

# Languages, Methods, and Tools for Software Specification

Invited paper<sup>1</sup>

Jochen Ludewig  
Swiss Federal Institute of Technology Zurich,  
Computer Science Departement<sup>2</sup>

Specification systems consist of methods, languages, and tools; the languages may be more or less formal. In this paper, the general ideas of "semi-formal" specification systems are presented, and some examples are shown.

## 1. Introduction

This paper is based on my experience in the field of Software Engineering, in particular in *Software Requirements Engineering*. Many observations and reports indicate, however, that there is not much difference between information processing systems in general (including hardware and software) and software in particular, as far as specification is concerned.

"Specification" is used in many different meanings. I am discussing the kind of specifications used by practioners in an industrial environment. My objective is to provide some information useful for achieving better requirements specifications, which in turn enables the developer to produce better software more efficiently, i.e. to improve *quality* and *productivity*. Note that the latter is the *overall* productivity, which does not necessarily imply cheaper specifications (see 2.2).

In the following chapter 2, some fundamentals are discussed. These include the life cycle model and the distribution of costs over the various activities, some definitions, and a rationale for semi-formal specification. Chapter 3 provides a general outline of a specification system, whose desirable properties are deduced from the qualities of good specifications.

---

<sup>1</sup> An earlier version of this paper was published before (Ludewig, 1987).

Any descriptions of specification systems are based on the material available to me. This information may be incomplete, or out of date. Therefore, I am sorry in case some features are not reported correctly. Please contact the suppliers (see 7.3).

Trademarks of software tools etc. are not indicated in this paper.

<sup>2</sup> Author's address: Institut für Informatik, ETH-Zentrum, CH 8092 Zürich

In chapter 4, we present some typical specification systems. The primary goal is to show some typical features of such systems rather than to describe them in detail. Chapter 5 reports some observations from the author's own experience with specification systems. In chapter 6, some general conclusions are drawn. The references in chapter 7 include a list of suppliers.

## 2. Fundamentals

### 2.1 Life Cycle Model

Only very small systems can be built in the same way as primitive peoples build houses. As soon as the system is slightly complex, a systematic approach is necessary. The sequence of steps to be taken from the first idea to operation and further on until the system is discarded, is called the *System Life Cycle*. Though there are many different life cycle models, they are all based on the distinction between certain *activities* or *phases*, namely

analysis and specification  
                           design  
                           implementation  
                           integration  
                           operation and maintenance

Note that the life cycle may be used as a *phase model*, or a *model of activities*, or a *list of roles*. In the sequel, the second meaning is assumed.

Recently, the life cycle concept has been attacked by several authors, not only because it does not reflect the experiences of many projects, but also because alternative ways of building systems (for instance by prototyping) are ignored. See the references in 7.1.

### 2.2 Cost Distribution

About half - or even two thirds - of the total cost of software are caused by activities which take place when the software is already operational, i.e. during "maintenance"<sup>1</sup> (Boehm, 1976). Therefore, every attempt to reduce the high cost of software should be focused on maintenance.

---

<sup>1</sup> Note that there is an important difference between maintenance of hardware and of software: while hardware is actually *maintained*, i.e. the original state is conserved or restored, software is corrected, extended, or adapted to new requirements, i.e. it is *modified*. A program is different from its original state after maintenance.

There are three ways of reducing the need for maintenance:

- reduce need for correction
- reduce effort for modification
- reduce total volume (by using standard components)

A good specification contributes to each of these subgoals. Therefore, the overall goal is not to reduce the effort for specification, but rather to invest more in specification in order to save much more during maintenance (and also during design, implementation, and integration).

## 2.3 Terminology

### 2.3.1 Specification

Like many other relevant terms, **specification** is defined in a standard by the IEEE (1983):

- (1) A document that prescribes, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or system component. See also design ~, formal ~, functional ~, interface ~, performance ~, requirements ~
- (2) The process of developing a specification.
- (3) A concise statement of a set of requirements to be satisfied by a product, a material or process indication, whenever appropriate, the procedure by means of which it may be determined whether the requirements given are satisfied. (ANSI N45.2.10-1973)

This definition is compatible with the following one (from Kramer et al., 1982):

A description of an object stating its properties of interest. It usually implies that the description should try to be precise, testable, and formal.

It is recommended that "specification" be used with some attribute, e.g. "requirement specification".

Specifications are written and read by many people, like analysts, customers, managers, and programmers. Since these people differ widely in their background, education, and interest, they have usually not the same idea about content and style of a specification. Tools, which can change the representation of a given information automatically, can help to meet the requirements of more than just one single group.

### 2.3.2 The System Triangle

When we talk about programming systems, or specification systems, we distinguish three components, or sets of components, namely *methods*, *languages*, and *tools*.

*Methods* indicate how to proceed, like recipes in a cookbook. *Languages* restrict the set of possible statements to a particular universe of

discourse, and to a certain syntactical representation. *Tools* check, store, and transform such statements.

All three are strongly interrelated by the *abstract concepts* of the (specification-) system. Note that the term "methodology" means "science of methods", though it is often misused for "method". Figure 1 exemplifies the system triangle:

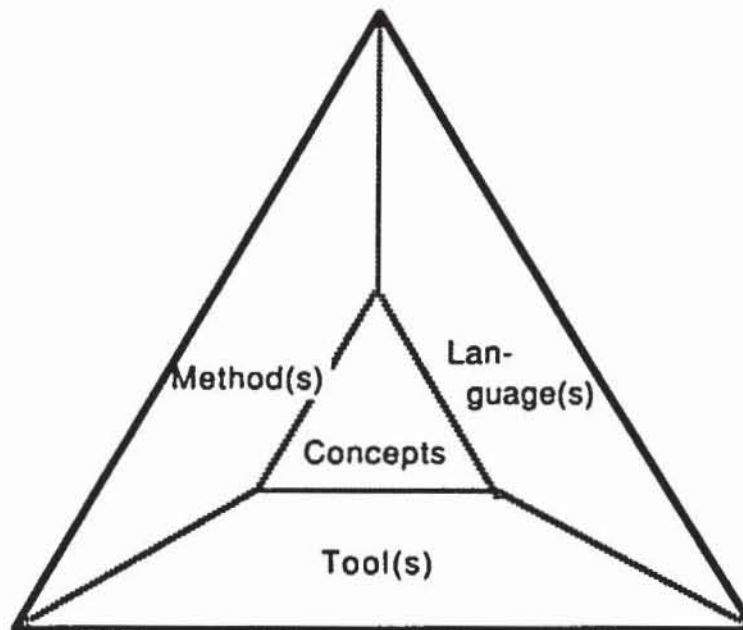


Figure 1: System triangle

### 2.3.3 Levels of Formality

The IEEE glossary (IEEE, 1983) contains also a definition of **formal specification**:

- (1) A specification written and approved in accordance with established standards.
- (2) In proof of correctness, a description in a formal language of the externally visible behavior of a system or system component.

The definition of **formal language** is, in turn:

A language whose rules are explicitly established prior to its use. Synonymous with artificial language. ...

The distinction of only two styles, formal and natural, is not sufficient for our purpose. Therefore, another two levels of formality are introduced, formatted and semi-formal, restricting "formal" to those languages which can be used for formal reasoning.

Figure 2 compares the styles, or levels of formality.

Style	Syntax	Semantics	Example
informal		not (precisely) defined	natural languages
formatted	restricted	not (precisely) defined	forms
semi-formal	defined	partially defined	pseudo-code
formal	defined	defined	programm. languages

Figure 2: levels of formality

For coding programs, we use a formal language. (Though the semantics of most programming languages are not precisely defined, if at all, there is always a translator which provides a de-facto-definition.) All other documents are written in informal language, sometimes on forms. Forms impose certain restrictions on the way natural language is used, and require the user to answer all relevant questions. Semi-formal languages are comparatively new; their first application was as program design languages (pseudo code).

#### 2.4 Semi-Formal Specification

Scientists all over the world have done much work on formal specification techniques, like algebraic specification. These techniques, however, have not yet reached a state sufficient for users in industry. Therefore, this paper does not treat formal specification. Semi-formal specification, i.e. an approach which is based on semi-formal specification languages, has (at least for the time being) several advantages:

- The languages can be learned and understood with limited effort by people who did not have extensive training in formal methods
- Documents resemble those written in natural language
- Incomplete and vague information fits better in such a system

On the other hand, semi-formal specification systems are superior to traditional informal specifications because

- many deficiencies which would be buried in plain text become visible
- it can be stored in, and retrieved from, a data base
- automatic tools can be used for checking and changing the notation.

Figure 3 shows schematically how the software development process is influenced by a system for semi-formal specification. In the traditional approach, there is practically no formalized information until the software is coded. Then, full formalization must be achieved in a single step. This method is, as we all know, error prone, because there are many misunderstandings, inconsistencies, simple errors and other shortcomings in the specs which are not discovered, because the document produced next, i.e.

the code, can only be understood at the level of single instructions. In the modern approach, there are much better chances for detecting deficiencies of the specs, and improving them. Therefore, specification systems do not accelerate the specification phase, but improve the quality of the resulting document.

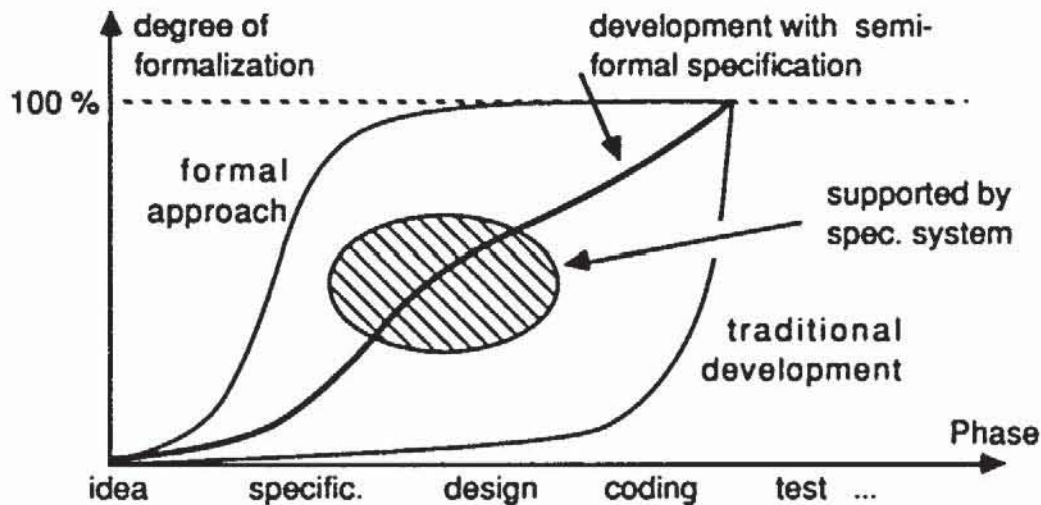


Figure 3: Degree of Formalization during the Software Life Cycle

### 3. Principles of Specification

#### 3.1 Qualities of Specifications

A specification should be

- correct (i.e. it should reflect the actual requirements)
- complete (i.e. it should comprise all the relevant requirements)
- consistent
- unambiguous
- protected against loss of information and unintended changes
- easily writeable and modifyable
- readable and concise (in order to ease the communication between user and analyst)
- implementable (i.e. it should ease design and implementation)
- verifiable (i.e. there should exist a procedure to check whether or not the product complies with its specs). This quality is also called "testable".
- validateable (i.e. there should be a mechanism to ensure that the specification really reflects the user's specification)
- traceable (i.e. when the specification is changed, it should be easy to identify all statements in other documents affected by that change).

Note that these goals are highly inconsistent. For instance, a formal (e.g. algebraic) specification is verifiable, but not readable for most people, at least not for the average customer. Therefore, it is not validateable.

The first four of the qualities listed above (correctness, completeness, consistency, and unambiguity) do not have the same meaning to all people: vendors of tools for specification, for instance, often claim that their system can guarantee correctness. This does, of course, not imply that the *content* of the specs is correct with respect to the intentions of the customer, but only that certain formal requirements are met. The reason for this is that there is no reference (except the user's brain) to prove specifications correct or complete, in contrast to programs being provably correct with respect to the underlying specification.

### 3.2 Useful Properties of Specifications

In order to achieve the qualities listed above, certain properties are obviously useful:

- The specifications must be recorded on some permanent medium (e.g. paper, magnetic tape).
- They should be as formal as possible, and as informal as necessary. Also, they should support the processing of information which is vague, incomplete, or not yet well defined (i.e. providing a fill-in that indicates the lack of information).
- Specs should exist only in one single copy ("single source concept").
- There should be tools for automatic checks and transformations between different representations.
- Specs must be available in representations appropriate for those who have to use them (e.g. graphical representations which naturally mirror human's way of thinking).

### 3.3 Specification Systems Requirements

From the useful properties stated above, we can derive the requirements of specification systems; such a system should provide

- a data base system as the central information repository,
- a semi-formal specification language and several representations, including a graphical one,
- tools for all clerical tasks (storing, retrieval, checking, transformation).

Since software systems are developed by several people, and usually exist in several versions and variants at the same time, the specification system should also provide

- multi user operation of tools,
- automatic management of versions and variants.

### **3.4 General Structure of a Specification System**

As mentioned above, an ideal specification system consists of a method, a language, and a set of tools, which are all based on a common set of concepts. The list following below summarizes the most desirable features.

#### **Abstract concepts**

- Life cycle model
- Stepwise completion
- Permanent validation

#### **Methods supported by the system**

- Enter every information immediately
- Allow for informal texts
- Check early for correctness, completeness, consistency, unambiguity
- Concentrate on information necessary for specification.

#### **Languages**

- Semi-formal specification language
- Several syntactical representations of a specification (e.g. graphics, tables etc.).

#### **Tools**

- Multi-user data base system, which provides for version- and variant management
- Tools for checking, retrieval and selection.

In reality, however, most systems are incomplete. They are usually based on either of the components, and do never cover the full scope. Some activities started from a particular method (e.g. SA, see 4.2), or from a certain representation (e.g. SADT, see 4.1), or from a set of tools (e.g. EPOS, see 4.5). In the following chapter, some specification systems are presented. Our goal is to give an idea of their dominant feature; we certainly do not attempt to provide complete information. Please refer to the references (7.1, 7.2), or contact the vendors listed in 7.3.

## **4. Specification Systems: Some Examples**

In this chapter, we present some examples of specifications in various languages. Additionally, we briefly describe their underlying methods. The purpose is to show some typical styles rather than to describe systems in detail. These are the examples chosen for this paper:

**SADT (4.1)** is one of the best known graphical languages for expressing specifications;



**Structured Analysis (SA) (4.2)** is similar to SADT, but has a wider range (towards design). We present it together with **teamwork** and **ProMod**, tools which support SA.

**PSL/PSA (4.3)** is the classical tool-based specification system.

**SREM (4.4)** is a very powerful system for describing, and simulating, real time software.

**EPOS (4.5)**, another tool dedicated to the development of real time systems, is fairly successful in Germany and central Europe.

**SPADES (4.6)** was developed by the author and co-workers. It is mentioned here because chapter 5 refers to it.

Many more systems should be presented, like HOS/USE.IT or PAISLey for functional specification, MASCOT/Perspective for high level design, NET for specification and simulation based on Petri-Nets, etc. (cp. 7.1, 7.3).

*Our list covers only some of the systems that we know, which in turn are certainly only a small fraction of those that exist. Therefore, our choice should by no means be interpreted as a judgement or recommendation.*

#### 4.1 SADT (Structured Analysis and Design Technique)

SADT was developed by SofTech between 1972 and 1975. It covers the requirements analysis, the design and the documentation of specifications, aiming at improved communication between analysts, developers, and users.

##### 4.1.1 The Method

The method SADT focuses on data flow and implies a stepwise refinement of so called SADT-diagrams which are hierarchically ordered. In its original definition (Ross, 1977), there is a duality between so called actigrams and datagrams modelling the data flow in two different ways representing different views of the system:

- **actigrams** identify functions as central elements of the description and data providing e.g. input or output for the functions
- **datagrams** identify data as central elements of the description and functions providing e.g. input or output for the data.

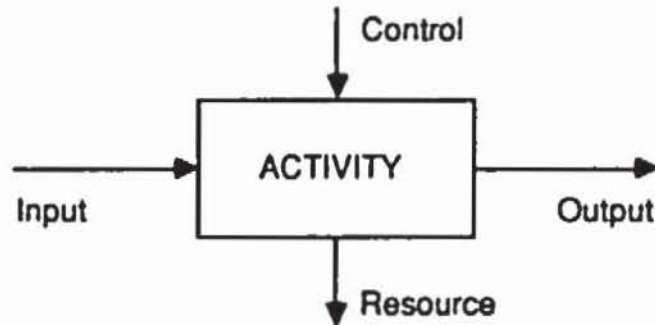
The redundancy makes it possible to prove consistency, i.e. one can check whether every function in an actigram is comprised in some datagram, and vice versa.

##### 4.1.2 The Language

SADT is a graphical specification language allowing the user to describe the system in terms of activities and data. As outlined above, on the one hand there are actigrams consisting of activities and data. Activities are represented by boxes and data by arrows. On the other hand there are

datagrams, where boxes stand for data, while arrows represent activities. Practical experience, however, indicates that most users tend to use only actigrams. In order to control complexity, the language restricts the number of boxes per SADT-diagram to seven.

Figure 4 shows an SADT-box with its typical components:



**Figure 4:** SADT-box (Actigram)

Actigrams on the following pages show an activity ("ASSIST SADT USERS") at two levels of refinement; the highest level (diagram SAS/A-0), where the whole system is represented as a single box with inputs from, and outputs to its environment, is not shown. Note that the second actigram (fig. 5 b) refines an activity ("CREATE KITS") of the first one (fig. 5 a). (Source: Lissandre et al., 1984, from IGL, Paris)

#### 4.1.3 Tools

SADT is still a paper and pencil method. And there is no problem in drawing all the diagrams once. However, when there are changes (and the need for change is the only property of software that does never change), diagrams must be redrawn again and again. This is very annoying. Therefore, there have been several activities for providing tool support; the examples in figures 5 a, b were produced by such a tool (but for technical reasons redrawn by the author). Their capabilities range from simple graphics (i.e. they are used as an automatic drawing machine) to fairly sophisticated programs which do some semantic checking and analysis. According to D.T. Ross, who invented SADT, "none (of the tools) is fully successful in implementing SADT" (Ross, 1985 b).

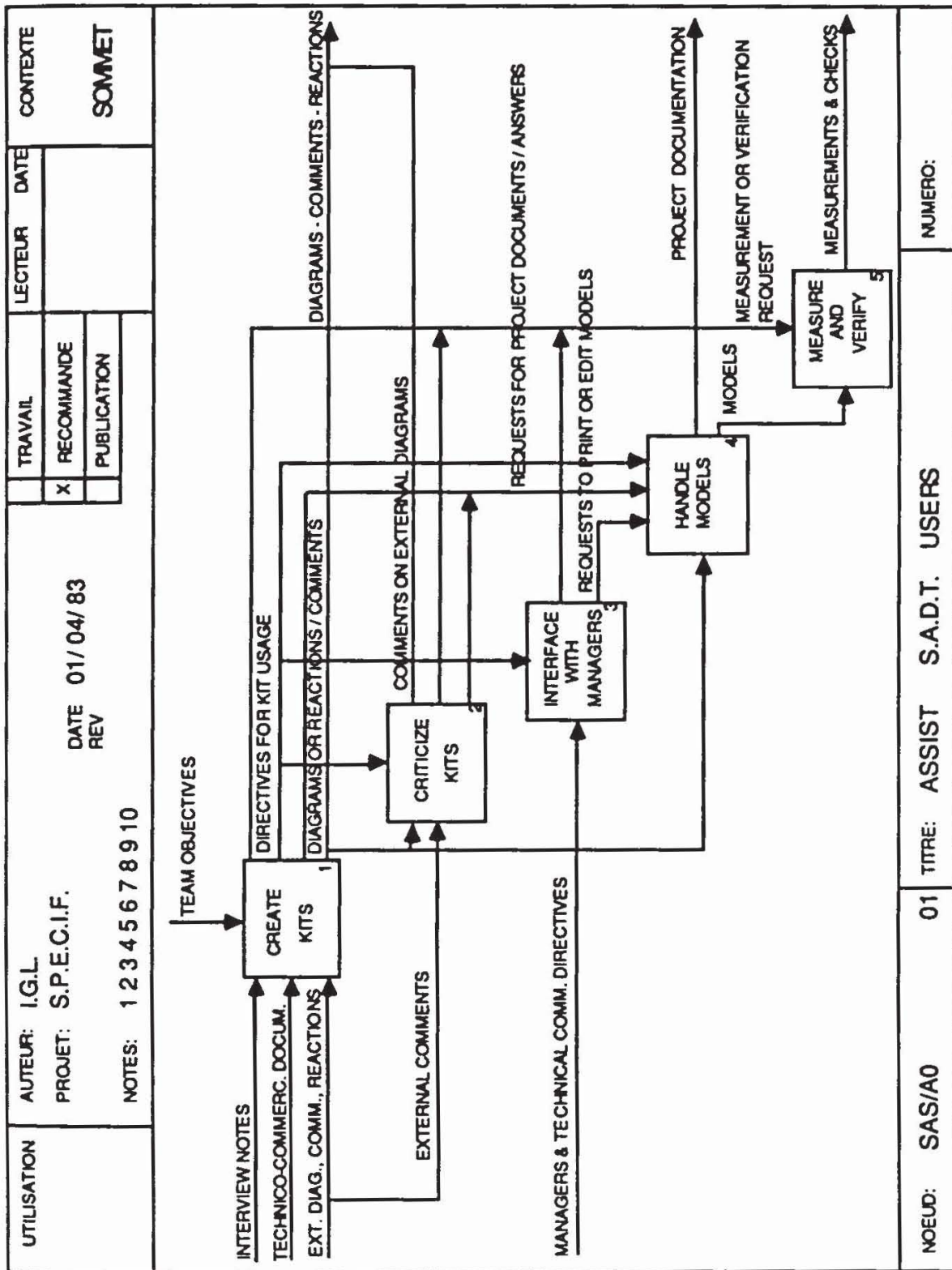
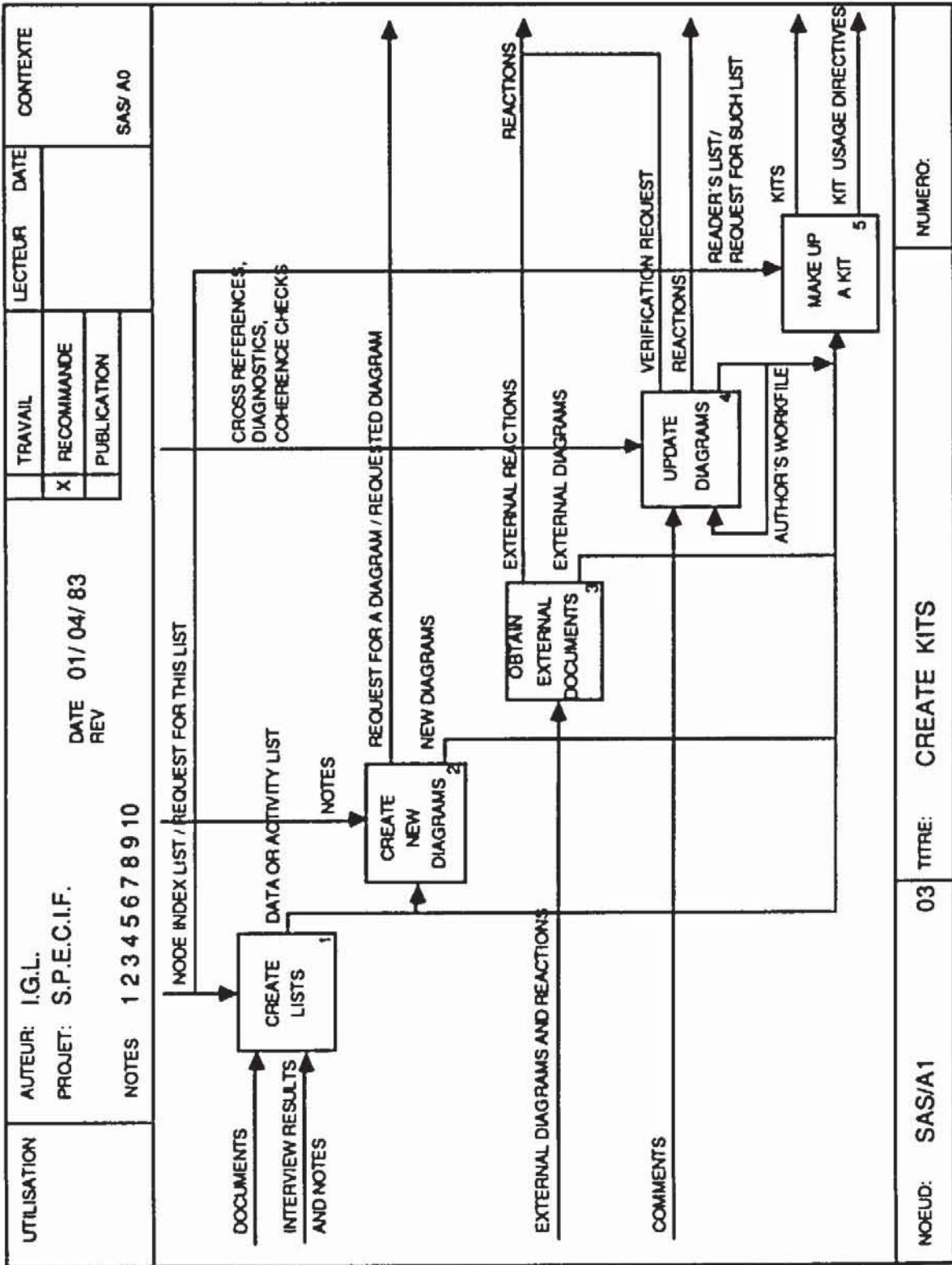


Figure 5 a: SADT-Diagram, first level below top



NUMERO:

03 TITRE: CREATE KITS

NOEUD: SAS/A1

Figure 5 b: SADT-Diagram, second level

## 4.2 Structured Analysis (SA)

SA was developed by deMarco (1978). Although the name is very similar to SADT, only the data flow as the central principle is common to both. SA is supplemented by Structured Design (SD), a design technique.

### 4.2.1 The Method

The method allows the user to model a system with **data-flow diagrams (DFDs)** consisting of data, and processes transforming the data. In other words, DFDs describe the flow of data through the system by denoting sources and sinks for data flows, the data flows itself, and processes. So called **minispecs** are used to describe processes in more detail. For refining the structure of data, a **data dictionary (DD)** is applied. The system's dynamic behaviour cannot be expressed in the notation of SA; therefore, **real-time diagrams (RTDs)** are used for this purpose (see 4.2.3). SA proposes a stepwise decomposition of DFDs so that each process in the parent DFD is broken down into several child DFDs. Consequently, several levels of DFDs emerge.

SA proposes two major steps. The first one is to develop a so called context diagram, which shows how the system is connected to its environment. Hereby, the user defines the interface in terms of sources and sinks of the environment, processes, data flows, and files. Note that the data flow consists of both the data and the direction of flow.

In the second step, the user partitions, and refines the system "as long as possible", i.e. each process of a DFD is described in more and more detail until the level of atomic processes is reached. Then the user writes minispecs demonstrating the algorithmic structure of these atomic processes. Also, a data dictionary is created containing the structure of the data. SA also gives naming conventions for processes, dataflows, files, which can help the user to express his understanding most clearly.

### 4.2.2 The Language

The sources and sinks belonging to the environment of the target system are shown as **boxes** on a data-flow diagram. Other symbols are **circles** representing processes, **arrows** representing data flows, and **bars** representing files. Please note that the first time a file is referenced in a DFD two bars are used (see fig. 7a, file "Bit Map") while further references to this file (in other DFDs) are denoted by a single bar (see fig. 8a, file "Bit Map").

The minispecs are written in pseudo-code, the data described in the data dictionary is written in a BNF-like notation.

The examples given below were taken from a paper on an early version of the Tektronix-tool (Bell, 1985, cp. TekCASE in 7.3). They show data-flow diagrams, together with minispecs and information stored in the data dictionary.

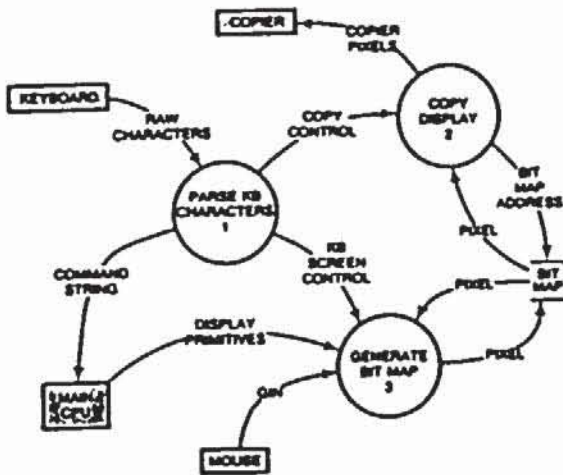


Figure 6 a:  
DFD for a display controller

```

COPIER_PIXEL = PIXEL
BIT_MAP_ADDRESS = INTEGER
PIXEL = LOGICAL
TEXT = ASCII_CHAR
GRAPHICS
    = [POLYLINE | POLYMARKER | AREA_FILL | GDP]
SCREEN_CONTROL
    = [SCROLL | ERASE | REVERSE | HORIZ_CONTROL]
KB_SCREEN_CONTROL = SCREEN_CONTROL
BIT_MAP = {PIXEL}
BIT_MAP_PIXEL = PIXEL
TEXT_PIXEL = PIXEL
GRAPHICS_PIXEL = PIXEL
COMMAND_STRING = {{ASCII_CHAR} + DELIMITER}
DISPLAY_PRIMITIVES
    = GRAPHICS + TEXT + SCREEN_CONTROL
GIN = X_POSITION + Y_POSITION
X_POSITION, Y_POSITION = INTEGER
    
```

Figure 6 b: Data Dictionary for 6 a

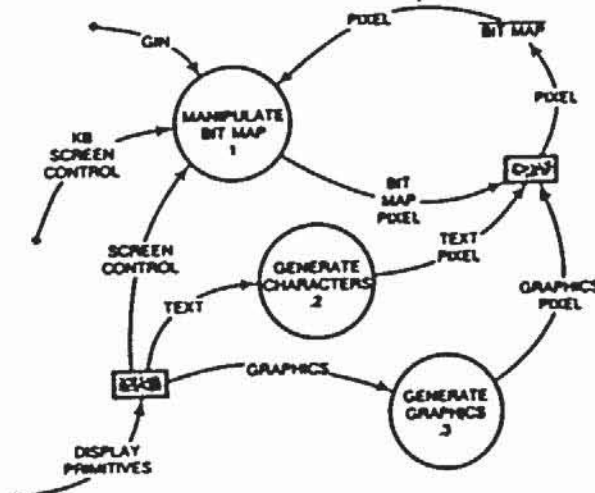


Figure 6 c: DFD for Generate Bit Map from fig. 6 a

```

CHARACTER_GENERATION_MAP_LOCATION
    = ASCII_CHAR
FOR I = 1 TO 12 DO
    CHAR_GEN_MAP_INDEX = 1
    FOR J = 1 TO 9 DO
        IF CHAR_GEN_MAP_CONTENTS (J) = TRUE
            SEND 1 TO BIT MAP
        ELSE
            SEND 0 TO BIT MAP;
        END
    END
END
    
```

Figure 6 d: Minispec for 6 c

#### 4.2.3 Tools for Structured Analysis

*Teamwork* is a product of CADRE Technologies Inc. It consists of a set of tools for Structured Analysis (*Teamwork/SA*: DFDs, process specifications), information modeling (*Teamwork/IM*: entity relationship diagrams, data dictionaries), real-time system modeling (*Teamwork/RT*), and Structured Design (*Teamwork/SD*). It is built on top of a data base system; other user-specific tools can access that data base via *Teamwork/ACCESS*.

The tools are implemented on powerful workstations (Apollo, DEC VAX-station, IBM RT, SUN, HP). The DBMS is not part of *Teamwork*; instead, on any of the *Teamwork* machines, an existing DBMS has been integrated. Therefore, *Teamwork* offers very fast access, quick consistency checking, and networking. Several users can work with the same data base, only the information currently shown on any of the screens is locked. *Teamwork* has a very nice user-interface, and it seems to be really fast.

Like most other SA-tools, *Teamwork* supports Structured Design (SD, see Yourdon, Constantine, 1979) as the method to be used for software design. SD, however, is not an up-to-date approach, so it does not give much help to a designer who strives for well structured programs. A new component recently announced is the *Buhr Structure Graph Editor*, which allows for graphical design of Ada programs.

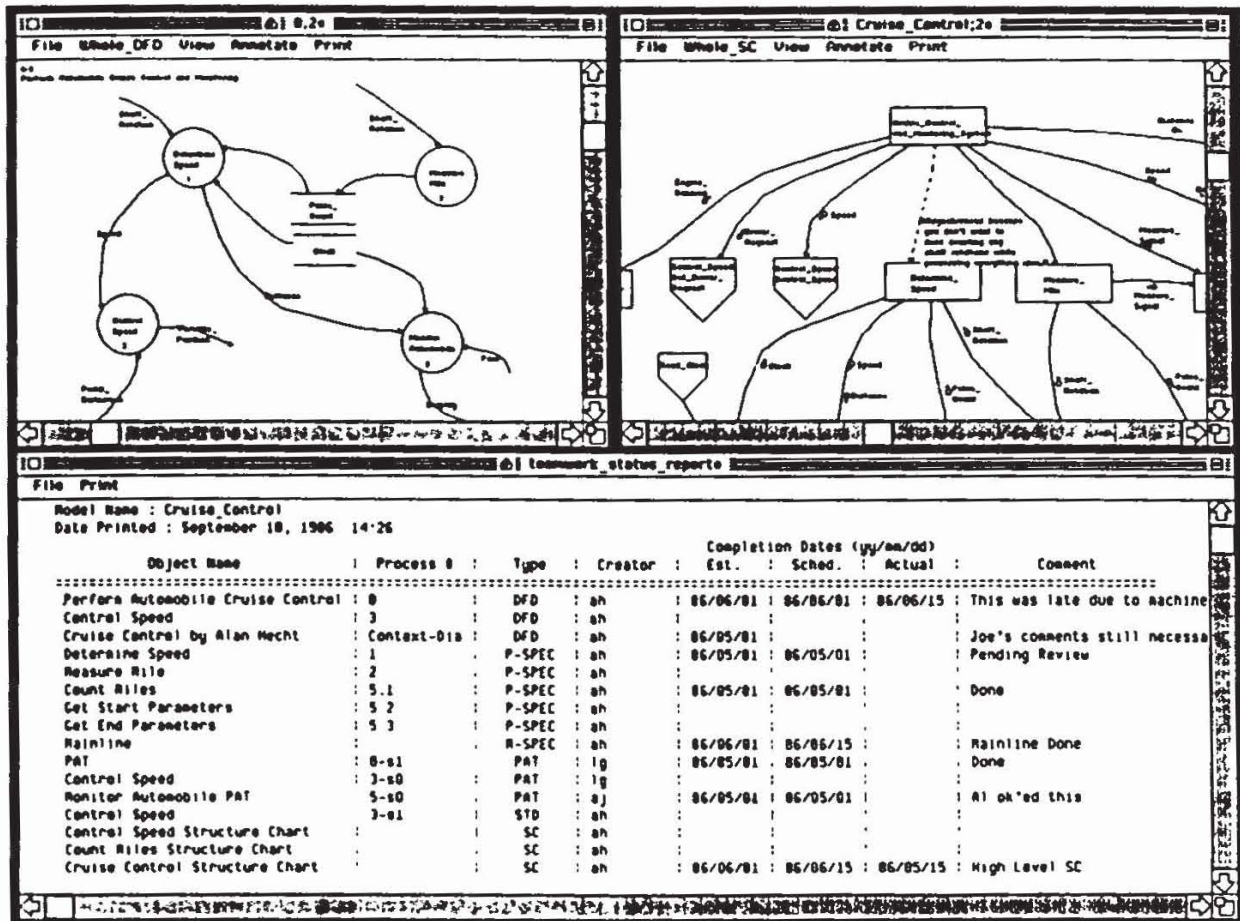


Figure 7: snapshot from *Teamwork*, showing three windows on a screen

*ProMod*, which was developed by GEI, Aachen, FRG, is another environment based on the SA-concept. Its central information repository is the so called *ProMod* project library.

Tools for Structured Analysis are:

- DFD-processor      editing and processing of data flow diagrams
- DD-processor      data dictionary system
- TD-processor      minispec-processor
- AAD-analyzer     cross checking between DFDs, DD and minispecs.

Like *Teamwork*, *ProMod* supports real time analysis by control specification (finite state machines, see fig. 8).

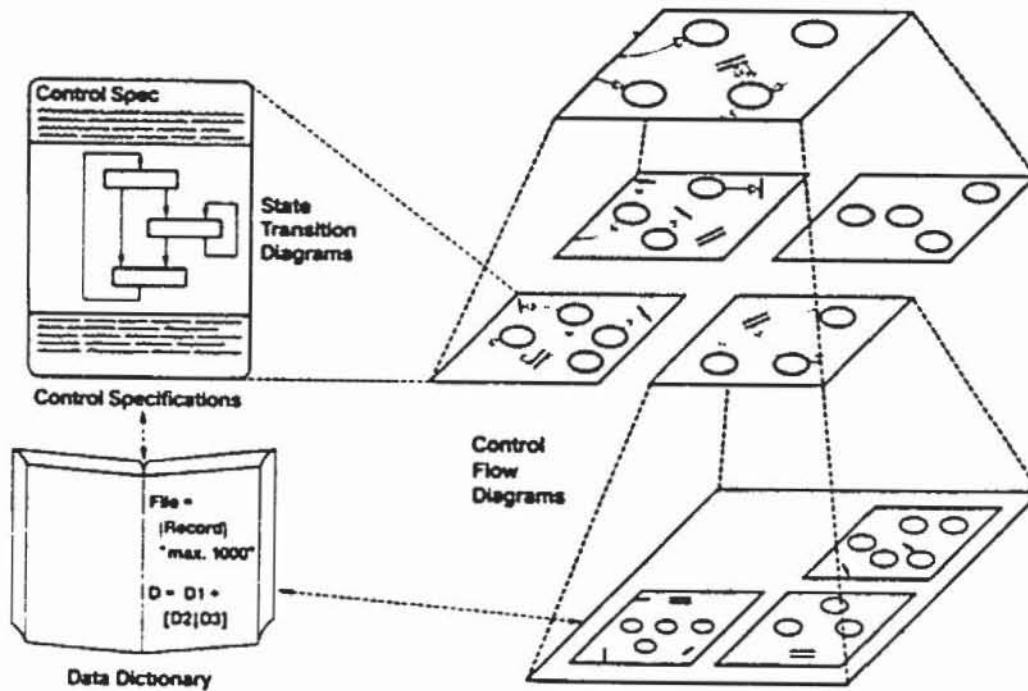


Figure 8: Real time modeling with control specifications

*ProMod* offers tools for Modular Design, an extension of SD which supports data abstraction:

Translator	from SA- to MD-System
MS-processor	for module specifications
FS-processor	for functional specifications
DD-processor	data dictionary system
SE-analyzer	cross-checking at design level

Below the level of Modular Design, *ProMod* provides *PDL* and *DARTS*, two pseudo-code-systems. Other tools generate code-frames in several languages (PASCAL, FORTRAN, COBOL).

*ProMod* is available on VAX/VMS, and IBM-PC (XT, AT)/PC-DOS. Compared to *Teamwork*, it is less impressive at the user interface. Since there is currently no real DBMS in *ProMod*, accesses are rather slow, and checks for consistency may take some time. On the other hand, *ProMod* does not leave the software developer alone after analysis. These components would be even more useful if there would be a way to trace late changes through the various documents (in both directions).

There is a large number of other tools that support SA/SD, including *IDE Structured Analysis and Design Tools*, which are based on *Software through Pictures*, *TekCASE* by Tektronix (*Analyst/RT*, *Designer*, *Auditor*), *Structured Architect* by META-Systems (see 4.3) and many others (cp. 7.3).



### 4.3 Problem Statement Language/Problem Statement Analyzer (PSL/PSA)

*PSL* was developed at the University of Michigan by the ISDOS-project (Information System Design and Optimization System) in the seventies. *PSL* primarily supports requirements analysis and documentation.

*PSL/PSA* was the very first tool-based system for semi-formal specification which was actually useful - and commercially successful. All other such systems are copies of *PSL/PSA*, at least in part. Like some other tools developed at universities, *PSL/PSA* is now supported, improved, and commercially distributed by a private company (META-systems, cp. 7.3).

#### 4.3.1 The Method

*PSL/PSA* emerged since 1970 in a very organic manner, and Daniel Teichrow and his co-workers did never put too much effort in writing down the method they had in mind. Still, there is a method behind *PSL*: It is the one sketched in 3.4.

#### 4.3.2 The Language

*PSL* is based on the entity-relationship approach first described by Chen in 1976 but applied long before. The entity-relationship model was originally used as a database model splitting the world to be described into entities, and relationships between these entities. The dominant feature of this approach is the similar treatment of entities and relationships.

Different from *SADT* and *SA*, *PSL* is a linear (textual) language. *PSL* provides some 30 entity-classes and 75 relations to the user. The most important ones are:

#### Entity-classes:

REAL WORLD ENTITY	objects outside the target system
PROCESS	activities
INPUT	input data
SET	set of data elements

#### Relations:

GENERATES	e.g. <process> GENERATES <data>
RECEIVES	e.g. <process> RECEIVES <data>
UPDATES	e.g. <process> UPDATES <data>
CONSISTS	describes data structures; e.g. colour CONSISTS yellow, red, green, blue

Figure 9 shows a fragment of a *PSL*-input source listing; the specification describes cargo-vessels and their organizational environment.

(Source of all examples in 4.3: Papers from ISDOS, 1983)

PSA Version A5.2R2M

Jul 23, 1983 20:05:19

PSL/PSA - ISDOS - VM/CMS

## IPSL Input Source Listing

Parameters: DB=VESSEL.DBF INPUT=VESSEL.PSL SOURCE-LISTING NOCROSS-REFERENCE  
 UPDATE DATABASE-REFERENCE NOWARN-NEW-OBJECTS NOSTATEMENT-NUMBERS  
 DBNBUF=200 WIDTH=84 LINES=60 INDENT=0 HEADING PARAMETERS PAGE-CC=0H  
 NOEXPLANATION

## LINE S T M T

```

1 > /* This is a set of PSL statements to define user views */
2 >
3 > /* Here is the global users' view */
4 >
5 > DEF ENTITY      UserViews;
6 >   TKEY          'Global';
7 >   SUBPARTS ARE  User-View-1,
8 >                 User-View-2,
9 >                 User-View-3,
10 >                User-View-4,
11 >                User-View-5,
12 >                User-View-6,
13 >                User-View-7;
14 >   DESC;
15 > This is a global view of a ship company.;
16 >
17 >
18 > /* ELEMENTS are declared */
19 >
20 > DEF ELE          Vessel,Cargo-Volume,Details,Port,Date-of-Arrival,
21 >                 Date-of-Departure,Consignee,Container#,Size,
22 >                 Shipping-Agent,Waybill#,
23 >                 Delivery-Date,Contents,
24 >                 Handling-Instructions;
25 >
26 >
27 > /* Here is the local users' view */
28 >
29 > DEF ENTITY      User-View-1;
30 >   TKEY          'V1';
31 >   CSTS OF       View1-Ship;
32 >   ATTR ARE      FREQUENCY-IS          100,
33 >                 TIMING-REQUIREMENT  25;
34 >   RPD IS        'E. Basar';
35 >   DESC;
36 > Information is stored about each ship, including
37 > the volume of its cargo storage capacity.;
38 >
39 >
40 > DEF ENTITY      User-View-2;
41 >   TKEY          'V2';
42 >   CSTS OF       View2-Ship,
43 >                 View2-Ship-Port,
44 >                 View2-Port;
45 >   ATTR ARE      FREQUENCY-IS          100,
46 >                 TIMING-REQUIREMENT  50;
47 >   RPD IS        'E. Basar';

```

Figure 9: PSL source listing (incomplete)

PSA Version A5.2R2M

Jul 23, 1983 20:05:19  
PSL/PSA - ISDOS - VM/CMS

Contents Report

Parameters: DB=VESSEL.DBF FILE=PSANAMES.PSATEMP NOCOMPLETENESS-CHECK  
NOINDEX NOPUNCHED-NAMES LEVELS=ALL LINE-NUMBERS LEVEL-NUMBERS  
OBJECT-TYPES PRINT NONEW-PAGE DBNBUF=200 WIDTH=84 LINES=60 INDENT=0  
HEADING PARAMETERS PAGE-CC=ON NOEXPLANATION

```

1* (ENTITY)      1 User-View-1
  1 (GROUP)      2 View1-Ship
  2 (ELEMENT)    3 Vessel
  3 (ELEMENT)    3 Cargo-Volume
  4 (ELEMENT)    3 Details
2* (ENTITY)      1 User-View-2
  1 (GROUP)      2 View2-Ship
  2 (ELEMENT)    3 Vessel
  3 (GROUP)      2 View2-Ship-Port
  4 (ELEMENT)    3 Port
  5 (ELEMENT)    3 Vessel
  6 (ELEMENT)    3 Date-of-Arrival
  7 (ELEMENT)    3 Date-of-Departure
    
```

Figure 10 a: PSA Contents Report (incomplete)

An \* in (i,j) means that column j is contained directly or indirectly in row i. The columns do not consist of anything further. Intermediate GROUPS are ignored.

14 Size	-----	/																	
13 Handling-Instructions	-----	/																	
12 Contents	-----	/																	
11 Delivery-Date	-----	/																	
10 Waybill	-----	/																	
9 Shipping-Agent	-----	/																	
8 Container	-----	/																	
7 Consignee	-----	/																	
6 Date-of-Departure	-----	/																	
5 Date-of-Arrival	-----	/																	
4 Port	-----	/																	
3 Details	-----	/																	
2 Cargo-Volume	-----	/																	
1 Vessel	-----	/																	
-----																			
1 User-View-1	-----		*	*	*														
2 User-View-2	-----		*		*	*	*												
3 User-View-3	-----		*		*	*	*	*	*										
4 User-View-4	-----		*		*	*	*	*	*	*	*								
5 User-View-5	-----		*		*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
-----																			
6 User-View-6	-----		*		*			*											
7 User-View-7	-----		*		*			*										*	*
-----																			

Figure 10 b: Basic Content Matrix

#### 4.3.3 The Tools

PSA, the tool, is actually the system. It is built upon a CODASYL-database system, and offers a large selection of services and report functions. PSA is a huge FORTRAN-program consisting of some 60 000 loc. It is available on many time-sharing machines and workstations (IBM/MVS, IBM/VM, VAX/VMS, MicroVAX/ULTRIX, HP9000/UNIX, Tandem Guardian, Apollo and others).

The report in fig. 10 a shows a tree-structure (the hierarchical content-relation) by indentation. The second one (fig. 10 a) shows part of the same information in a table. These examples represent the traditional position of the ISDOS-project, where all output had to be line-printer oriented. Therefore, pseudo-graphics was the best representation available. But the system has now been extended by new tools, which support also high-resolution diagrams (not shown here).

#### 4.4 Software Requirements Engineering Methodology (SREM)

*SREM* is directly based on *PSL/PSA*; it was developed by TRW since about 1975. It supports the early phases (analysis, definition, verification, and validation of requirements) of the software development process. It is especially tailored for the development of large, embedded, real-time systems; the U.S. Air Force was the contractor of that project. For more information on *SREM*, see 7.1.

##### 4.4.1 The Method

*SREM* possesses two important features missing from most other methods or languages for specification. Firstly, it allows the stepwise development of specifications beginning with informal descriptions, from which an increasingly formal specification is developed. Secondly, data on performance (estimated or required) of the target system can be formally included in the specification. Since there is a tool for simulating specs, software designers can check early whether or not they will be able to meet response time requirements.

The method (*SREM*) is applied in seven steps:

1. **Define kernel:** identify the interface between the system and the environment and describe the data flows and the data-processing units inside the system.
2. **Establish baseline:** outline the very first description of the system using either the graphical R-Net formalism (R-Net means requirements-net, a stimulus-response network) or the linear language *RSL* (requirements statement language).
3. **Define data:** define data input to, and output from, each so called ALPHA (active component); complete, and improve the *RSL*-specification developed so far; implement Pascal-procedures for ALPHAs.

4. **Add project information, and establish traceability:** add management informations, e.g. deadlines, milestones, needed tools etc.
5. **Simulate functionality:** prove syntactical correctness and simulate dynamic behaviour
6. **Identify performance requirements:** define traceable, testable performance requirements; each path should be constrained by response time and accuracy
7. **Demonstrate feasibility:** prove that the current design is useful as a basis for a technical realization by means of a analytical feasibility study

#### 4.4.2 The Language

*SREM* offers the user two means of description, a graphical language (*R-Nets*) and a textual language (*RSL*).

#### Elements

are standard types defining features of each object of such a standard type. For example, MESSAGE, DATA, and FILE are standard types used to describe data; e.g. ALPHAs stand for processes. Elements represent nouns in the language.

#### Relationships

express logical links between Elements, e.g. <data> INPUT TO <alpha>. They represent verbs in the language.

#### Attributes

are used to complete the description of Elements, e.g. <data> INITIAL VALUE <value>. They represent adjectives in the language.

#### Structures

are used to define the sequences of processing steps and represent R-Nets, SUBNETs, and VALIDATION-PATHs in terms of RSL-statements.















ALPHA	
AND	
ENTRY NODE ON R_NET	
ENTRY NODE ON SUBNET	
EVENT	
FOR EACH	
INPUT_INTERFACE, OUTPUT_INTERF	
IF OR	
CONSIDER OR	
SELECT	
SUBNET	
RETURN	
TERMINATE	
VALIDATION_POINT	

Figure 11: Types and symbols in SREM

*R-Nets* are stimulus-response networks describing reactions in a system evoked by events. An *R-Net* consists of *nodes* (ALPHAs and SUBNETs) and *arcs* connecting the nodes. While ALPHAs are functional specifications of processes, SUBNETs are specifications of processes at a lower level of hierarchy. The flow of control is described by some single entry - single exit constructs (AND for parallel execution, OR for a multiway branch, FOR EACH for a loop). Additionally, validation-points can be inserted in order to express performance requirements.

See figure 11 for a list of all symbols used in R-Nets.

RSL is also used to enter the R-Nets, which are then automatically drawn.

A few examples are given below. Figure 12 shows a schematic R-Net. In figures 13a and 13b both the RSL-representation and the flow graph representation of a sample R-Net are exhibited (from papers by M. W. Alford).

#### 4.4.3 The Tool

Like PSL/PSA, SREM is based on a large tool, called REVS (Requirements Engineering Validation System). Beyond the abilities of other tools, REVS allows for project dependent extensions of the specification language, and for simulation of the specs. Maybe that REVS is currently the most powerful tool for specification; but prospective customers in Europe cannot buy it because its distribution is still limited to the U.S.

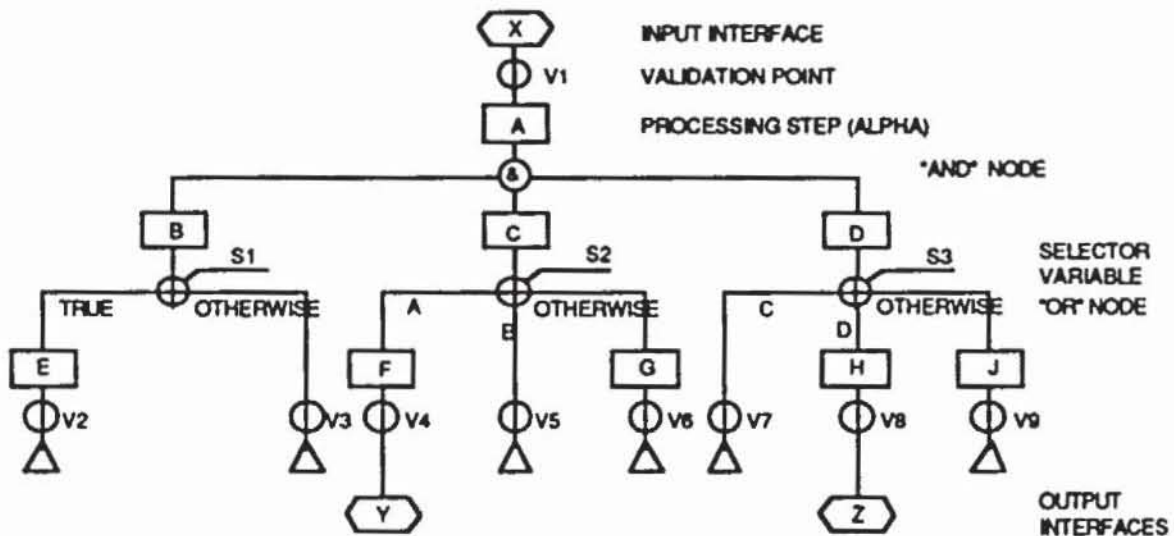


Figure 12: A schematic R-Net

R\_NET: PROCESS\_RADAR\_RETURN.

STRUCTURE:

```

INPUT_INTERFACE RADAR_RETURN_BUFFER
EXTRACT MEASUREMENT
DO (STATUS = VALID_RETURN)
  DO UPDATE_STATE AND KALMAN_FILTER END
  DETERMINE_ELEVATION
  DETERMINE_IF_REDUNDANT
  TERMINATE
OTHERWISE
  DETERMINE_IF_OUTPUT_NEEDED
  DO DETERMINE_IF_REDUNDANT
    DETERMINE_ELEVATION
    TERMINATE
  AND DETERMINE_IF_GHOST
  TERMINATE
END
END
END.
```

Figure 13 a: A sample R-Net, textual representation

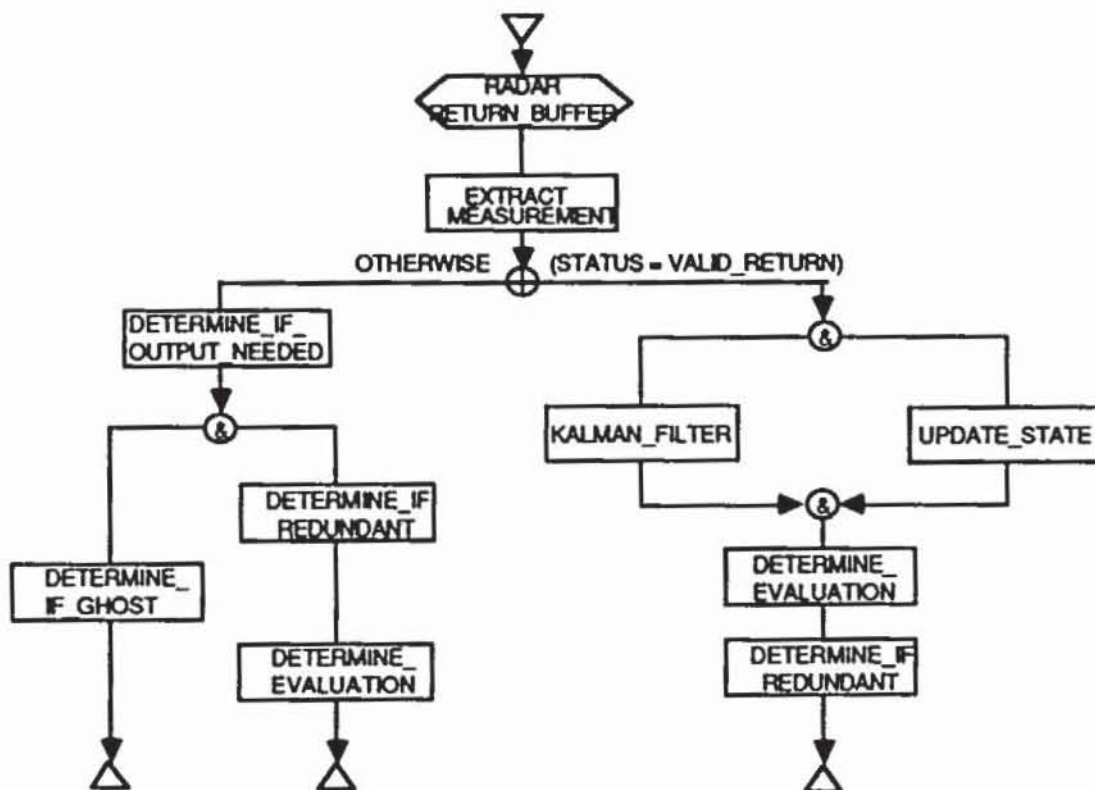


Figure 13 b: A sample R-Net, flow graph representation

#### 4.5 EPOS (Engineering and Project-management Oriented Support system)

EPOS was developed at TU Stuttgart by R. Lauber and co-workers since 1978. The product is now sold and supported by GPP (see 7.3).

##### 4.5.1 The Method

EPOS is one of the systems which do explicitly not support a particular method (though they do refer to the general principles of SADT). Several styles which are related to some method are supported.

##### 4.5.2 Languages

In EPOS, there is no clear distinction between languages and tools, i.e. the same name is used both for the language and for the program which is used for processing that language. Therefore, the following list may be inconsistent with other papers on EPOS.

There are three languages used for input:

EPOS-R	language for requirements definition (formatted)
EPOS-S	language for system design (semi-formal)
EPOS-P	language for project management information (semi-formal)

Several graphical representations can be generated by the tools, for instance Petri-Nets, Nassi-Shneiderman diagrams.

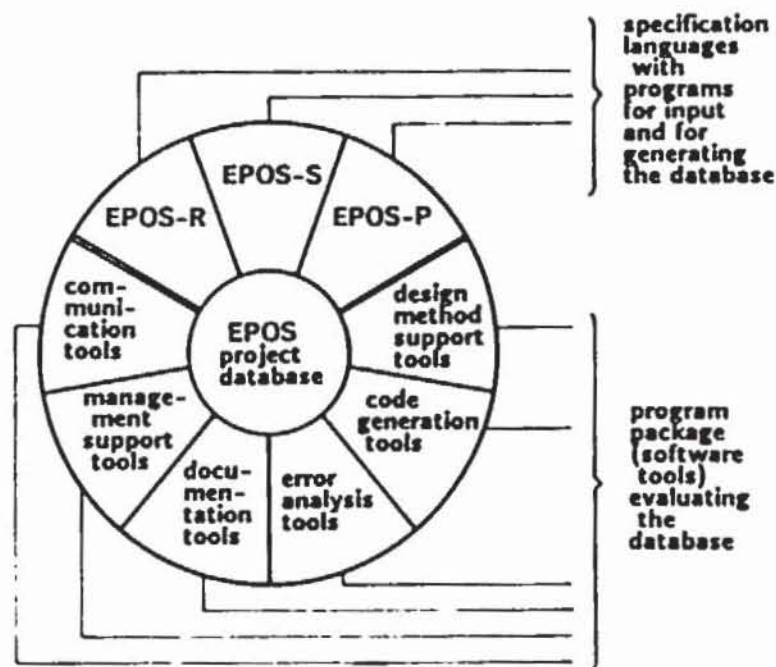


Figure 14: EPOS languages and tools



### 4.5.3 The Tools

The tools of EPOS are shown in fig. 14. The documentation tools offer a particularly large choice of diagrams, which can be generated from the EPOS project database (e.g. Petri-Nets, Nassi-Shneiderman diagrams, data flow diagrams and many others). All tools access the (non-standard) data base.

EPOS is available on most micro-computers and workstations (e.g. Apollo, Data General, DEC VAX and MicroVAX, HP 9000, IBM PC XT, AT, Intel 8086/80286, PCS Cadmus, SUN) and also on mainframes (IBM with VM or MVS, Siemens 7000 with BS 2000).

## 4.6 SPADES (Specification and Design System)

SPADES is based on ESPRESO, which emerged from 1977 to 1980 as a part of the authors doctoral dissertation. The name of the system was changed to SPADES after recoding in MODULA-2. SPADES was enhanced and extended at Brown Boveri & Co., Baden, Switzerland, until 1987<sup>1</sup>, when a decision was taken against further support.

### 4.6.1 The Method

Like other systems, SPADES is based on the idea of stepwise enrichment of a specification. As its name expresses, there is no clear distinction between specification and design. In the begin, a specification consists of informal texts. When the perception of the target system is more complete, objects like modules and data are introduced, and the description changes slowly from a specification to a (high level) design.

### 4.6.2 The Language

There are two notations for SPADES, linear and graphical. The latter is used for inspection of the data base only.

The linear notation, named SPADES-L, is a simple, but recursive language, i.e. hierarchies of objects may be represented by a nested description.

In SPADES-L, the target system and its environment are modeled by *entities* and *relationships*. *classes* for entities (e.g. "module") and *relations* for relationships (e.g. "contains") are predefined. Classes are structured hierarchically; when an entity belongs to a super-class (like "medium", which comprises "variable" and "buffer"), it means that its definite nature is not yet known.

All objects may contain any number of informal *texts* as attributes. Though these texts do not have a formally defined semantics, they may contain references to other objects which can be automatically evaluated. (This feature of SPADES is now fairly common.)

---

<sup>1</sup> Major extensions were done by M. Glinz.

A unique property of SPADES-L is its formal definition by an attribute grammar. This definition does not only cover the full context-sensitive syntax, but also the semantics, i.e. it provides a formal specification of the tool for entering specifications into the data base. (Note that languages like SPADES-L do not have semantics like programming languages, i.e. there is no formal mapping onto a sequence of actions.)

#### 4.6.3 The Tools

SPADES-T (see fig. 15), the tool of SPADES, is rather primitive, compared to modern tools with windowing etc. Specifications must be entered into a text file, which is then fed into CONV, the tool for processing SPADES-L. DECONV retrieves information from the data base. The data base system is in fact nothing but a data management based on the Entity-Relationship-concept. During work, all data is kept in virtual memory. This DBMS<sup>1</sup> has been successfully applied in other systems as well.

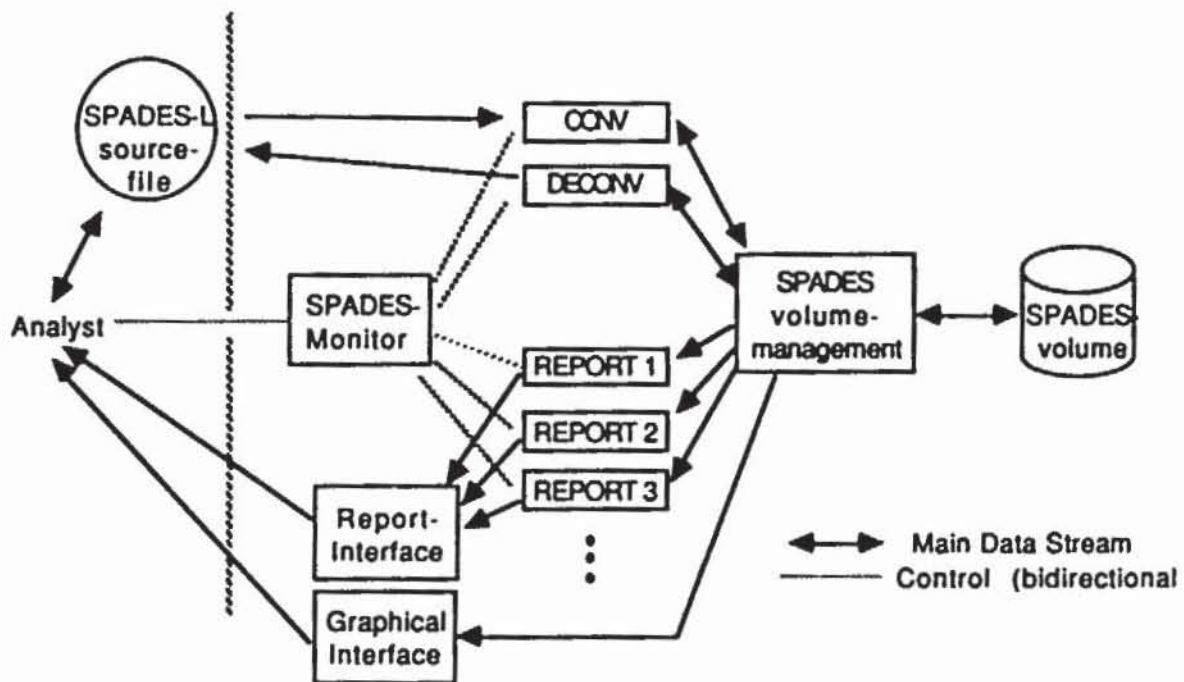


Figure 15: Structure and data flow of SPADES-T

## 5. Lessons from experience

From our work on ESPRESO and SPADES, and from many discussions with colleagues and users, I can report some experiences:

<sup>1</sup> Implemented by Hj. Huser at Brown Boveri

1. **A single person (or a group of two), with a sound background in Software Engineering and specification systems, is the prerequisite for the conception of a specification system.** The task requires much creativity, and cannot be well distributed. Experience with existing systems is mandatory.  
ESPRESSO was done in such a way; a recent attempt to define a software development environment in a team of well educated, but unexperienced freshmen failed.
2. **Writing a formal definition (not simply BNF!) of a specification language is hard work; building a system without such a definition is much harder.** The grammar of ESPRESSO was extremely useful, preventing any disagreements about the language.
3. **Building a specification system is certainly not a small task.** Such a tool is much more complex than an average compiler, not only because it requires a DBMS and a comfortable, graphical man machine interface. The most important difference is that the goal of a compiler project is fairly clear, while our ideas of specification systems are still rather vague. Therefore, even a mere prototype-system requires at least some ten to twenty developer years.
4. Different from our situation ten years ago, there are now good **standard components** (DBMSs, Window-packages, menue-generators, etc.). Extensive use of such components may reduce the effort. Even when analysis shows that such modules are deficient for some reasons, one should use them until a prototype has been implemented (maybe their deficiency does not matter, or first experiences make a major revision necessary).
5. **Graphical languages** are very important for two reasons. Obviously, users like graphics. Therefore, people will not buy a specification system without smart graphics. But there is another, more subtle reason: Only structures which are fairly simple can be graphically displayed (like trees). If one starts from a graphical representation, chances are good to obtain also an elegant, simple linear language.
6. In order to be useful, a specification system must be quite large. It is not possible to do only half of the work: Either the system is fairly **complete, comfortable, and reliable**, or it is condemned to die. Even when it is acceptable, it requires **steady support**.
7. A specification system is not just a tool, it is a whole **philosophy**. Users may be able to learn in a few hours how to operate the tool, but it takes months until they understand the philosophy. Good training, and a hot line for problems, are essential.
8. The decision to use a specification system, and the choice of a particular product, require a **commitment of the management**. Introduction of a specification system is very expensive. The cost of

the system itself and, possibly, of new hardware is often high, but it is usually negligible compared to the cost of training (or the failures due to insufficient training). The step to using a specification system is of similar importance like the step to using a computer; if you are not prepared to do it right, don't do it at all! Problems are inevitable, and there will be a situation when an important project seems to be late, because it is done with a specification system. If the management is not prepared to show a bold front against the breakers, they will not succeed.

9. The specification system may improve **quality assurance and project control**. Most vendors advertise some management tools as part of their products. To date, these are not very powerful. The real improvement stems from the discipline and standardization implied by the application of a specification system. This side effect is in fact the main improvement!

## 6. Conclusions

- There are many specification systems commercially available. Everybody who uses any of the more common machines, and operating systems, will find a specification system, if he or she wants to.
- It is obviously still possible to produce software (and systems) without a specification system. Special problems, like developing user interfaces, are actually better done by other approaches, e.g. prototyping.
- A specification system causes large expenses, mainly for training, but can improve quality and productivity significantly. Therefore, it should be regarded as a (medium- or long-range) investment.
- A specification system improves standardization in the way that every member of a project uses the same method, the same language, and the same tool. Moreover, the documents themselves have standardized features. This implies a discipline which is the real benefit of a specification system!
- Evaluation of tools should start with a decision for a method of software development. The tool should fit the method, not reverse.
- Currently, tools do not support maintenance of specifications. Therefore, the responsibility to change all documents, when one is modified, rests with the user. If he or she fails to do so (what is the normal situation), the specification becomes obsolete.
- Implementing one's own specification system is hardly feasible, because it takes at least ten person years to develop nothing but a prototype.

## 7. References and Addresses of Suppliers

### 7.1 Index to References

Books and papers in German are indicated by an asterisk; all others are in English.

#### Books on Software Engineering

Boehm, (1980), Fairley (1985), Sommerville (1985)

#### Fundamentals and principles of specification

Balzer, Goldman (1979), Boehm (1976; 1983), Brooks (1981), IEEE (1983), Kramer (1982), Lehman (1980), Ludewig (1982), Parnas, (1977), Swartout, Balzer (1982), Timm\* (1982)

#### Surveys (articles and books)

Hommel\* (1980), COMPUTER (1982, 1985), IEEE-SE (1977), Ohno (1982), Prentice (1981)

#### Particular specification methods and systems (see also 7.3)

EPOS: Biewald et al.(1979), Lauber, Lempp (1987)  
 ESPRESO: Ludewig (1983); SPADES: Ludewig et al. (1985)  
 HOS/USE.IT: Hamilton, Zeldin (1976)  
 PAISLey: Zave (1982)  
 PSL/PSA: Teichroew Hershey (1977)  
 SADT: Ross (1977, 1985 a,b); SADT/SPECIF: Lissandre et al. (1984)  
 SA: deMarco (1978), Bell (1985); SD: Yourdon, Constantine (1979)  
 SDL: CCITT (1984)  
 SREM: Alford (1985)

#### Prototyping

Boehm, Gray Seewald (1984), Budde et al. (1984)

#### Software Engineering Environments

Howden (1982), Hünke (1981), Osterweil (1981)

### 7.2 References

- Alford, M. (1985): SREM at the age of eight: The distributed computing design system. **IEEE COMPUTER** 18, 4, 36-46.
- Balzer, R., N. Goldman (1979): Principles of good software specification and their implications for specification languages. in **Proceedings of Specification of Reliable Software (SRS)**, IEEE, pp.58-67.
- Bell, R. (1985): Structured analysis aids in micro-computer system design. **EDN**, March 21, 1985, 251-257.
- Biewald, J., P. Göhner, R. Lauber, H. Schelling (1979): EPOS - a specification and design technique for computer controlled real-time automation systems. **4th Intern. Conf. on Software Engin.**, IEEE, pp.245-250.
- Boehm, B.W. (1976): Software Engineering. **IEEE Transactions on Computers**, C-25, pp.1226-1241.

- Boehm, B. W. (1980): **Software Engineering Economics**. Prentice Hall, Englewood Cliffs, N.J.
- Boehm, B.W. (1983): Seven basic principles of Software Engineering. **Journal of System. and Software**, 3, 3-24.
- Boehm, B.W., T.E. Gray, Th. Seewald (1984): Prototyping versus Specifying: A multi-project experiment. **IEEE Trans. on SE, SE-10**, 290-303.
- Brooks, W.D. (1981): Software Technology Payoff: Some statistical evidence. **Journal of Systems and Software**, 2, 3-9.
- Budde, R. K. Kuhlenskamp, L. Mathiassen, H. Züllighoven (eds.) (1984): **Approaches to Prototyping**. Springer-Verlag, Berlin etc.
- COMPUTER (1982): Special issue on application oriented specification. **IEEE COMPUTER** 15, 5 (May 1982), 10-59.
- COMPUTER (1985): Special issue on requirements engineering environments. **IEEE COMPUTER** 18, 4 (April 1985), 9-91.
- deMarco, T. (1978): **Structured Analysis and System Specification**. Yourdon Press, New York.
- Fairley, R. (1985): **Software Engineering Concepts**. McGraw-Hill Book Company, New York usw.
- Hamilton, M., S. Zeldin (1976): Higher Order Software - a methodology for defining software. **IEEE Trans. on SE, SE-2**, 9-32.
- Hommel, G. (Hrsg.) (1980): **Vergleich verschiedener Spezifikationsverfahren, am Beispiel einer Paketverteilanlage**. KfK-PDV 186, Teile 1 und 2, Kernforschungszentrum Karlsruhe, BRD.
- Howden, W. (1982): Contemporary software development environments. **Comm. ACM**, 25, 5, 318-329.
- Hünke, H. (ed.) (1981): **Software Engineering environments**. North Holland Publishing Company, Amsterdam, New York, Oxford.
- IEEE (1983): Standard glossary of software engineering terminology. **IEEE Std 729-1983**.
- IEEE-SE (1977): Special collection on requirements analysis. **IEEE Trans. SE, SE-3**, 2-84.
- Kramer, J. (ed.) (1982): Glossary of terms. **TC on Application Oriented Specification**. Jeffrey Kramer, Imperial College, 180 Queen's Gate, GB - London SW7 2BZ.
- Lauber, R.J., P.R. Lempp (1987): **EPOS Overview**. Report, IRP, Stuttgart University and GPP (see 7.3)
- Lehman, M.M. (1980): Programs, life cycles, and laws of software evolution. **Proc. of the IEEE**, 68, 9, 1060-1076.
- Lissandre, M., P. Lagier, A. Skalli, H. Massié (1984): **SPECIF - A specification assistance system**. Institut de Génie Logiciel, Paris, France.

- Ludewig, J. (1982): Computer aided specification of process control software. **IEEE COMPUTER**, 15, 5, 12-20.
- Ludewig, J. (1983): ESPRESO - a system for process control software specification. **IEEE Trans. on SE**, SE-9, 427-436.
- Ludewig, J., M. Glinz, H.J. Huser, G. Matheis, H. Matheis, M.F. Schmidt (1985): SPADES - A Specification and Design System and its Graphical Interface. **8th Intern. Conf. on Software Engin.**, IEEE, 83-89.
- Ludewig, J. (1987): Practical methods and tools for specification. in A. Kündig, R.E. Bühner, J. Dähler (eds.): **Embedded Systems**. Lecture Notes in Computer Science 284, Springer, Berlin etc., 174-207.
- Ohno, Y. (ed.) (1982): **Requirements Engineering Environments**. Proceedings, North Holland, Amsterdam usw. (partially reprinted in **COMPUTER**, 1985).
- Osterweil, L. (1981): Software environment research: directions for the next five years. **IEEE COMPUTER**, April 1981, 35-43.
- Parnas, D.L. (1977): The use of precise specifications in the development of software. in Gilchrist, B. (ed.): **Information Processing 77**. North Holland Publishing Company, Amsterdam etc., pp.861-867.
- Prentice, D. (1981): An analysis of software development environments. **ACM SIGSOFT Software Engineering Notes**, 6, No.5, 19-27.
- Ross, D.T. (1977): Structured analysis (SA): A language for communicating ideas. **IEEE Trans. on Software Engineering**, SE-3, 16-34.
- Ross, D.T. (1985 a): Applications and extensions of SADT. **IEEE COMPUTER** 18, 4, 25-34.
- Ross, D.T. (1985 b): Douglas Ross talks about Structured Analysis. **IEEE COMPUTER** 18, 7, 80-88.
- Sommerville, I. (1985): **Software Engineering**. Addison-Wesley Publishing Company, London etc., 2nd ed.
- Swartout, W., R. Balzer (1982): On the inevitable intertwining of specification and implementation. **Commun. ACM**, 25, 7, 438-440.
- Teichroew, D., E.A. Hershey III (1977): PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems. **IEEE Trans. Software Eng.**, SE-3, 41-47.
- Timm, M. (1982): **Grundlagen von Anforderungs- und Entwurfs-spezifikationen Im Prozeß der Software-Entwicklung**. GMD-Studien, No. 66.
- Yourdon, E., L.L. Constantine (1979): **Structured Design: Fundamentals of a disciplin of computer programs and systems design**. Prentice Hall Inc., Englewood Cliffs.
- Zave, P. (1982): An operational approach to requirements specification for embedded systems. **IEEE Trans. Software Eng.**, SE-8, 250-269.

### 7.3 Addresses of Vendors

Please note that the following list is rather arbitrary, and far from complete, and *it does not imply any judgement or recommendation !*

**EPOS** (Engineering and Project-management Oriented Support system)

GPP, Kolpingring 18a, D 8024 Oberhaching. Tel. D (+89) 61 10 42 18

**HOS** (Higher Order Software) and **USE.IT**

Higher Order Software, Inc., 2067 Massachusetts Avenue Cambridge, Massachusetts 02140, USA, Tel. USA (617) 661-8900

**IDE Structured Analysis and Design Tools**

Interactive Development Environments, 150 Fourth Street, Suite 210, San Francisco, California 94103, Tel. USA (415) 543-0900

**Information Engineering Workbench/Workstation**

KnowledgeWare, Inc., 3340 Peachtree Rd., N.E., Atlanta, GA 30326, Tel. USA (404) 231-8575

**MASCOT** (Modular Approach to Softw. Construction, Operation, and Test)

MASCOT Suppliers Association, c/o Computing Standards Section, Room L303, Royal Signals and Radar Establishment, St. Andrews Rd., Malvern, Worcestershire, WR14 3PS, GB

**Perspective (Includes a tool for MASCOT)**

Software Technology Centre, System Designers Ltd., Systems House, 1 Pembroke Broadway, Camberley, Surrey GU15 3XH Great Britain, Tel. GB (+276) 62244

**ProMod** (Projektmodell)

GEI, Pascalstr. 14, D-5100 Aachen, Tel. D (+2408) 130

**PSL/PSA** (Problem Statement Language/Analyzer) and related systems

META-systems, 315 E. Eisenhower Pkwy., Suite 200, Ann Arbor, MI 48104, USA; Tel. USA (313) 663-6027

**Softool** (Softw. Managem., Developm., Maintenance, and Conversion Tools)

SOFTOOL Co., 340 S. Kellogg Av., Goleta, CA 93117

**SPECIF** (Specification System) for SADT-Diagrams

Institut de Génie Logiciel (IGL), 39 rue de la Chaussée d'Antin, F-75009 Paris, France; Tel. F (+33) 1 281 41 33

**Teamwork**

Cadre Technologies Inc., 222 Richmond Street, Providence, RI 02903; Tel. USA (401) 351-5950

**TekCASE** (Analyst/RT, Designer, Auditor, Table Editor)

Tektronix Inc, CASE Division, P.O.Box 14752, Portland, Oregon 97214, Tel. USA (503) 627-7111

**PSIttools: NET** (editor and simulator for extended Petri-Nets) and **BOIE**

(tree-oriented development tool) PSI GmbH, FB Software Engineering, Kurfürstendamm 67, D 1000 Berlin 15, Tel. D (+30) 88 42 30