed, but most other problems should be taken care of.

On top of graphic representation modules will be specified, just as with essential modelling. This is done by stating

- pre/post conditions,
- structured language,
- a pseudo-code.

1. Structure

The control transformations from task model as well as essential model form the highest level and data transformations the lowest.

Should it be the case that there are no control transformations, they have to be invented in order to take on the highest control level of (sequential) processing.

No program in any classic programming language can manage without a state diagram for control at the highest level.

The control structure can be realized by

a) a superordinated module (Note: no accumulation of IF/THEN/ELSE structures) (cf fig. 4.13) or
b) Segmentation of the machine table by linking state and action (cf fig. 4.14).

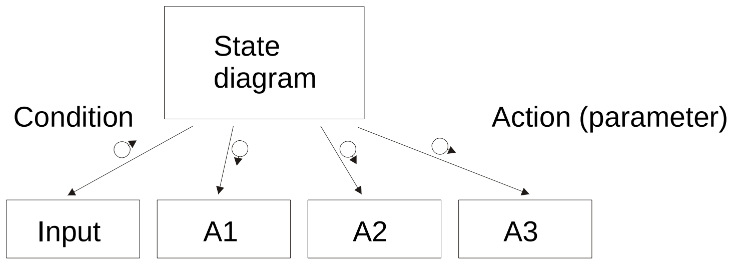


Fig. 4.13: Control of Module A1 to A3 by a Superordinate Machine Dependent on Input

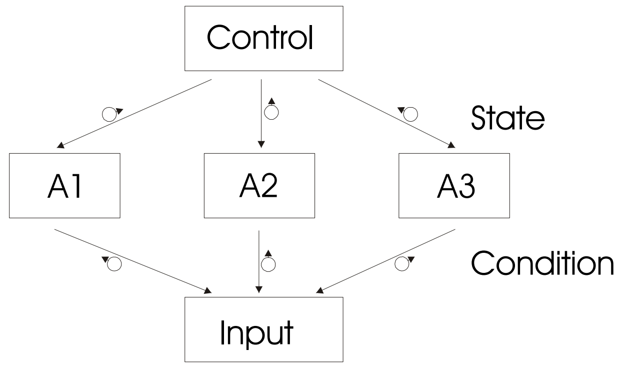
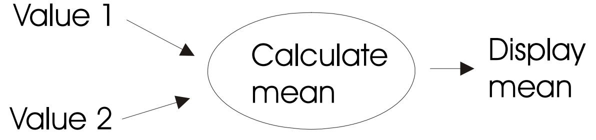


Fig. 4.14: Execution of Modules A1 to A3 Dependent on Input and Alternative Updates of the Machine State (Control)

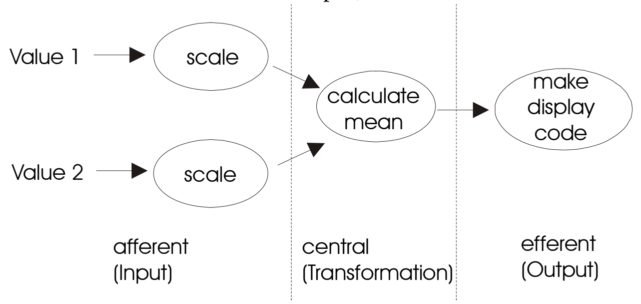
2. Translation of Single Data Transformations of the Essential Model

With regard to implementation, a division of data transformations according to the schema in fig. 4.15 is usually necessary.

a) original data transformation



b) division of data transformation into input, actual transformation and output



c) resulting module schema

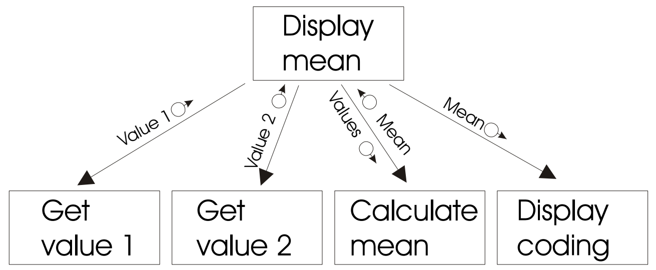


Fig. 4.15: Translation of a Data Transformation into a Module Diagram

The following conventions have been decided upon:

- afferent: control reads output
- efferent: control provides input
- central : control provides/reads input/output

This method is based on the initial isolation of input and output from the original data transformation. These then become parameters for the respective modules. Furthermore, the original transformation is isolated. This makes implementation easier and only makes the creation of reusable modules possible in the first place because all problem-related operations (e.g. scaling) are excluded. Finally, the division directly provides the sequential control mechanism as it has to be executed by the control module, i.e. in the order of calls.

- all afferent modules, then
- the central module, then
- all efferent modules

Here, the control module is named after the task executed. However, parallel input actions are special (cf fig. 4.16).

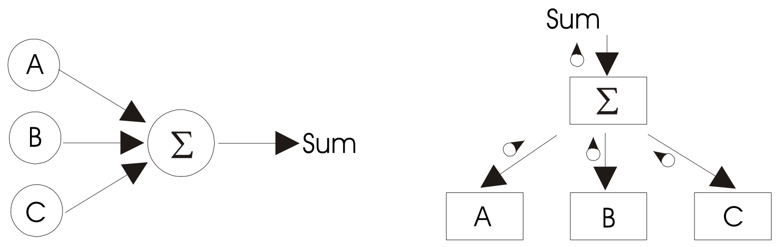


Fig. 4.16: Translation of a Data Transformation into a Corresponding Module Model

The control at the highest level supplies the output data of the system.

Several transformations of the essential model are mapped to one task: Again, one module is formed per transformation controlled by the task, formed with given data- or time-dependence.

Data dependence at the input side is a special case. In this case the transformation analysis has to be preceded by a transaction analysis as shown in fig. 4.17.

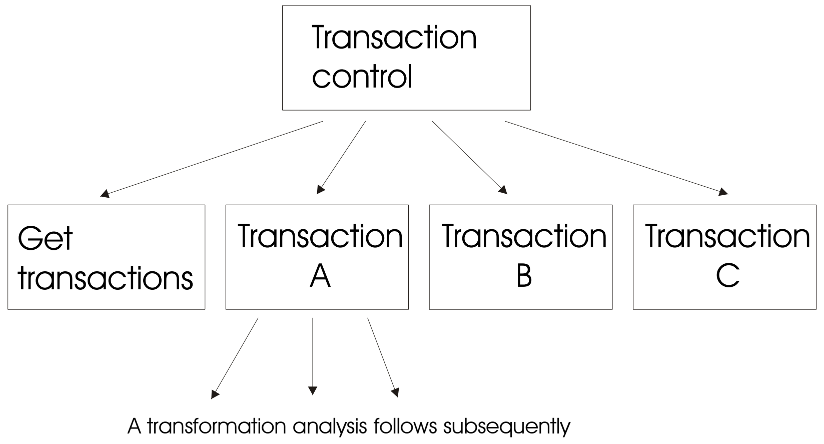


Fig. 4.17: Transaction Analysis

The result of a translation according to the rules explained provides the following:

- a balanced hierarchy,
- the separation of essential and implementation features,
- information hiding = encapsulation of functions in modules and module groups respectively

4.3.3 Defining Goals for Module Design

Design Goal: high module independence
- Maximising <u>interior</u> dependencies of the module

⇨ Minimising exterior dependencies <u>between</u> modules

To further explain different methods for achieving these goals, module features, that have been constructed according to different principles (module strength), and then linking mechanisms among modules are to be described in the following.

1. Module Strength

There are seven categories listed in the following, with 1 being the best and 7 the worst. We will start with the description of the worst category.

Incidental Strength ●●●●●●●

This term applies if

- the total function of the module cannot be defined, or
- it contains several, non-coherent functions.

Such modules are

- not independent of each other and
- their use is worse than not forming modules at all

Logical Strength ●●●●●●

Such a module contains several related functions that are called separately. A structure like this is questionable if the single functions are called via a select code and a fixed list of parameters. Interpreting the call could then lead to false references, since one needs to know <u>all</u> functions and mistaken parameter transfers that can occur.

Moreover, changes in the code of single functions affect the whole module, since selected code and the parameter list represent a rigid linking of functions.

Classic Strength ●●●●●

This is a module with weakly coherent, sequential functions. Initialization and termination of a program are examples of this. The single functions can fulfil completely different tasks. They only share the time of execution.

Procedural Strength ●●●●

Procedural strength is attributed to modules with problem-related, strongly dependent, sequential functions. Extending the definition of classic strength, the problem-related dependence leads to an improved functional demarcation.

Communicative Strength ●●●

Apart from strong problem-related dependence these modules are characterized by strong data dependence. In addition to the functional demarcation to other modules, a high degree of data exchange between the functions of this module is noticeable.

Functional Strength ●●

This definition describes a module with a single function (collective description of a set of subfunctions as one superordinate function). A further subdivision of the function(s) during design is optional, but their outside appearance is critical.

Informational Strength •

The design principle of such a module is information hiding, i.e. the hiding of concepts, data structures, resources and other elements of the module, the knowledge of which is only needed inside the module.

The following features result:

1. Several call entry points,
2. A single function per entry point,
3. functions are connected via a common concept/data structure/resource and
4. there is no dependence on control flows between the individual functions.

It follows that
- this module alone regulates format and organization of the symbol tables used,
- internal changes/optimizations have no influence on the other modules and
- the separation of interface and implementation of each module is possible.

Advantages are
- easy maintenance and optimisation/improvement (no side effects),
- fast prototyping,
- optimal conditions for software testing,
- optimal conditions for teamwork,
- possible reuse and
- extensible modules.

In design the overall goal should thus be to design modules with functional and informational strength only.

2. Linking of Modules

We will now describe the following classification into six categories of module linking, starting once again with the least recommendable category:

Textual Linking ●●●●●●

This kind of linking describes the use of

- direct access (data access via absolute addresses) and/or
- direct control (program flow via jumps instead of calls).

Binding mechanisms and control mechanisms are not used, thus complicating the checking of results as well as almost completely preventing maintenance and changes.

Global Linking ●●●●●

Here the linking is executed via global data structures. Free access of all modules to shared data seems ingenious, but a little caution is called for:

- The code is difficult to read and understand,
- there are possible side effects,
- dependences are created between otherwise disconnected modules,
- when multi-tasking, data integrity is at risk,
- It might not be reusable, as names have been fixed and data structures have been accessed that are unnecessarily complex in their shared use, as seen from the perspective of the individual module,
- data administration and data control are also made more difficult due to the last named reason.

External Linking ●●●●

Again, a global linking is used, but this time it operates with homogeneous shared data. The result is hardly any better than with common global linking.

Control Linking ●●●

This refers to the use of explicit control elements, such as

- function code,
- switch or
- arguments.

Despite the restriction of the linking to a few objects the resulting modules are not strictly independent of each other.

Local Linking ●●

This principle corresponds to global linking. However, in the case of local linking the data are defined as shared only for a subset of modules. Shared data are a case in point. Specific symbols have already been introduced for the modelling of shared data.

Data Linking ●

This refers to

- direct communication between modules or
- via homogeneous arguments (scalars, lists or arrays with fields of equal meaning).

Since data linking only exists for calling (and called) modules it leads to maximum independence of the modules.
In a good design the list of arguments is short (≤ 7) and yet, it supports the documentation well.

The considerations stated above lead to the conclusion that the following concepts are of critical importance:

- Informational strength with the demarcation of internal functionality of modules <u>and</u>
- data linking for the communication among modules.

Both concepts share the following goals:

- the separation of modules, as far as this is possible, and additionally,
- separate definition and implementation of interface and
- implementation of each module.

Various programming languages support these general guidelines for program design, more or less consistently, with corresponding agreements. For example:

C : Interface definition xyz.h
Bound by # include "xyz.h"

C is not a modular language. The approach via the header files described is not sufficient for the expectations stated above.

PASCAL: There are no modules planned in the ISO standard.

Turbo-PASCAL: module = unit
UNIT
INTERFACE
Constants, types, variables
Procedure heading, function heading
IMPLEMENTATION
Implementation of all characteristics of the unit defined above,
if necessary, use of additional variables, procedures, etc.

Co-operation	Program	Unit
Information Exchange	Import	Import and Export
Execution	Yes	No

Modula 2: Division of the module into
- definition module xyz.def
- implementation module xyz.mod

Module	Separately Compilable	Information Exchange
Program Module	Yes	Import
Definition Module	Yes	Import and Export
Implementation Module	Yes	Import
Local Module	No	Import and Export

4.4 Process Administration with an Operating System

The administration of several processes is often the task of the operating system. Accordingly, the function of a scheduler in a multi-tasking operating system will be described in the following (cf fig. 4.18). This representation is a continuation of the real-time-system functions that have already been introduced.

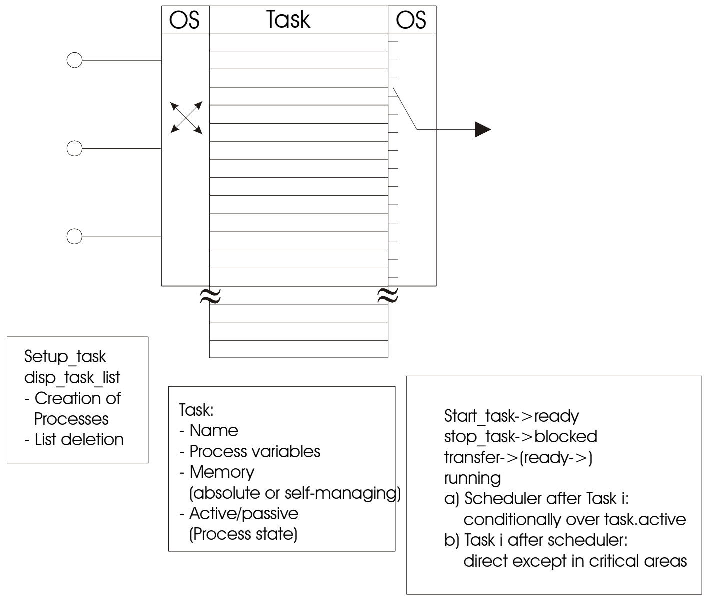

OS
Task
OS
Setup_task
disp_task_list
- Creation of Processes
- List deletion
Task:
- Name
- Process variables
- Memory (absolute or self-managing)
- Active/passive (Process state)
Start_task->ready
stop_task->blocked
transfer->(ready->) running
a) Scheduler after Task i: conditionally over task.active
b) Task i after scheduler: direct except in critical areas

Fig. 4.18 Process Administration by the Operating System with Explicit Multi-Tasking

Under the central control of the operating system the function of the scheduler in fig. 4.18 is to manage a series of processes (tasks) in an appropriate list. The most important components of the list entries are the names of the processes (task i), the assigned variable store (ASi) and its current program counter (PCi). Via the additionally stored state of the processes (state i) the scheduler can control which process is being executed and which processes are due to be executed, possibly according to priority, sequence, etc. The mapping of these virtual procesors onto the working memory is divided according to variable

store and code store. It is then possible to make programs (Pi) also available to several processes without having to produce a copy for every single process, since all variables have been transferred to the private variable store of the processes. This feature of the program code is referred to as "re-entrant", i.e. a process can easily "re-enter" the consistent/sequential processing of its program (solely by setting its program counter), even after it has been interrupted by other processes.

Thus, context change between processes is the changing of different filters through which the processor views its working memory. As far as function is concerned, this is similar to the mechanism of subroutine processing; the critical difference is, however, that only a small number of process parameters have to be saved for re-entry and a corresponding new number has to be loaded, with all variables of the productive process part remaining without explicit cache memory in the working memory.

4.5 Process Administration with Programming Languages

Several programming languages, e.g. ADA, Modula-2 as well as special derivatives of programming languages that do not know processes as such, support similar functions at the level of applied programming.

Especially in Modula 2 the concept of coroutines was introduced for this purpose. The underlying principle is a quasi-parallelity via
- purpose-produced processes
- with explicit transfer of flow control.

Three language constructs have been introduced to realise this concept:

1. Type Declaration:
   The type PROCESS serves for the labelling of procedures as processes.

2. Call:
   The command NEWPROCESS (p=PROC; Wsp: ADDRESS; Size: CARDINAL; VAR new: PROCESS) introduces processes during a program run.

   This elevates the procedure p to the rank of a process (task) with the name PROCESS during operation. At the same time, private memory is allocated to the size *Size* beginning at ADDRESS in order to store
   - command counter,

- registers and
- local variables.

3. Control Flow:
   Changing between processes is explicitly executed via the command TRANSFER (PROCESS 1, PROCESS 2).

This is equivalent to a context change from process 1 to process 2. The agreements No. 1 to 3 are analogous to dynamic data structures: The dynamic creation of processes with algorithmic control flow leads to an active process system. Its function and dynamic behaviour (storage size and running time) only follow the programmer's defaults.
Since the context change via the TRANSFER command replaces the usual call mechanism of a procedure, it is important to make sure that no procedure tries to return to another program with the otherwise common mechanism END= return from subroutine. Accordingly, an activated coroutine should never terminate in itself, the coroutine body is thus an infinite loop. This leads to the following construction:

```
Coroutine
(Process):   - Control block definition
             (-Initialisation)
             - LOOP
             Body
             END (is never reached)
```

There are various applications for the concept of coroutines:

- problems pertaining to the operating system,
- object-oriented programming,
- Complex (recursive) algorithms are replaced by the principle "Divide and rule"
- ...

Communication between Coroutines:

Since the usual call mechanism is prohibited, communication between coroutines (processes) becomes a problem. Communication can take place through
a) global variables or
b) the separation of
   - initialisation part and
   - statement part.

Fig. 4.19 shows an example of how a coroutine can be created and controlled by the mechanisms described above and how directed communication (without the use of global variables) can be constructed.

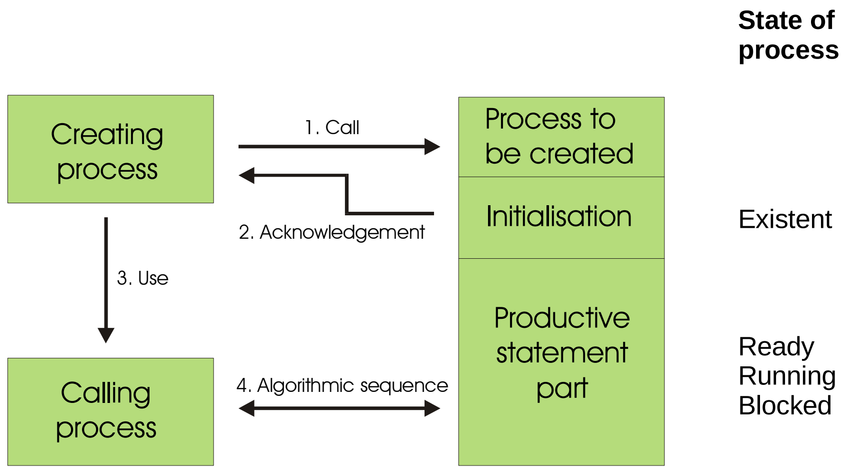


Fig. 4.19: Creation, Call and Directed Communication of a Coroutine

Since the process flow can be interrupted at any time, it becomes difficult to ensure the consistency of data that are shared by several processes. As an example, consider a signal box that is run by a process computer. The process computer's main functions

• writing procedure	(train announcement)
• reading procedure	(display station) and
• procedure route	(signals, points)

communicate with each other

via the global data in
train record
    train number
    train location
    place of origin
    destination

A train approaching the signal box is announced by the data set "train record". Thus, this data set has to be logged into the process computer of the signal box by the function "writing". Depending on the dynamic list of trains announced the signal box now has to set the points for entry and execute the corresponding announcement for the passengers at the display board. Supposing that all of the three kernel functions listed can be interrupted at any time while the train record is being accessed, an infinite number of situations can be imagined that might cause false alarms within the information system and possibly result in serious dangers.

This is why it is important to define critical sections.

Definition: Critical sections are program parts that access shared data.

The consistency problem can be avoided if the following four conditions are fulfilled:

1. At no time are two processes allowed to be in critical zones simultaneously.
2. Assumptions about the relative speed of processes are not acceptable.
3. No process is allowed to obstruct other processes outside its critical sections.
4. No process is allowed to wait arbitrarily long for the entry into a critical zone.

This leads to the conclusions given in fig. 4.20.

	OS Scheduler	Own Scheduler
1.=> mutual exclusion	suspend scheduler	take into account during coding
2.=> results from 1. and undetermined clock frequency	!	!
3.=> Take priorities into account	!	!
4.=> Deadlock	only short suspension of scheduler	No long loops

Fig. 4.20: Consequences of the Conditions for Avoiding the Consistency Problem (Note: There is no Avoidance Strategy)

The following solutions are feasible:

a) Semaphores are often used for external schedulers.

Semaphore: synchronisation variable S

Principle: Boolean variable with s=0: blocked and s=I: free
- Process A marks its data access by S
- Process B reads S and sets S:= 0 (i.e. blocks data for other processes)
- then B checks the read value of S
  - S = 1: Process B continues, since access is allowed
  - S = 0: Process B does not take over control, lets A continue its computing process

b) When using a separate scheduler, i.e. with explicit data transfer, there are two possibilities:

execute no TRANSFER in critical section
- Conditional TRANSFER IF s=I THEN TRANSFER (self- instead of external blocking via s=0) The resulting code can lead to unnecessarily long running times.
- Omitting TRANSFER commands in critical sections. There is a danger of blocking other processes for too long. Thus, it is important that zones are exactly marked in the commentary.

5 User Interfaces

5.1 Significance

Definitions:

• User interface:	Medium of interaction between user and application system
• Dialog:	Interaction between user and application system

One important criterion for evaluating the ergonomics of software is usually whether and to what extent the user interface has been adapted to the needs of the individual. In modern concepts of software development it is not the individual that has to adapt to the computer and the respective programs, but program operation has to be adapted to the needs of the individual. Before developing software, three different criteria have to be considered:

1. It is important to examine the skills and abilities of potential users. This analysis can help to achieve a sensible division of labour between individual and computer. Moreover, complex tasks can be procedurally divided into subtasks.

2. The mode of interaction has to be determined. It has to be decided on input/output mode, (text, graphics,...), input/output devices (keyboard, mouse, screen, ...) and the presentation mode of information (windows, scrollbars,...)

3. Above all, the choice of an appropriate mode of dialog between user and computer is critical. Generally, the three modes of computer-guided dialog, user-guided dialog and hybrid dialog can be distinguished.

Computer-guided dialog offers menus, dialog boxes with input requests, forms and decision making aids with Yes/No choice. This kind of dialog is easy to learn and input errors are unlikely. On the other hand, an extensive, sometimes inefficient dialog is necessary, leading to reduced flexibility.

User-Guided dialog consists of query languages. The user gives commands which gives the user to have a high degree of control over the system and thus enables him/her to be flexible. A disadvantage of user-guided dialog is, however, that it becomes necessary to learn how to use such systems and the likelihood of errors increases.

The hybrid dialog mode offers a sensible way in the middle. It combines the positive features of both methods described above. Alternative input possibilities (dialog boxes with either Yes/No choice or command level) enable beginners as well as experienced users to operate effectively. Another variety are dialogs with alternating initiative.

The various dialog modes are described in the following table:

	6 Dialog type	7 Description
Controlled by computer	Question/Answer	Computer asks questions, user answers them
	Form/Dialog box	The user fills out a form on the screen
	Menu	User selects options, which are arranged into subsets.
more or less user controlled	Function keys	User chooses options by pressing certain keys to which specific functions are assigned.
	Command language	User formulates inputs in a special language
	Query language	A special command language used to query databases.
	Natural language	Information exchange between the user and computer results via natural language.
	Direct manipulation	User interacts with the presented graphical objects and gets immediate responses.
	Multimedia	User interacts with videos and speech signals.

Fig. 5.1: Various Forms of User Interfaces

Question – Answer Dialog
The computer asks questions and the user answers. This design mode is commonly used when the user has little or no experience with using the system. Question order should always be fixed. Since this form of dialog usually applies to an inexperienced user it is only recommendable if the choice of possible answers is limited.

Form Dialog

A mask or a form is provided in which the user can fill in the input fields. This form of dialog is very flexible since the order of data input can be determined by the user.
To make things easier, defaults can also be suggested that can either result from previous sessions or simply from the probability of their occurrence. This form of dialog becomes attractive even for inexperienced users by employing some kind of pointing device, e.g. a mouse.

Menu Dialog
In menu dialog different input options are presented to the user in a list, from which s(he) has to choose. Options are presented either alphanumerically or using symbols on a screen. The overall number of options can be divided into subgroups that the user can pass through in submenus step by step. This arrangement increases clarity and the number of decisions required by the user is decreased.

The extent of input with keyboard, mouse or any kind of contact increases with the number of menu levels. Usually, though, it is very limited. Menu dialog is especially suitable for inexperienced users because complex interaction is broken down into a sequence of small steps. However, as levels increase, this leads to navigation problems: The user may get lost within the menu system or choose an inefficient way to achieve his/her goal. In addition, the breakdown into steps might be too rigid for experienced users and might prove too time-consuming when used frequently.

Menu Types		**Description**
Permanent Menus:	Explicit Menu (explicit)	Separate list of options (horizontal, vertical, radial)
	Implicit Menu (implicit)	Options appear as marked places in graphics or text
Transient Menus:	Pop-up Menu (pop-up)	Options appear in window, their place is arbitrary
	Pull-down Menu (drop-down, pull-down)	Options appear in window, linked to the menu bar
	Slide-out Menu (slide-out)	Further options appear when sliding-out sideways with the mouse

Function Key Dialog
A specific function is assigned to each key; by pushing the key the function is called. A problem is the limited number of keys which can necessitate a multiple allocation of keys. This is confusing for the user since there is no room for labelling the keyboard. An example of function key dialog is the pocket calculator.

Command Language
A special language enables the user to formulate his/her input and requests to the system. Command language is only feasible for experienced users since it presupposes a knowledge of vocabulary and its grammar.

Query Language
Query language is a special kind of command language that enables the user to put queries to a database systems in order to receive the information required. (Examples are SQL (*Sequential Query Language*) or QBE (*Query-by-Example*).

Natural Language Dialog
Natural language dialog would be the ideal form of dialog, since there would not be any need for the user to learn any new language or behaviour. The communication between individual and computer is modelled after the everyday dialog of an individual with its environment. This enables a broad range of users to use the system. However, except for a few specialised research systems (data processing, database queries) this concept cannot be realised at present, since human language is not easy to functionalise (problem: the ambiguity of_concepts and words).

Direct Manipulation
While all forms of dialog described so far can be placed in the world of conversation, direct manipulation is characterised by the metaphor of the "model world". Schneiderman coined this term and defined it in the following way:

Direct manipulation is characterised by:
- Continuous graphic representation of objects and actions of interest.
- The execution of physical actions rather than the formulation of commands with complex syntax.
- The realisation of fast, step-by-step operations. Their consequences for the objects of interest are immediately recognisable.

Direct manipulation is based on the graphic representation of the user's working environment on a graphic screen. Instead of alphanumerical input with a keyboard the user can choose and move visible objects by operating a pointing instrument, e.g. a mouse, as well as initiate actions. A well-known application of direct manipulation is the X-Windows System. Here symbols, such as documents, folders, files and a trash can, reproduce the desk on the screen. With the help of the mouse operations like select, open, transfer, copy, delete, etc. can be executed. Direct manipulation is especially suitable for inexperienced users, since the user is already acquainted with its images from other activities.
Experiments comparing direct manipulation with other forms of dialog present an inconsistent picture. Studies quoted by ZIEGLER and FÄHNRICH (1988) only partially proved the advantages of direct manipulation. KARAT, on the other hand, showed that command languages generally require a much higher execution time with different tasks than direct manipulation.

Multimedia Dialog
Experts regard the development of interactive digital video systems and their use in connection with other media, such as language input and output as multimedia dialog systems, as a revolutionary change in communication between individual and computer.
Often user interfaces are not an explicit element of this task, but it is still customary that approximately 60% of the program code is responsible for the user interface. User interfaces comprehensively define the total system for the user and thus determine the acceptance and marketing of the product.

Example 5.1

Except for state diagrams and other machine representations, all design procedures described so far are hardly suitable for the design of user interfaces.

There are two possibilities to look at user interfaces:
- Representation of user interface to the exterior
- Integration of user interface into the total system

The Seeheim model is often used for a systematic and detailed breakdown of user interfaces.

1. I/O-Interface	1.1 Instrumental Layer	commands, menus, direct manipulations
	1.2 Syntactical Layer	commands, menus, direct manipulations, User objects, feedback
	1.3 Semantic layer,	commands, menus, icons, feedback
	1.4 Information representation	Dialog boxes, Coding, emphasising
2. Dialog-Interface	2.1 Dialog syntax/Dialog style	commands, menus, direct manipulation
	2.2 Course of dialog	2.2.1 Initialising and ending of tasks
		2.2.2 Activation and Reactivation of commands
		2.2.3 Processing/Displaying of working data
		2.2.4 Feedback/Display of presence
	2.3 Backup concept	UNDO, REDO, BREAK, etc.
	2.4 Dialog control help	
	2.5 Error processing	
	2.6 System reaction times	Input delays, Answer time
3. Tool-Interface	3.1 Possibilities of access to system services.	
	3.2 Possibilities of controlling system and program parameters.	
	3.3 Generic functions	
	3.4 Interface functionality	

Fig. 5.2: Seeheim Model of User Interfaces including Relevant Aspects

1. Presentation	1.1 Classes of display (Output and Input)	- Tables and Lists - Dialog boxes - Text objects - graphical objects and pictures
	1.2 Application control:	- Command line-Interpreter - Full page-menus - Pull-down menus - Functions-/control keys
2. Dialog-Control	2.1 Sequential Dialog	- For each point in time there is exactly one system state with a menu of currently availible options. - Input usually results using a keyboard. With additional graphic representation inputs using a mouse are also possible.
	2.2 Asynchronous Dialog	- direct manipulation - Multi-thread-Dialogs (at each point in time there are several different task paths) - event controlled (object orientated communication) - parallel Dialog (like Multi-thread, except that it is possible to choose several options at the same time)
3. Applications-interface	3.1 Database access	- Objects, Relations
	3.2 Procedural access	- Access control by interposing available library procedures.
	3.3 Direct access to internal application data structures	

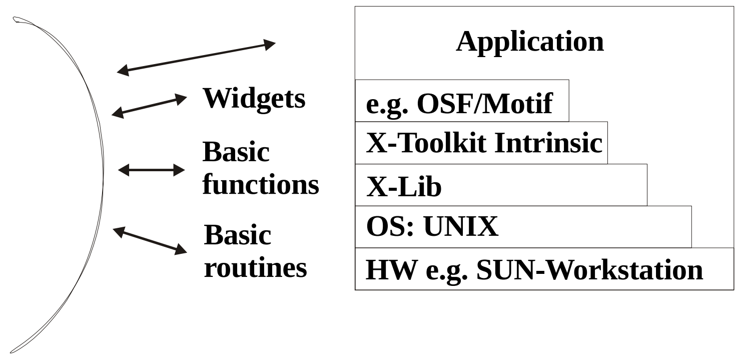
Fig. 5.3: Implementation Forms

7.1 Aids for User Interface Design

7.1.1 Toolkits

Toolkits present a choice of functions and presentations in the form of routines. They are offered in the form of libraries, e.g. MS-Windows, X-Window.

Fig. 5.4 gives the example of a user interface for a computer system working with the operating system UNIX, just as various, hierarchically layered libraries are used in practice. The arrangement of the layered bars, representing the individual libraries in fig. 5.4, expresses that the application can access any of the libraries, if necessary. Libraries, however, can only access the library directly arranged underneath.



Widgets	⇒	Windows Gadgets
Basic functions	⇒	basic elements:bars, lines, etc.
Basic routines	⇒	operating system, hardware
X	⇒	portable
Toolkit	⇒	tool box
Intrinsic	⇒	essential
Lib	⇒	library

Fig. 5.4: Constructing a User Interface by Falling Back on Various Different Libraries

Toolkits are the first step towards the separation of user interface and application; they do not allow a complete separation, however, and do not offer any design aids.

7.1.2 Development Systems

Development systems, on the other hand, allows the complete separation of user interface and application. They support abstract specification and capture presentation and dialog design likewise.

7.2 Design Methods

7.2.1 Context-free Languages

The Backus-Naur form allows the formal description of functions and data access, for example.

Symbol	Description	To be read as
=	**Equivalence**	**Is composed of**
+	**Sequence, concatenation**	**together with**
[ ]	**Selection of 2 or more possibilities**	**one of....**
|	**Seperator for these**	
{}	**Repeat of the embraced components**	**Sequence of**
()	**optional: 0- or 1 times**	

Fig. 5.5: The Backus-Naur form[WAR 91]

- "The equal symbol represents a data item to be described.
  aircraft position = latitude + longitude + height

  The data element on the left is composed of the items on the right. The composition on the right side of the equal symbol may be arbitrarily complex, it can consist of names and the three other symbols.

- The plus symbol is used to denote concatenation. The symbol is read as "together with"; it is neither connected with mathematical addition nor the Boolean "or" symbol. No order is implied.

- Selection is described by listing each item that may be chosen between square brackets, separated by vertical bars:
  Gender = [Male | Female]
  i.e. "gender" is either "male" or "female" (exclusive or).

- An arbitrary number of items, separated by vertical bars, can be specified.

- Iteration is marked by braces:
  Logic state = {channel level }
  A "logic state" is defined as a collection of all "line levels". It is often advisable to specify limits of iteration, thus:
  Logic state = 4 {channel level} 10
  This expresses that there must be at least 4, but no more than 10 "channel levels". By default, if there is no lower bound, the assumed minimum is 0, and if the upper bound is missing, there is no upper limit.

- The DEMARCO notation also uses parentheses to denote the optional presence of a data item within a composite. This notation is redundant, since the following expressions are equivalent:
  X = Y + (Z)
  X = Y + 0 { Z } 1

"[WAR91]

Example 5.3 ***Example 5.4***

YACC, the UNIX parser generator, was derived from that. YACC can be used for describing a command line user interface. (**y**et **a**nother **c**ompiler **c**ompiler)

7.2.2 State Diagrams

Here actions are assigned to state transitions or machine states.

At present, the most important tool is USE (User Software Engineering) with state diagrams according to WASSERMAN. This is a common state diagram which has been extended by:
+

- Specifications in the textual part,
- Assignment of actions by symbols upon transition (CA: call action) No.: reference to the textual part and
- Subdiagrams for transitions as well as states that are treated as subroutines.

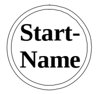
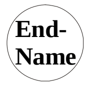
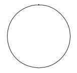
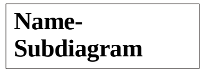
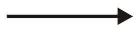
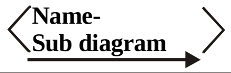
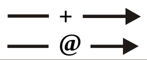
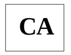
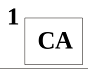
	**Starting node**
	**Ending node**
	**State**
	**State for which a subdiagram will be defined.**
	**Transition**
	**Subdiagrams which refine a transition**
	**Transition always results automatically when pressing any key**
	**An action which is assigned to a transition**
	**Assigned whole number as a reference to the textual specifications**

Fig. 5.6: Extension Symbols for State Diagrams according to WASSERMANN (cf [NEM 90])

"A state diagram consists of one or more nodes that are connected via arrows. In every diagram there is a starting node and an ending node respectively that have been clearly marked. The nodes correspond to states with which output messages can be connected. These outputs are specified in the textual part. For every class of possible user input resulting from a state, there is a corresponding transition. Transitions are marked by arrows between nodes. Input required for a transition, is written next to the arrows. An action can be connected with a transition. These actions are marked by small squares with the label "CA" (call action) and integers are assigned as reference to the textual specification.
To reduce the complexity of diagrams lower-level subdiagrams are provided as a structuring technique. These subdiagrams work in the same way as subroutines. Transitions in the current diagram are delayed until the called subdiagram has been processed. Subdiagrams can be defined for states as well as transitions. For states, rectangles are substituted for circles containing the names of the corresponding subdiagrams (subconversation). For subdiagrams that represent a

lower-level of a transition the name of the subdiagram is written in pointed brackets (< >) next to the arrow." [NEM90]

Example 5.5

7.2.3 Event-controlled Representation

This is the most powerful of all forms of representation. Events (input events) reach the system via keyboard, mouse or general computer interfaces. An event handler (special administration process) recognises and assigns events to objects.

7.3 Design of Optical Characteristics

A total input of approximately 10^9 bit/s
- with the consciousness only grasping about 100 bit/s -

continuously affects the system "human being".

Due to this enormous discrepancy the individual has to select and filter characteristic features from the flood of information pouring in. This is why a user surface should be designed ergonomically, in the sense that all important information should be presented to the individual as compressed and easily understandable as possible.

Thus, criteria are:

a) perceptibility of information
b) coding
c) organisation of location and time

⇒ A) includes the brightness of the screen on the one hand and minimal character size on the other caused by the human visual angle of 18°.

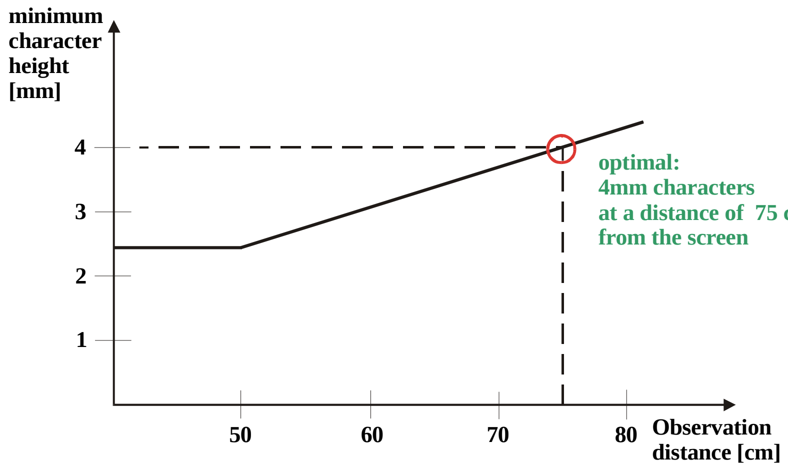


Fig. 5.7: Minimal Character Size Depending on the Observation Distance

⇒ B) implies that forms and range of coding have to be selected. The following forms are available:

- Shape,
- Colour,
- Location,
- Time,

and the following code range is feasible:

- Colour: 3.3 bit -> 8 to 10 colours,
- Brightness: 6 grades,
- Size: 5 grades.

The range values of different forms of coding stated last result from empirical research and the general context as shown in fig. 5.8, an individual can only distinguish information coded with up to about 3.3 bit.

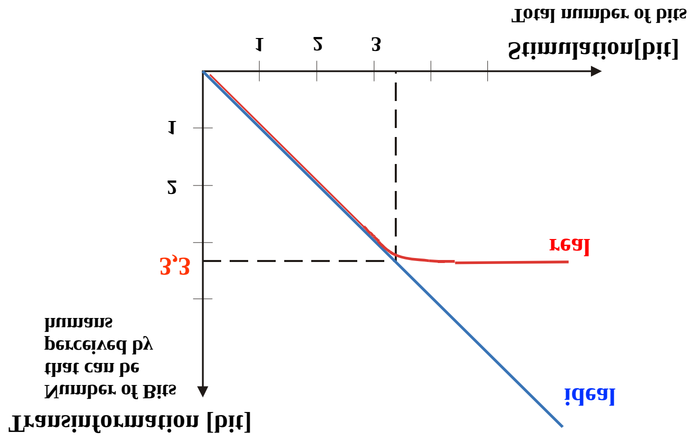


Fig. 5.8: Range of Perception of a Human Being [Miller 1956]

⇒ As for c), design features can additionally be used for organizing information, such as

- Closeness
- Symmetry
- Similarity

The following technical framework results from these considerations:

7.3.1 The Classic Screen

The structure of a character-oriented screen consists of 80 characters/line and 25 lines/screen. This equals 2,000 characters/screen.

The characters themselves are represented by a matrix of 8 to 10 pixels. This results in a screen resolution of

- 80 * 8 = 640 pixels horizontally,
- 81 * 10 = 250 pixels vertically
- therefore: 160,000 pixels per screen.

Visual storage thus contains 160,000 : 8 = 20kbyte when representing pixels in black-and-white.

The pixel frequency for the structure of a screen is 14MHz and correspondingly, pulse time is t_P = 71,42 ns.

Data throughput for realising a pixel frequency of 14Mbit/s amounts to 1,75Mbyte/s and thus the cycle time for a byte-organised memory amounts to
t_Z = 571 ns.

The character set is usually either fixed or switchable, as a whole or related to language-specific characters and additional characters. It is customary to use the ASCII character set (national character selection); special block graph symbols have been introduced for the IBM-PC which help to draw up simple graphics on the basis of characters.

7.3.2 <u>Colour Graphics</u>

• Consider the example of a workstation. For a 19" screen the following applies:

- Resolution is about 10^6 points. Additionally, 8 bit each are needed per pixel to control brightness for the colours R(ed), G(reen) and B(lue). The colour of the points results from an additive mixture.
- This leads to a memory requirement of $2^8 * 2^8 * 2^8 * 10^6$ (approximately 160 Mbit) = 20 Mbyte with the possibility of selecting one of 16 million shades of colour for all of the 10^6 screen points.

To save costs the memory requirement is reduced by limiting the amount of colours allowed per screen to 256. This subset is stored individually for each screen in a colour chart (or colour palette).

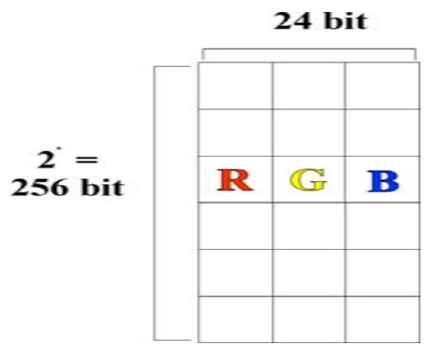


Fig. 5.9: Structure of a Screen Colour Chart

This leads to a space requirement of:
Screen Palette
$10^6 * 8 \; + \; 256 * 24 \; \cong \; 8$ Mbit = 1 Mbyte

The addressing of such screens is generally pixel-oriented, that is why form and size of the character set are easily adjustable by the screen driver software.

7.3.3 <u>Input Systems</u>

Generally, there are two different input devices, namely

- Keyboard for alphanumerical character input:
  - Command input/selection
  - Cursor movements (block by block)

- Mouse for pointing on pixels:
  - Character input point by point
  - Selection of objects, interpretable as
    - data or
    - commands.

Historically speaking, the keyboard is the first input device. Now keyboard and mouse are simultaneously available. Since icons as object representations are easy to grasp and differentiate, graphic user interfaces have been introduced to facilitate the communication between computer and individual. Additionally, input syntax errors can be ruled out from the start. After an initial euphoria, the attitude has now become more sceptical. That is also why both input forms are likely to co-exist in the future.

6 Software Testing

6.1 <u>Introduction</u>

"Testing is the process of executing a program with the intention of detecting errors."

From MYERS definition [MYE 91] it can be concluded that a test has been successful if an error has been found. It would be analogous to conclude that a visit to the doctor's has been successful if an illness has been diagnosed.

In contrast to this, the following definitions are wrong:
A test is supposed to show that
a) there are no errors,
b) the program is running correctly or
c) one can trust that the program does what it is supposed to.

These definitions are widespread, but wrong due to the following reasons:

to a) There is no definition of the term "error" and the number of possible errors has to be estimated as "at least infinite".

to b) and c) These statements lead to two conclusions:
1. The program does not do what it is supposed to do <u>and</u>
2. the program does what it is **not** supposed to do!

The first conclusions demands a complete description of functions for the program. As explained in earlier chapters, such a description is impossible. The second conclusion cannot be verified, since all cases <u>not</u> defined in conclusion No. 1 would have to be tested against. Even when supposing that this is feasible, it would be an infinite set.

6.2 <u>Principles</u>

For every test a complete definition of values expected (valid as well as invalid) is needed, on the basis of a function description that is as complete as possible.

- The development of test cases is expensive, so test cases should be documented and saved.
- It is psychologically not advisable for a programmer to test his/her own program.
- Testing is creative and intellectually challenging.
- A test should never be planned on the tacit assumption that a program will not contain any errors.

In practice, errors are often cumulative and not at all evenly distributed over the whole code. Thus the more errors are found in one module the likelier it is to detect even more errors in exactly the same module.

6.3 <u>Methods</u>

6.3.1 <u>Inspection, Review, Walkthrough</u>

These terms refer to the discussion of the source code within the team (i.e. programmer, test expert, guest). This method presupposes an easily readable programming language, a sufficient amount of commentary in the source code and a design documentation. Logical errors that cannot be discovered by a machine will be found with this procedure. Company-internal inquiries at IBM showed that 80% of all errors had already been discovered during inspection, i.e. without actually running the program.

This procedure includes
- discussion with non-authors,
- use of error check lists,
- record of results and especially
- error removal, instead of avoidance or symptom removal.

This kind of error check list depends on the experience of the test team and, partly, on the type of program. Fig. 6.1 shows an example of such an error check list.

a)

Data references	Calculations
1. Are variables accessed that have not been initialised?	1. Are calculations conducted with non-arrhythmic variables?
2. Are all indexes in their boundaries?	2. Are there calculations with different data types?
3. Are non integer indexes used?	3. Are calculations conducted with variables of different length?
4. Has the pointer still got a valid address assigned to it? (dangling reference)	4. Is the size of a target variable smaller than that of the assigned value?
5. Are the attributes of a redefinition correct?	5. Are there any over- or underflows in temporary variables used during calculation of the final result?
6. Do set and structure attributes match?	6. Division by zero?
7. Are the addresses of the bit chains OK? Are bit chain attributes passed?	7. Are there any inaccuracies caused by the representation of fractions using floating point variables?
8. Are attributes correct for "based"-memory requirements?	8. Does the value of a variable contain a value in a sensible range?
9. Do the data structures match in different procedures?	9. Are the priorities of certain operators correctly understood?
10. Are string boundaries exceeded?	10.Does the division of a number by an integer lead to a correct result?
11. Are there any "off by one" errors?	

b)

Data declaration	Comparisons
1. Are all variables declared correctly?	1. Are comparisons conducted between two inconsistent types?
2. Were the standard attributes understood correctly?	2. Are variables of different data types compared?
3. Are arrays and strings initialised correctly?	3. Are the comparison operators correct?
4. Are the correct lengths, types and memory classes used?	4. Are the Boolean expressions correct?
5. Does the initialisation fit to the type of memory class used?	5. Are comparisons and Boolean algebra mixed?
6. Are there variables with similar names?	6. Are there comparisons with fractions of base 2?
	7. Has the priority of operators been correctly understood?
	8. Has the compiler representation of Boolean expressions been understood correctly?

c)

Control flow	Input/ Output
1. Are all jump possibilities taken into account in statements where multiple decisions are made?	1. Are the file attributes correct?
2. Is every loop completed?	2. Is the OPEN-Statement correct?
3. Is the program closed correctly?	3. Do the I/O statements fit the format specifications?
4. Is a loop not executed due to the input condition?	4. Does the buffer size fit to the clock sizes?
5. Are ways that are implemented to avoid loops correct?	5. Is the file opened before use?
6. Are "off by one" errors possible during an iteration?	6. Are the end-of-file conditions respected?
7. Do the DO/END keywords belong together?	7. Are I/O errors handled?
8. Are there incomplete decisions?	8. Are there any text errors in the output information?

d)

Interfaces	Other checks
1. Do the number of input parameters match the number of arguments?	1. Are there any unused variables in the cross reference list?
2. Do parameter- and argument attributes match?	2. Does the attribute list do what one expects?
3. Do the units of the parameters and arguments match?	3. Are there any warning or informative messages?
4. Does the number of arguments passed to the called module match the number of parameters?	4. Is the input data checked for validility?
5. Do the attributes of the arguments passed to the called module match the attributes of the parameters?	5. Error functions?
6. Do the units of the arguments passed to the called module match the units of the parameters?	
7. Are number, attribute and sequence of the arguments correct for "built-in" functions?	
8. Are there references to parameters which are not associated with the current entry point?	
9. Are arguments changed that should have been inputs only?	
10.Is the definition of global variables consistent across all modules?	
11.Are constants passes as arguments?	

Fig. 6.1: Error Check List
a) Data Reference - Calculations
b) Data Declaration - Comparison
c) Control Flow- Input/Output
d) Interfaces - Other Tests

6.3.2. Test Case Design

The main problem is to determine which subset out of all imaginable test cases offers the highest possibility of detecting as many errors as possible. For carrying out the test a program run on the computer is now necessary.

Black Box Test

This test is data-driven, i.e. it is developed without any knowledge of the implementation.

Test data is derived from the specification. That is why this test also captures logical errors.

The task of test development is the definition of all input patterns, not only for specified functions, but also for all other cases. Especially the latter is impossible, as the example of the compiler shows: For testing its correctness all imaginable programs would have to be applied in the respective language.

White Box Test

Based on the logic (control flow) of program implementation

- every command is executed once (weak criterion) or
  every path is executed once (better criterion, since it contains every command <u>and</u> the control).

This procedure usually misses logical errors.

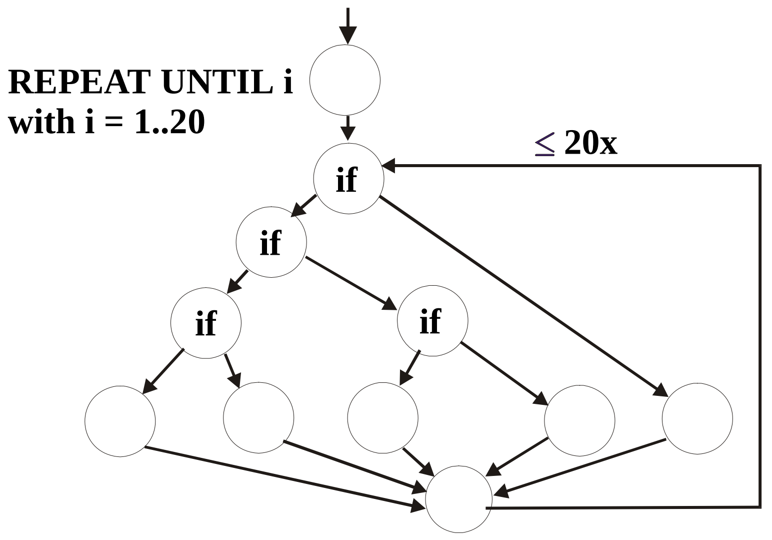


Fig. 6.2: Construction of all Paths in the White Box Test

As shown in fig. 6.2, there are five paths in the loop body. The number of variants in a complete loops with i repetitions is thus 5^i. Should i then be variable from 1 to 20, the number of paths is:

$$\sum_{i=1}^{20} 5^i = 5^{20} + 5^{19} + \ldots + 5^1 \approx 10^{14} = 14 \text{ quadrillions.}$$

Since both methods are clearly limited in their efficiency, but discover different error types, a combination of both methods is often used in practice.

Aids for Developing Test Cases

Fig. 6.3 lists the usual aids for the development of test cases.

Black Box	**White Box**
Equivalence Classes	Coverage of all Commands
Threshold Value Analysis	Decisions
Cause-Effect Graph	Conditions
Error Guessing	Multiple Conditions

Fig. 6.3: Aids for Developing Test Cases

Equivalence Classes

Since it is impossible to conduct a complete test, a test should have the following two characteristics:

- it reduces other test cases and
- it covers a large number of test cases .

Accordingly, input data is first divided into equivalence classes and then test cases are selected from these classes.

Input condition	Valid Equivalence Class	Invalid Equivalence Class
Number of Fields		
Field Types		
String		
Value Range		
.		

Fig. 6.4: Chart for the Formation of Equivalence Classes

Threshold Value Analysis

Here equivalence classes are specialised into
- threshold value of input - equivalence classes and
- threshold value of output - equivalence classes.

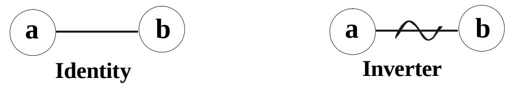
e.g. defined range of values: -1,0...+1,0
=>test cases : -1,001; -1,0; -0,999;
0,999; 1,0; 1,001

As far as input data is concerned, these test cases are easy to set up (provided that the specification of the program is good). However, it is often difficult to determine the input for threshold values of output.

Cause-Effect Graph

With the help of an appropriate representation it is possible to design test cases systematically from input combinations, which also cover logical errors. A formal language was developed as an appropriate tool, which transforms specifications into a combinatory graph. This language has the following components:

1. Breakdown into separate functions,
2. -Cause: Input condition
   -Effect: resulting output condition or system transformations
   - Numbering of all phases within the specification
3. Mapping of the semantics in a graph (commented, if necessary) in the sense of Boolean algebra,
4. Transforming the graph into a decision table => table - test case



With the following limitations

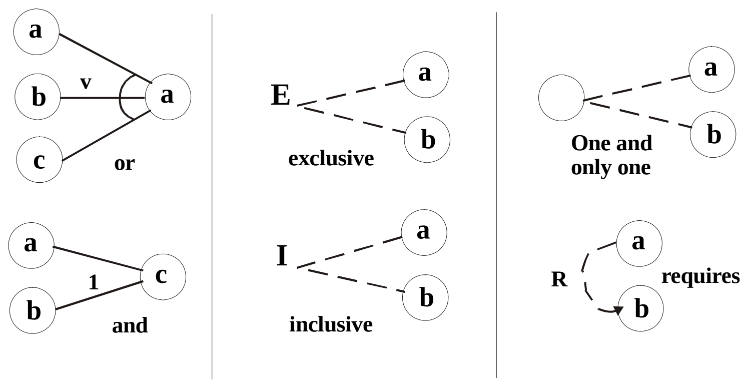


Fig. 6.5: Basic Elements for the Construction of Cause-Effect Graphs

6.4 Comparing Top-down and Bottom-up Testing

As with design and implementation of the program, there is a bottom-up and a top-down approach to testing.

Top-down Testing

Advantages	**Disadvantages**
1. Has advantages when errors tend to occur nearer the top.	1. Sub modules have to be created
2. Test cases become easier to implement if the I/O functions are already implemented	2. Sub modules are often more complex than they at first appeared to be.
3. An early program skeleton allows presentations and boosts moral.	3. Test cases are difficult to create before the I/O routines are integrated.
	4. Creating test cases can be very difficult, in some cases even impossible.
	5. Observing the test-output is more difficult.
	6. May lead the tester to believe that design and test can be done at the same time.
	7. May be a reason for delaying complete tests of certain modules

Bottom-up Testing

1. Has advantages if errors occur in lower level modules.	1. Driver modules have to be created.
2. Test cases are easier to conceive.	2. The program as a whole does not exist until the last module has been linked in.
3. Checking of the test results is easier.	

Fig. 6.6: Comparison between Top-down and Bottom-up Testing

7 Examples

7.1 <u>Chapter 2</u>

7.1.1 <u>Example 2.1 (Christmas Presents)</u>

This example describes a decision making process that could very well become relevant for a company shortly before Christmas: "Customers of the company are presented with complementary gifts. The value of the gifts depends on the volume of the orders the respective customers have placed throughout the year." Here the company has to ensure that declared teetotallers are presented with a book instead of a bottle of wine.

Christmas presents	**Rules**					
	1	**2**	**3**	**4**	**5**	**ELSE**
0<OV<=5.000	**Y**	**N**	**N**	**N**	**N**	
5.000<OV<=20.000	**N**	**Y**	**Y**	**N**	**N**	
20.000<OV	**N**	**N**	**N**	**Y**	**Y**	
Teetotaller	-	**Y**	**N**	**Y**	**N**	
Christmas card	**X**	**X**	**X**	**X**	**X**	
Wine present			**X**		**X**	
Book present		**X**		**X**		
Personal telephone call				**X**	**X**	
illogical						**X**

OV: Order value

Fig 2.A: Decision Table for Choosing Christmas Presents for Customers dependent on Company Order Volume = OV [NEM 90]

The logic of a rule can now be read from the table, clockwise, starting from the upper left-hand corner.
The following applies to rule No. 1, for example:

Y	If order volume lies between DM 0 and DM 5.000
N	and does not lie between DM 5.000 and DM 20.000
N	and does not amount to more than DM 20.000
-	and the customer is either a teetotaller or not then
X	send him a Christmas card. [NEM90]

Here the ELSE column either serves as
– Error abort (illogical entry) or
– as reference to further tables.

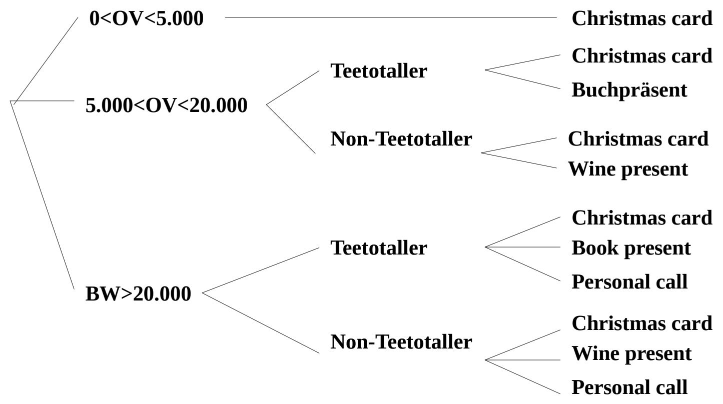


Fig. 2.B: Decision Tree of the Table Christmas Presents at Fig. 2.A [NEM 90]

"The starting point of the decision sequence is the root of the tree which first of all branches to the three possible intervals for order volume. A branch of the tree then represents the respective series of decisions, the order of which is determined by the tree structure. The leaves of the tree are the actions that are carried out for each respective combination of decisions, from top to bottom. In this tree diagram conditions are first checked and then actions are executed." [NEM90]

7.1.2 Example 2.2 (Combination Lock)

X	State of safe
0 0 7	open
All other combinations	closed

Fig. 2.C: Decision Table for a Combination Lock

In this example the combination lock has a three-digit code and the code "007" for the system state "open".

7.1.3 Example 2.3 (Telephone Call)

Example 3 describes a telephone call:

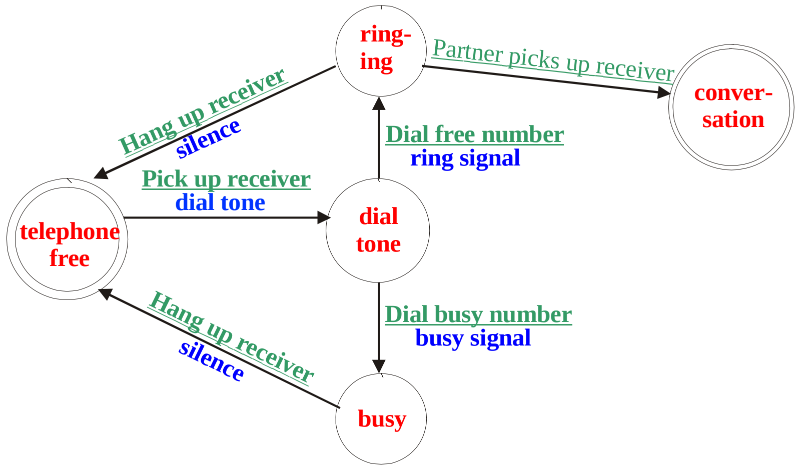


Fig. 2.D: State Diagram for a Telephone Call [NEM 90]

- The phone is free, the receiver is picked up and there is a dial tone.

- Next, there are three possibilities:
  1. The receiver is replaced again. This results in silence and the telephone is free again.

  2. The number has been dialled is busy and there is an busy signal.
  3. The number that has been dialled is free and there is a ring signal.

- After it has been established that the number dialled is engaged, the receiver is replaced. This results in silence and the telephone is free again.

- After it has been established that the number dialled is free, there are two possibilities again:
  1. The desired party does not answer the phone and the receiver is replaced again. This results in silence and the telephone is free again.
  2. The desired party answers the phone, a connection is established and a conversation can take place.

- The machine shows two final states represented by double circles:
  - initial state: phone free
  - final state: conversation

Event / State	Pick up receiver	Hang up receiver	Dial free number	Dial busy number	Partner picks up receiver
telephone free	dial tone				
	dial tone				
dial tone		silence	ring signal	busy signal	
		telephone free	ringing	busy	
ringing		busy			-
		telephone free			conver-sation
busy		silence			
		telephone free			
conver-sation					

Fig. 2.E: State matrix of a telephone conversation [NEM 90]

Fig. 2.E: State Matrix Telephone Call [NEM 90]

For each matrix element there is a next state and an action: If, for instance, there is the dialling tone and this is followed by the event that the number being dialled is free, then the next state will be a ringing at the desired party's phone and the action will be a ringing tone.

Current state	Event	Action	Next state
telephone free	User picks up receiver	dial tone	dial tone
dial tone	User hangs up receiver	silence	telephone free
	User dials free number	ring signal	ringing
	User dials busy number	busy signal	busy
ringing	User hangs up	silence	telephone free
	Partner picks up receiver	-	conversation
busy	User hangs up	silence	telephone free
conver-sation	-	-	-

Fig. 2.F: State Table Telephone Call [NEM 90]

"The state matrix has four columns: In column 1 current states are given. These are then transferred to next states in column 4 via the events that are listed in column 2. Upon state transition the actions listed in column 3 are carried out." [NEM90]
As far as the example at hand is concerned, the current state is the dialling tone. This is followed by the event of dialling a free number which in turn leads to the next state of a ringing at the desired party's end via the action of a call signal.

Example 2.4 (Payroll)

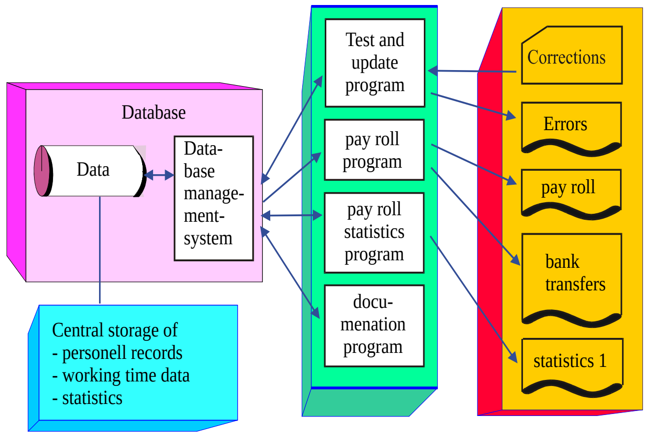


Fig. 2.G: Payroll Program with Database [DU93]

"At a conceptual level it has to be determined for the database of the payroll program that there are the data sets "employee" that contain a personal identification number, a name and a salary, etc. The personal identification number is always an integer, the name always contains a maximum of 20 characters, etc. Furthermore, it has to be recorded that there are dependencies between the data sets "employee", such as "is subordinate of".
At an internal level it has to be determined in which order the fields "personal identification number", "name", "salary", etc. of the data set "employee" are saved, how long they are and how the data is encoded. In addition, the internal schema contains information about file organisation as well as data access mechanisms.
At an external level it has to be defined that only salary and department are needed as personal data (statistics are to remain anonymous). The statistics program also presupposes that the salary is being provided by the database as an integer, i.e. it has previously been rounded off to full units of currency." [DU93]

7.1.4 Example 2.5 (Company Structure)

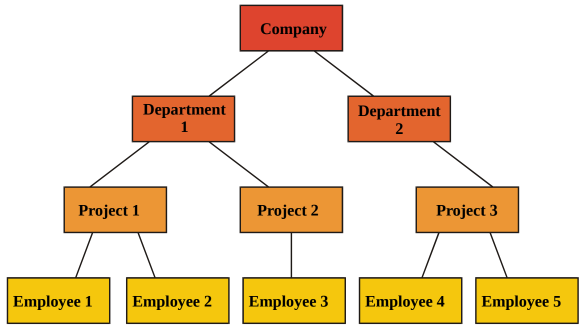


Fig. 2.H: Structure of a Company Using a Hierarchical Data Model [DU93]

The company's hierarchical structure can be easily recognised in fig. 2.H. However, cases in which one employee is involved in several projects or several departments take part in one common project cannot be shown.

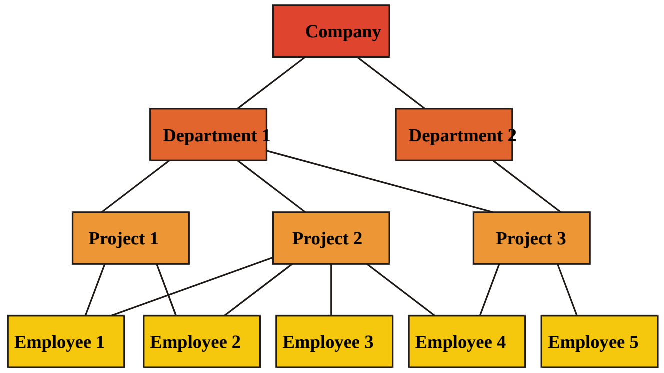


Fig. 2.I: Structure of a Company Using a Network Data Model [DU93]

This problem has been solved in the network model (fig. 2.I) by networking, but now data is no longer easily accessible.

Relation "Employee"		
Personell-number	**Name**	**Salary**
1427	Meier	3.217,33
8219	Schmidt	1.425,87
2624	Müller	2.438,21
..	..	..
..	..	..

Relation "Employee in department"	
Personelll-number	**Department**
1427	3 - 1
8219	2 - 2
2624	3 - 1
..	..
..	..

Fig. 2.K: Structure of a Company Using a Relational Data Model [DU93]

Relations can easily been recognised from the tables. The tuples can be read like actual sentences: "Employee Meyer has the personal identification number 1427, receives a salary of 3,217.33 currency units and works in the departments 3 – 1."

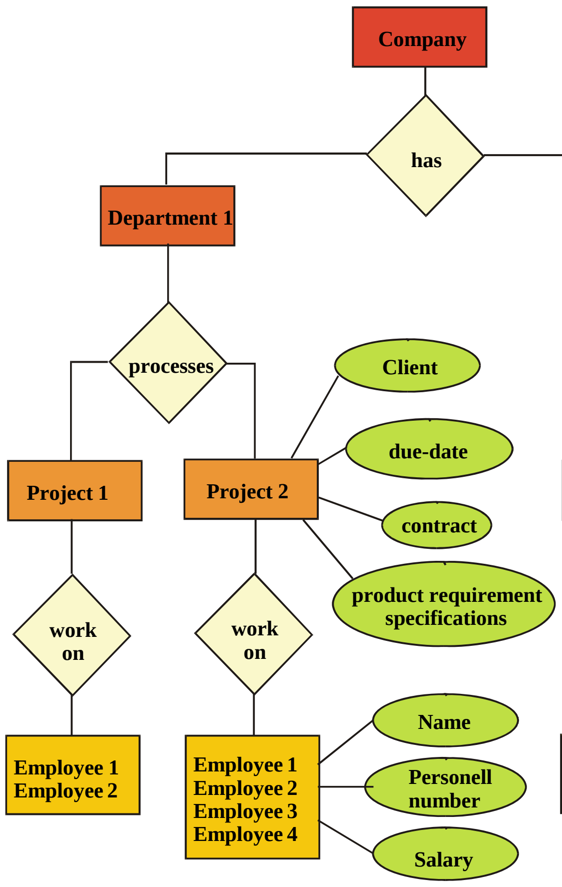


Fig. 2.L: Structure of a Company Using an Entity Relationship Model

In fig. 2.L the hierarchy is easy to recognise as well. However, an employee who is involved in several projects has to be listed several times. His/her data that were listed in the tables above (cf fig. 2.K), are listed once again in this entity relationship model.

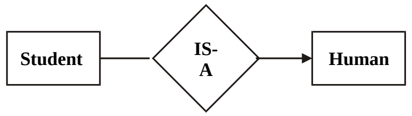
7.1.5 <u>Example 2.6 (Point P)</u>

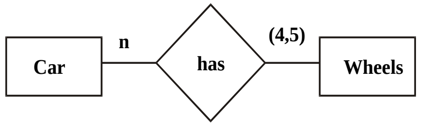
The following data can clearly determine the position of a point P in space:

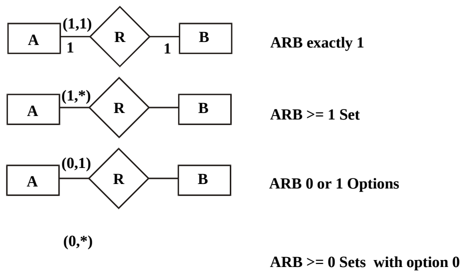
```
ENTITY POINT P;
  x : real;
  y : real;
  z : real;
END_ENTITY;
```

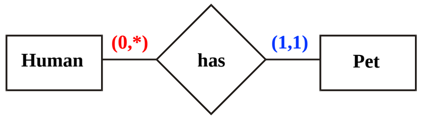
Fig. 2.M: Position of a Point P in a Three-Dimensional Space

7.1.6 <u>Example 2.7 (Entity Relationship Models)</u>









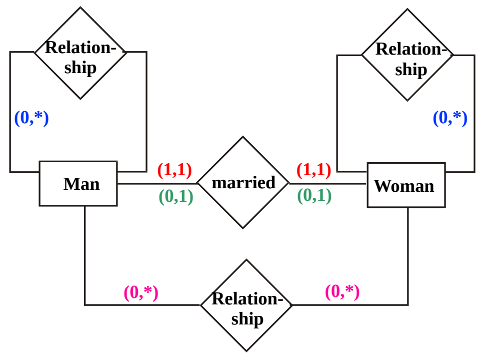


Fig. 2.N: Some Examples of Simple Entity Relationship Models
a) Example of an IS-A relationship
b) Application of the Frequently Used Relationship "has" for the Description of Common Cars and their Number of Wheels (Including Spare)
c) Several Variations in the Valency of a Relationship R
c) Applied Example *Pet*
d) Applied Example *Family relations*

a) Every student is a human being, since the arrow is pointing in the direction of *human being*. The opposite is impossible, since not every human being is also a student.

b) A car has four (including spare: five) wheels. Wheels are parts of cars.

c) Exactly <u>one A</u> has a relationship R with exactly <u>one B</u>.
One A has a relationship R with <u>one or more Bs.</u>
One A has a relationship R with <u>no B or one B</u>.
One A has a relationship R with <u>no, one ore more Bs.</u>

d) (0,*) : An individual has no, one or more pets.
(1,1) : A pet only belongs to one individual (only one person to relate to, even in families).

e) (1,1) : A description of all <u>monogamously married</u> individuals.
A man is married to a woman.
A woman is married to a man.
(0,1) : Description of <u>all</u> human beings
One man is not married at all or married to one woman.
One woman is not married at all or married to one man.

(0,∗) : Man has no relationship or a relationship with one to X women.
Woman has no relationship or a relationship with one to X men.
(0,∗) : Woman has no relationship or a relationship with one to X women.
Man has no relationship or a relationship with one to X men.

7.1.7 <u>Example 2.8 (Library)</u>

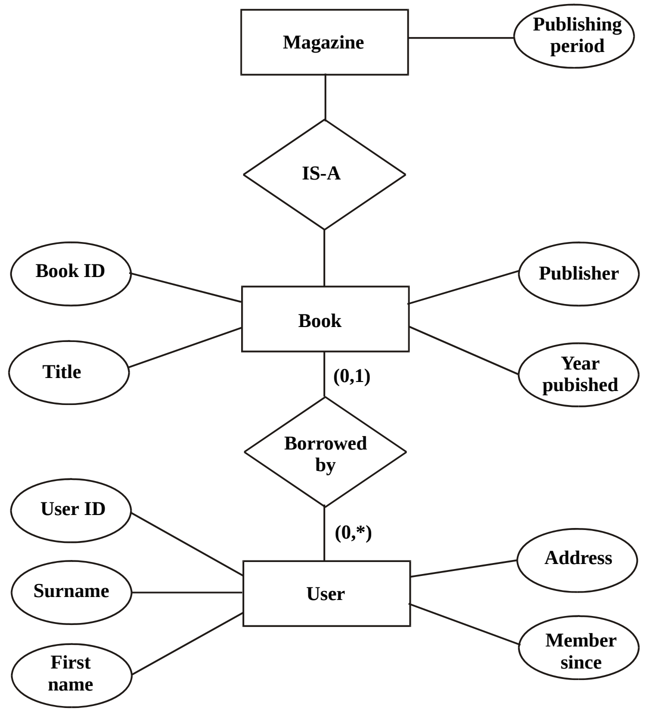


Fig. 2.O: Extract of a Library Data Model
a) Graphic Representation of the ERD (Entity Relationship Diagram)

a) There are two databases in the library, one for books and magazines and one for user data.

A book is labelled with a book ID, title, publisher, and year of publication. Magazines are also listed as books and is additionally labelled with publication period.

A library user is listed in the respective database with last name, first name, address and date of admission and receives a user (registration) number.

The relationship between these two databases can be characterised by the relation "borrowed by".

- A registered user can borrow none, one or more books.
- A book, however, cannot be taken out at all (it is still on the shelf) or it can be taken out by only one user.

```
ENTITY SET:
    Name : BOOK
Attribute:
     BookID      : -1- CHAR(7)
     Title       : -1- CHAR(35)
     Publisher   : -1- CHAR(35)
     YearPub     : -1- INTEGER(4)
KEY: BookID

ENTITY SET:
    Name : MAGAZINE
    IS-A : BOOK
Attribute:
     PubPeriod : -1- CHAR(15)
KEY: BookID

ENTITY SET:
     Name : USER
Attribute :
     UserID      : -1- INTEGER
     Surname     : -1- CHAR(20)
     Firstname   : -1- CHAR(20)
     Address     : -1- CHAR(50)
     Membersince : -1- DATE
KEY: UserID

RELATIONSHIP SET:
       NAME : BORROWED_BY
  ENTITY-SETS : BOOKS, USERS
        TYPE : n:1
```

Fig. 2.O: Extract of a Library Data Model
b) Textual Representation of the ERD

b) With entity sets the figures in front of the data type indicate how many values of this attribute are allowed for the description of an entity set. This figure has no counterpart in the ERD. Afterwards, the data type used is declared and in brackets the size of the corresponding data field is determined. Textual representation highlights how obvious the transition from the model as a result of problem analysis to the formulation of the first constructive design is.

7.2 Chapter 3

7.2.1 Example 3.1 (Electronic Components)

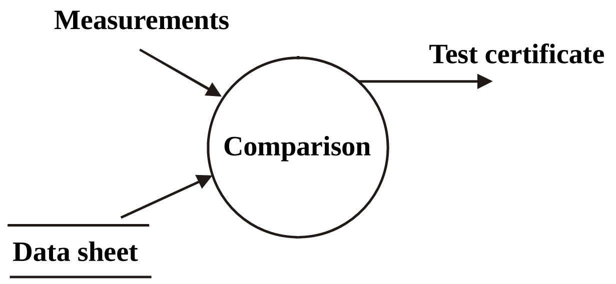


Fig. 3.A: Simple Data Flow Diagram – Testing Electronic Components

Compare incoming measurements with the data stored in the database "data sheet" and draw up a test certificate.

7.2.2 Example 3.2 (Selling Airline Tickets)

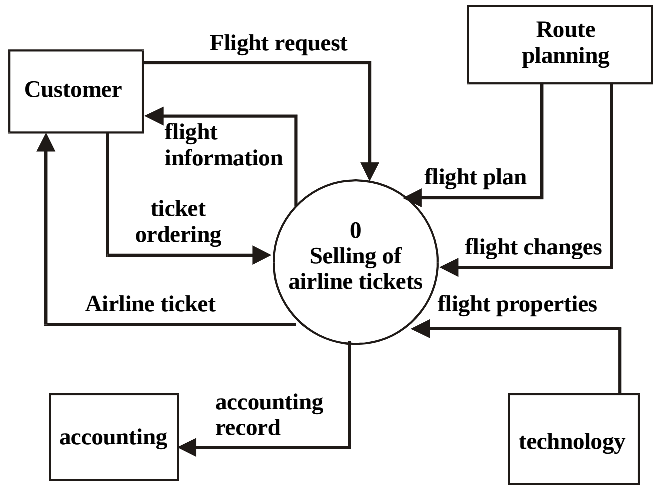


Fig. 3.B: More Complex Data Flow Diagram – Selling Airline Tickets [RAA93]

7.2.3 Example 3.3 (Order Entry)

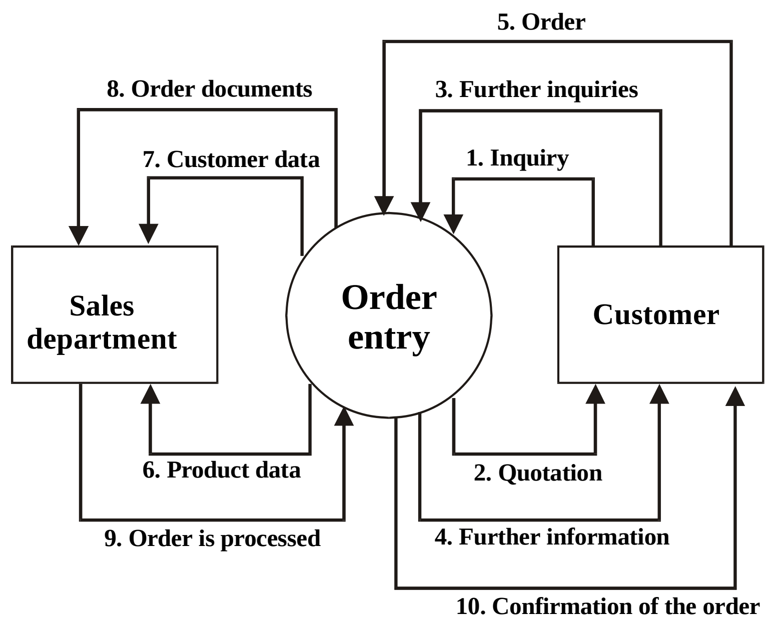


Fig. 3.C: Context Diagram - Order Entry System (cf [NEM90])

1. A customer inquires at a specific company about the manufacturing of a specific product.

2. He then receives an offer,

3. which leads to further inquiries on his part.

4. In turn, the company gives further information about the requested product.

5. The customer orders the product.

6./7./8. The company provides the sales department with product data, customer data as well as order documents and

9. the sales department handles the order.

10. Simultaneously, the customer receives a confirmation of the order.

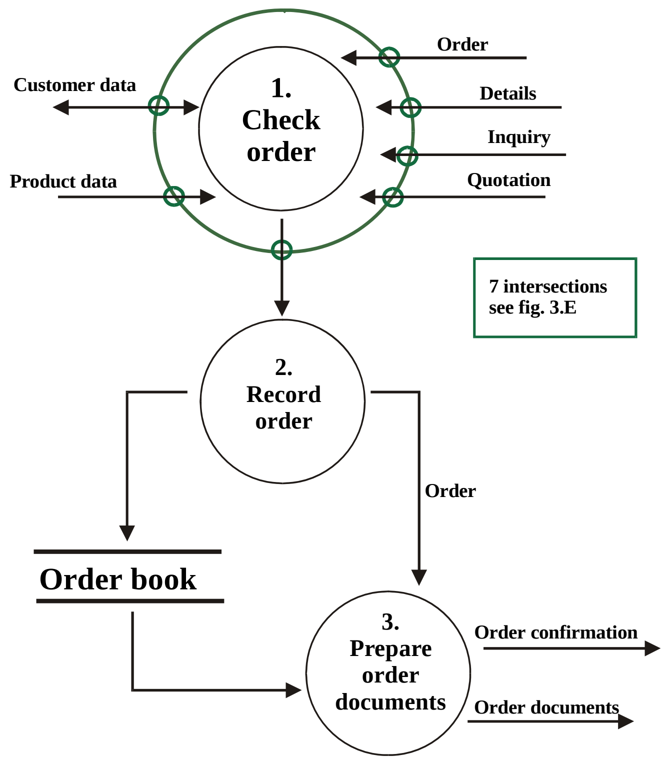


Fig. 3.D: First Transformation of the Context Diagram "Order Entry" – Level 0 (cf [NEM 90])

The process of entering orders is divided into three further processes:
1. Check order
2. Record order
3. Prepare order documents

As can be seen, all data flows from the context diagram reappear (cf colour-coding). Additionally, a file "order book" has been included, that only responds to the processes in this diagram and is not available externally.

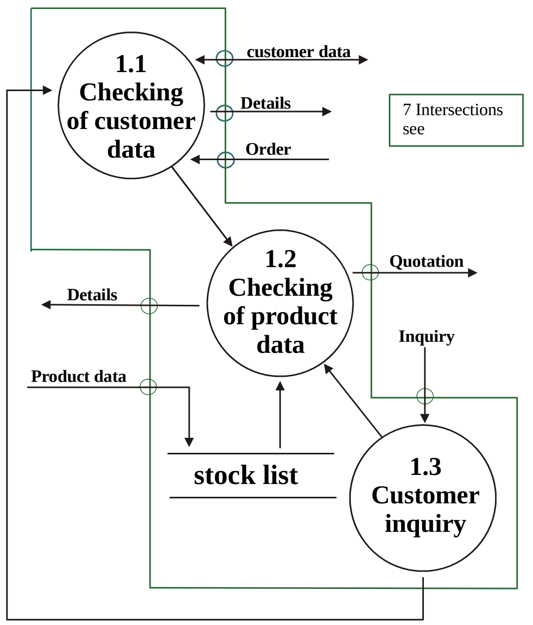


Fig. 3.E: Downward Levelling of Transformation 1 Checking Orders (fig. 3.D) – Level 1 (cf [NEM 90])

Since even the processes at level 0 are still quite complex, they have to be further subdivided.
The first process ⇒ Check Orders is further levelled downward into
1.1 Check customer data
1.2 Check product data
1.3 Customer request

Again, all data flows are preserved (cf seven interfaces).

7.2.4 <u>Example 3.4 (Roll Regulation)</u>

The speed of a roll for wire production has to be regulated. The speed depends on the electric current that is used by the roll motor. Regulation results as follows: actual and nominal speed are compared and so the respective manipulated variable (motor current) is established. Actual speed is established by a sensor; nominal speed is determined by a user. The regulation process is initiated by a clock pulse (regulator clock pulse) every 10 ms.
A maximum value for motor current has also been determined by the user. If this maximum value should be exceeded,
- the user receives a message,
- the motor is switched off (This should only be done if there is no more wire being processed.) and
- the system returns to its initial state.
Sensors inform the system whether the roll is currently processing any wire.

The following transformation schema has been designed for this regulation:

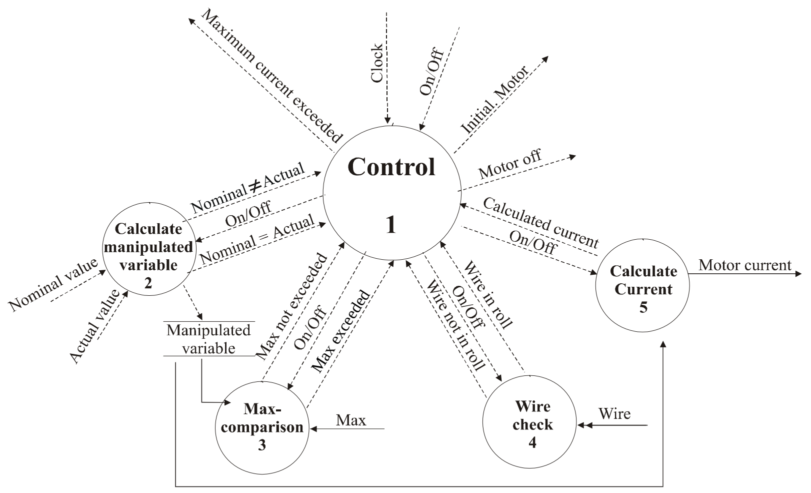


Fig. 3.F: Transformation Schema for Speed Regulation of a Roll for Wire Production

Please note that:
- The temporal sequence of the data transformations results from numbering.
- The value Max represents the maximum value that has been determined by the user.

- Should it be that nominal = actual, one has to wait for the following clock pulse.
- The motor should be switched on as well as initialised at the beginning of the process. If the event "off" is transferred to the control transformation (control), any currently active data transformation is switched off and the system returns to its initial state.

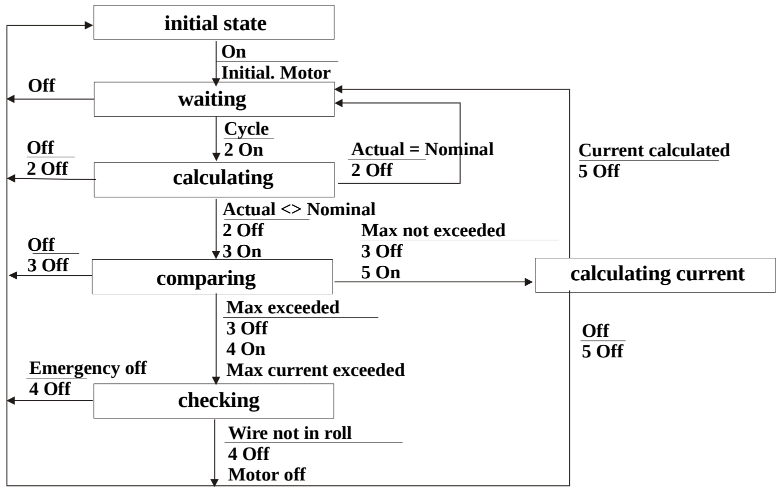


Fig. 3.G: State Transition Diagram of Fig. 3.F

2:	Pre:	Nominal Value
		Actual Value
	Post:	Manipulated Variable
3:	Pre:	Max
		Manipulated Variable
	Post:	----
4:	Pre:	Wire
	Post:	----
5:	Pre:	Manipulated Variable
	Post:	Motor Current

Bild 3.F: Pre-Post-Conditions for Fig. 3.F

7.3 Chapter 5

7.3.1 Example 5.1 (Car)

Consider an everyday example: the car, or rather driving a car. Again, there is a user interface between the vehicle and the driver. On the one hand, correct operation as well as economical use of the system "car" have to be guaranteed; on the other hand, trust in the system "car" has to be boosted by the perceptible quality of external car features, such as doorhandle, the door sound and dashboard. The user's working environment, i.e. the serviceability of the car by the driver, has been specifically designed according to principles of work sciences and ergonomics: adjustable seats, easy access to all instruments, etc. One of the main tasks of the interface is, however, to provide a simplified mental model of the actual functionality of a car: Accelerator + clutch + gearstick serve as a model of the actual, more complex motor technology, since the actual sequence of functions is not known and not comprehensible to most drivers. (cf [RAA93])

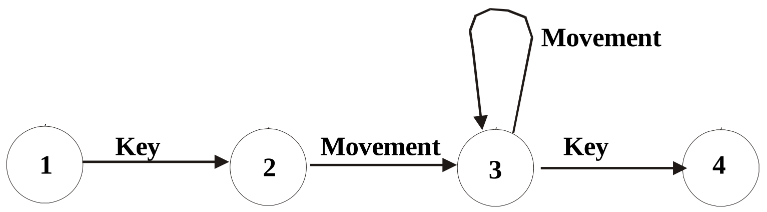
7.3.2 Example 5.3 (Elastic Band Function)

The elastic line function of a graphics editor serves as a design example in three variations. The starting point (1) is fixed, then a variable line is drawn to a continuously changing point (2) until the point is fixed with a mouse click (point 3).

Line	-> Key d1 End_point
End_point	-> Movement d2 End_point | Key d3
d1	-> {store first point}
d2	-> {draw line to current position}
d3	-> {store second point}

Fig. 5.C: Context-Free Language – Elastic Band Function [NEM 90]

Representation with context-free grammar extended by actions that have been executed by the program (enclosed by { }).



2: store start point
3: draw line to current position
4: save finish point

Fig. 5.D: State Diagram – Elastic Line Function [NEM 90]

Representation as state diagram with states being assigned to program actions. The assignment of actions to state transitions, i.e. arrows, is also feasible.

```
EVENT HANDLER line;
TOKEN
     key Key;    /* Def. of Tokens */
     movement Movement;
VAR
     int state;
     point start, finish;
EVENT Key DO {
     IF state == 0 THEN
        start = current position;
        state = 1;
     ELSE
        finish = current position;
        deactivate (self);
     ENDIF;
};
EVENT Movement DO {
     IF state == 1 THEN
        draw line from start to current position;
     ENDIF
};
INIT
     state = 0
END EVENT HANDLER line;
```

Fig. 5.E: Event-controlled Representation – Elastic Band Function [NEM 90]

7.3.3 <u>Example 5.4 (Checking Orders)</u>

Let's return to example 3.3 of chapter 3. The data flow diagram of fig. 3.E walks through the downward level 1 – orders. According to the Backus-Naur form data specification could be as follows:

a) CUSTOMERDATA= {CUSTOMER}

b) CUSTOMER = ID# + NAME + ADDRESS
c) ADDRESS = POSTCODE + CITY + COUNTRY (*only for internatonal customers*)

d) FURTHERINQUIRIES =
[INQUIRY-CUSTOMERDATA
INQUIRY-PRODUCTDATA
INQUIRY-CUSTOMERDATA+
INQUIRY-PRODUCTDATA]

e) INQUIRY-RESULT = INFORMATION

Fig. 5.F: Data Specification – Checking Orders [NEM 90]

a) The customer file consists of a sequence of data elements which belong to the same type (here CUSTOMER).
b) Composition follows as a sequence with three components.
c) A further address specification takes place, again as a sequence. An additional comment has been added that only in the case of foreign customers the country should also be part of the address.
d) The specification shows the composition of further inquiries: they are inquiries about either customer data or product data or both. Naturally, the structure of the different inquiries would have to be further specified.
e) It is often useful to work with aliases that can be defined by using the equivalence symbol. (cf [NEM90])

7.3.4 Example 5.5 (Selection of Functions)

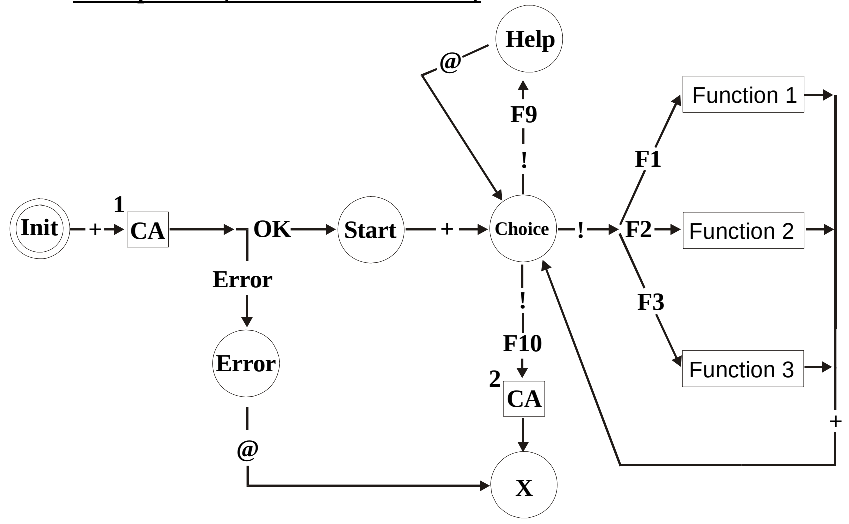


Fig. 5.G: State Diagram for the Selection of Functions [NEM 90]

"A state transition with the action 1 is automatically executed, starting from the node **Init**, the input node of the diagram. Automatic execution is represented by the symbol +. The further course of the state transition then depends on the result of the action that has been described in the textual part. In the case of an error it changes to the **error** state and, after pressing any key (marked @ at the arrow), it leaves the diagram via the output node X. In the case of OK it branches to the **selection** node via the **start** node. The screen output assigned to the respective nodes is specified in the textual part. State transitions can take place by pressing the corresponding function key (! And F1, F2 and F3 respectively at the arrows). The rectangles labelled **Function1, Funktion2** and **Function3** have their own diagrams for detailed specification of the state. With a associated state transition the processing of the current diagram is interrupted until the lower-level diagram has been processed. After processing one of the three diagrams it automatically returns to the **selection** node. If the F10 key is pressed, action 2 is executed and it returns to the state of the input/starting node X. " [NEM90]

```
ACTIONS
1 call loaddb (exmpld)
2 call unload (exmpld)

DIAGRAM: Startmenu            ENTRY: Init            EXIT: X
DATABASE: exmpld                DIRECTORY: \example\example

TAB t_1 20                    MESSAGE Header
                                    cs, r2, c_, ′EXAMPLE SYSTEM‘
NODE Init                     NODE Choise
                                    tomark_A, ce

NODE Start                    r+5, t_1, ′F1    - Function 1‘,
     Header, mark_A          r+2, t_1, ′F2    - Function 2‘,
                              r+2, t_1, ′F3    - Function 3‘,
NODE X                        r+2, t_1, ′F9    - Help‘,
     cs                       r+2, t_1, ′F10  - Quit‘,

NODE Error
    cs, r10, c_, bell, rv, ′Database cannot be opened‘, sv
    r$, c_, ′To continue = press any key‘
NODE Help
     cs, r$-4, c0, ′Keys F1 – F3 start functions 1, 2, 3‘,
     r$-3, c0, ′For more information on a command ‘,
     r$-2, c0, ′please press the corresponding function key and then F9‘,
     r$, c0, rv, ′Please press any key to return to the menu!‘
```

Fig. 5.H: Extract of the Corresponding Text Description - Selection of Functions [NEM90]

8 Bibliography

8.1 Further Reading on the Subject Matter:

[ACH 91]
Achatzi, G.: *Praxis der strukturierten Analyse*.
Hanser, München, 1991

[BEI 95]
Beims, H.D.:*Praktisches Software Engineering. Vorgehen - Methoden - Werkzeuge*.
Hanser, München, 1995

[BRA 90]
Brauer J.: *Datenhaltung in VLSI-Entwurfssystemen. Leitfäden der angewandten Informatik*.
Teubner, Stuttgart, 1990

[CHE 76]
Chen P.: *The Entity-Relationship Model. Torwards a Unified View of Data*.
ACM Transaction on Database Systems, Vol. 1, 3 / 1976

[DeMA 78]
DeMarco, T.: *Structured Analysis and System Specification*.
NewYork, Yourdon Press, 1978

[DIJ 89]
Dijkstra, E.W., Feijen, W.H.J.: *A Methode of Programming*.
Addison-Wesley Publ. Co., Bonn, 1989

[DIN 90]
Dinyarian F.: *EDIF Test view*, Proceedings of the 4. European EDIF Forum,
Warrington, England, 1990

[DU]
Duden Informatik.
Dudenverlag, Mannheim, 1993

[GIB 94]
Gibbs, W.W.: *Software: chronisch mangelhaft*.
Spektrum der Wissensschaft, 1994, S. 56 - 63

[LEXIN]
Lexikon der Informatik und Datenverarbeitung.
Oldenburg, München, 1991

[LEXKO]
Lexikon der Informatik und Kommunikationtechnik.
VDI, Düsseldorf, 1990

[MAN 91]
Manche, A.: *CASE - Ende der Softwarekrise*.
c´t, 1991 Teil 1: H.8, S.54-60, Teil 2: H.9, S.212-220

[MYE 91]
Myers, G.J.: *The Art of Software Testing*.
John Wiley & Sons, New York, 1991

[MYE 95]
Myers G.J.: *Methodisches Testen von Programmen.*
Reihe Datenverarbeitung, VIII. Oldenburg, München, 1995

[NEM 90]
Schönthaler, F., Nemeth, T.; *Software - Entwicklungswerkzeuge: Methodische Grundlagen*. Teubner, Stuttgart, 1990

[RAA 93]
Raasch, J. *Systementwicklung mit strukturierten Methoden*.
Hanser, München, 1993

[SCH 90]
Schenk D.: *EXPRESS Language Reference Manual*.
Mc Donell Aircraft Company, 1990, ISO TC 184 / SC4 / WG1

[SCL 83]
Schlageter G., Stucky W.: *Datenbanksysteme: Konzepte und Modelle*.
Teubner, Stuttgart, 1983

[STE 82]
Steinmetz, G.: *Was ist ein Pflichtenheft?*
Elektronische Rechenanlagen, H.5, 1982, S.225-229

[VER 1]
Verhelst B.: *Test Specification Format.*
EVEREST report No.: PHI 187 CRTN

[VER 2]
Verhelst B., Retz H.: *TSF procedural interface, TSF viewer*.
EVEREST report No.: PHI 200 CRTN

[WAR 91]
Ward, P.T., Mellor, S.J.: *Strukturierte Systemanalyse von Echtzeit- Systemen.*
Hanser, München, 1991

[WIR 86]
Wirth, N.: *Algorithmen und Datenstrukturen mit Modula-2.*
Teubner, Stuttgart, 1986

[YOU 90]
Young D.: *The X- Window System. Programming and Application with Xt.*
OSF/ Motif Edition, Prentice hall, Englewood Cliffs, New Jersey, 1990

[ZEH 87]
Zehnder C.A.: *Informationssysteme und Datenbanken*.
Teubner, Stuttgart, 1987

[ZIM 91]
Zimmermann, J., Schwiderski, L.: *Intuition mit Methode*.
c `t, 1991, H. 8, S. 70-78

8.2 Further Reading on the Theoretical Background:

Fähnrich, K.-P.: *Software- Ergonomie*. (State of the Art 5)
Oldenburg, München, 1987

Hunger, A., Retz, H: *Datenmodelle und deren graphische Repräsentation. Kognitive Ansätze zum Ordnen und Darstellen von Wissen*.
INDEKS Verlag, Frankfurt a. M., 1992, S.121-130

Miller, G.A.: *The MagicalNumber Seven, Plus or Minus Two:*
Some Limits on our Capacity for Processing Information.
The Psychological Review, Vol, Nr. 2., 1956

Miller, G.A.: *The MagicalNumber Seven after Fifteen Years.*
Studies in Long-Term Memory, ed. by A. Kennedy. Wiley, 1975

Special Issue on Real- Time- Systems.
Proceedings of the IEEE, Vol. 82, 1994, No. 1, S. 1-192

Wirth, N.: *Systematisches Programmieren*.
Teubner Studienbücher, 17, Stuttgart, 1972

Woodman, M.: *Yourdon dataflow diagrams:*
a tool for disciplined requirements analyses.
Information and Software Technology, H.9, 1988, S. 515-533

8.3 Further Reading on Other Areas of Application:

End, W., Gotthardt, H., Winkelmann, R.: *Softwareentwicklung.*
Siemens AG, München, 1984

Heinz, E.: *Paralleles Programmieren mit Modula-2.*
Vieweg, Braunschweig, 1991

Schäfer, S.: *Objektorientierte Entwurfsmethoden*.
Addison-Wesley, Bonn, 1994

Wallmüller, E.: *Qualitätssicherung in der Praxis*.
Hanser, München, 1990