Specification Techniques for Real-Time Systems

J. LUDEWIG and H. MATHEIS

ETH-Zentrum, Institut für Informatik, CH-8092 Zurich, Switzerland

This paper is a course on Specification. Since it is based on experiences in the field of Software Engineering, it applies primarily to Software Specifications. Many observations and reports indicate, however, that, from specification aspects, there is not much difference between information processing systems in general and software in particular. Therefore, most of this course applies also to System Specification. There are methods, languages, and tools for writing specifications. In this paper, we concentrate on methods and languages.

We start with the definitions of a few fundamental terms and of the qualities of specifications. In the main part of the paper, we present four specification methods together with their underlying languages, namely Structured Analysis, SADT, PSL, and RSL. A few sample specifications written in these languages are given in order to convey an optical impression of each language. The paper ends with some general conclusions and a list of references.

Keywords: Software Engineering, specification methods, specification languages.

J. Ludewig was educated at TU (Technical University) Hannover (Electrical Engineering) and TU München (Computer Science), where he also received his PhD. He worked six years at Nuclear Research Center, Karlsruhe, FRG, and five years at Brown Boveri Research Center, Baden, Switzerland. Since January 1986, he is an associated Professor of Computer Science at the Swiss Federal Institute of Technology (ETH) Zürich. His research interest is in Software Engineering.

North-Holland
Computer Standards & Interfaces 6 (1987) 115–133

1. Introduction

This is a course on *Specification*. Since it is based on experiences in the field of Software Engineering, it applies primarily to *Software Specifications*. Many observations and reports indicate, however, that, from specification aspects, there is not much difference between information processing systems in general and software in particular. Therefore, most of this course applies also to *System Specification*.

In the field of System Specification in general and Software Specification in particular, one distinguishes three components, namely methods, languages, and tools. In this paper we concentrate on *methods* and *languages*. The primary goal is to show some typical features of methods and languages for Specification rather than to describe them in detail.

The first section starts with a few fundamental terms of interest in order to motivate the use of specification. Based on the qualities of specifications, the properties useful for specification and the requirements for specification systems are summarized in section 2. In section 3 four selected specification languages are outlined together with their underlying methods. A few examples are given in order to convey an optical impression of each language. Section 4 addresses management aspects. The paper ends with some general conclusions and a list of references.

H. Matheis received the diploma (MS) in Computer Science from University of Kaiserslautern, FRG, in 1984. For two years, he worked at Brown Boveri Research Center, Baden, Switzerland. Currently, he is working with J. Ludewig, aiming at a doctorate.

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.

However, it came out that all the established life cycle models lack end-user involvement. Therefore, new ideas arose (e.g. prototyping) to overcome those defencies. They led to a new view of the life cycle [3,6]. Newer life cycle models also envisage to support activities in the areas of project management, quality assurance, and configuration management, which cover the whole life cycle.

2.2. Cost Distribution

About 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) [9]. Therefore, every attempt to reduce the high cost of software has to focus on maintenance. Note that software maintenance and technical maintenance have different meanings. While software maintenance means correction and modification of software (e.g. based on user requirements), technical maintenance (e.g. maintenance of cars) means the process of replacing attrited parts. That is, it attempts to *repair* the old state of the product.

Now, what are the subgoals to attain the reduction of maintenance?

The need for correction and modification must be reduced as well as the total volume of software (by integration of standard components or old software).

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

Unfortunately, there are no precise figures indicating that more investment for specification implies less effort for maintenance. Nevertheless, based on our experience we estimate the effort per phase of the life cycle as shown in fig. 1.

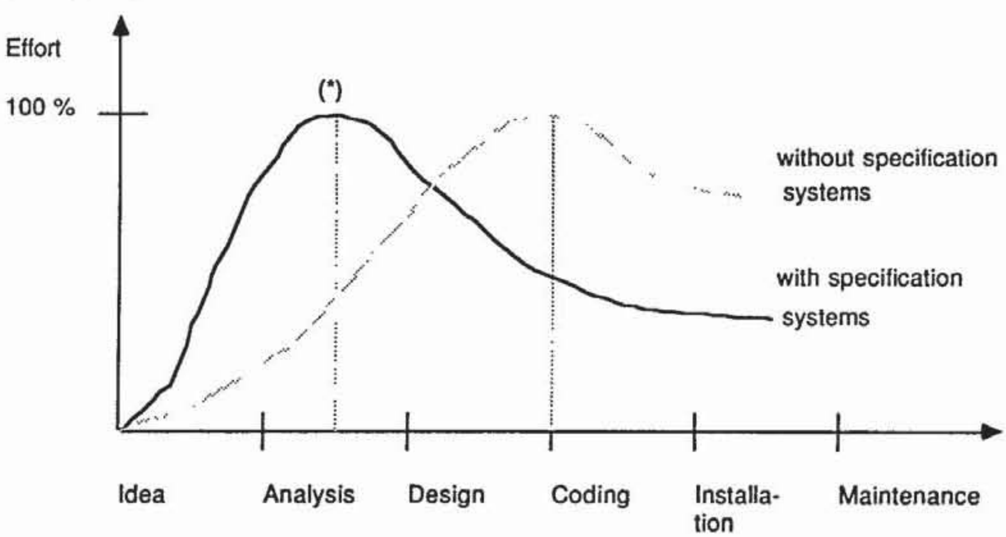


Fig. 1. The Effort per Phase with Specification Systems and Without Them.

If a specification system is used, the critical point (marked with (*) in Figure 1) arises. At this time nobody in the project is satisfied with the specification system and sceptics will say: "Specification systems are useless and only delay the project. They contribute nothing to a systematic development process."

This psychological aspect has to be considered very carefully. If a specification system is to be introduced, the project management must be well prepared.

2.3. Terminology

2.3.1. Specification

To date, we have not achieved a stable and well recognized terminology in Software Engineering. In the following, we use a simple, pragmatic definition of "specification" [14]:

"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. Specification is frequently used to mean functional specification which contains both requirements and design aspects. This form of use is imprecise."

Many more relevant terms are defined by the IEEE [13] and by Hesse et al. [12].

The reason why people could not agree with a general definition of specification might be a specification-immanent problem expressed by "properties of interest" in the definition above. What *are* the properties of interest? This question can not be answered objectively. It primarily depends on the specifier's and the user's subjective point of view.

2.3.2. 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 certain syntactical representation. *Tools* check, store, and transform such statements.

All three are strongly interrelated by the abstract concepts of the (specification-) system. Note

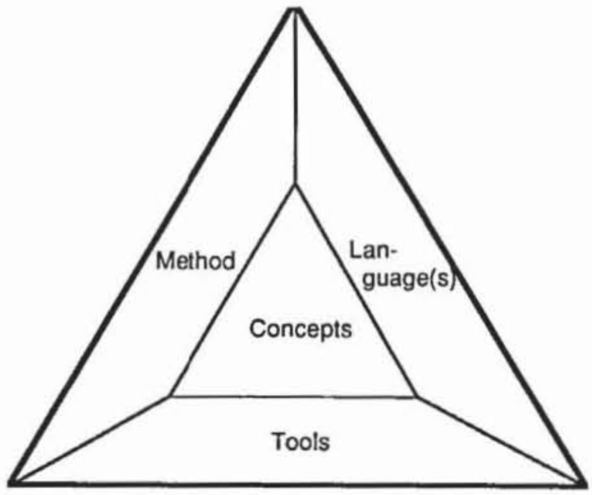


Fig. 2. System Triangle.

that the term "methodology" means "science of methods", though it is often misused for "method". Figure 2 shows the system triangle.

2.3.3. Levels of Formality

There are languages of various formality. For our purposes, we distinguish four levels (see Table 1).

Table 1

	Syntax	Semantics	Examples
informal	not (precisely) defined	not (precisely) defined	natural languages
formatted	restricted (by forms)	not (precisely) defined	forms
semi-formal	defined	partially defined	pseudo-code
formal	defined	defined	(modern) programming languages

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 modifiable

- readable and concise (in order to ease the communication between user and analyst)
- implementable (i.e. it should ease design and implementation)
- verifyable (i.e. there should exist a procedure to check whether or not the product complies with its specifications)
- validatable (i.e. there should be a mechanism to ensure that the specification really reflects the user's intention)
- 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 *implementable,* but not *readable* for anybody not very familiar with algebraic specifications. Another example concerns *consistency* and *correctness*. Usually, user-defined requirements are not consistent, but each of them may be correct.

Another remark concerns traceability. If changes have to be done, theory requires that first of all the *specification* is changed. Unfortunately, this does not work in practice where people only alter the *corresponding program*, leaving the specification unchanged.

The first four qualities for specifications can be considered either from a *syntactical* point of view, or from a *semantical* point of view. Syntactically, it is possible to check whether a specification is correct, complete, consistent and unambiguous. Unfortunately, this is not true for the semantics. Even if the semantics of the specification language are completely defined, it is neither possible to prove the semantical completeness, nor the semantical correctness. 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 provable 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 defined (i.e. provide a filler that indicates the lack of information).
- They should exist only in one single copy ("single source concept").
- There should be tools for automatic checks and transformations between different languages.
- They 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).
- They should support the processing of fuzzy logic, because the sharp distinction between the values true and false is not always sensible and possible.

3.3. Specification Systems Requirements

First, it is necessary to say what we denote with the term "specification system". In our terminology, a specification system comprises languages, methods, and tools supporting the activities before programming.

Considering the properties stated above, we can derive the requirements of specification systems:

- Database as central information repository
- Semi-formal specification language
- Several representations, supported by tools.

Since systems are developed by several people, and usually exist in several versions and variants at the same time, we must also provide

- multiuser operation of tools
- automatic management of versions and variants.

3.4. Influence of Semi-Formal Specification Systems on the Working Technique

Generally speaking, semi-formal specification systems imply that the problem is formalized not in one but in two steps, starting when the system is specified and designed. Without using a specification system, the formalization process is almost exclusively concentrated on the implementation phase. See Fig. 3.

Further on, several changes in the working techniques can be noticed:

- Division of problem into smaller steps
- Separation of setting a task and solving a problem
- Better communication before implementation
- Better documentation and easier modification.

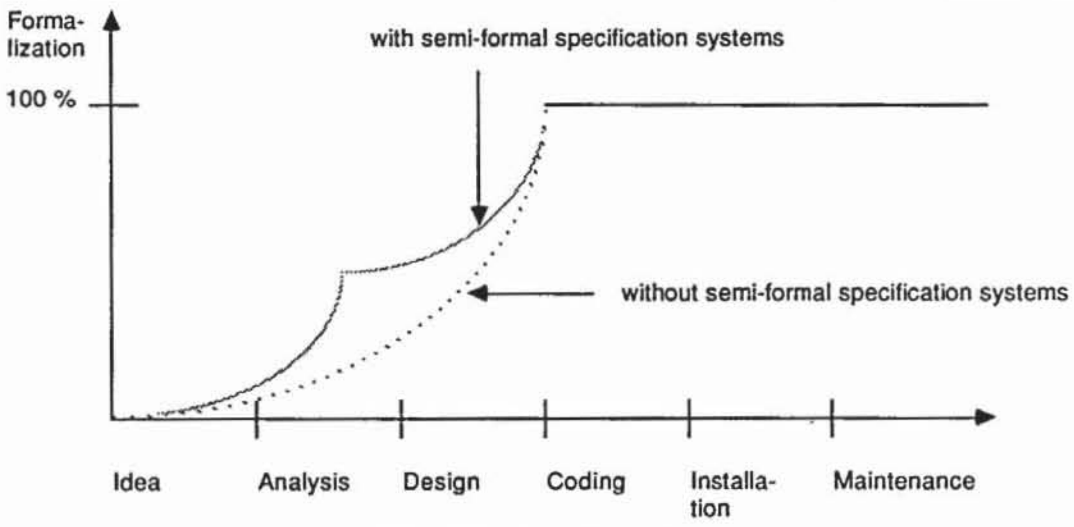


Fig. 3. Degree of Formalization per Phase with Semi-formal Specification System and Without One.

3.5. General Structure of a Specification System

Compared with the development of programming languages, the development of specification systems is at the very beginning. People assume that we are about in 1955 when Fortran appeared.

A rough classification of specification systems distinguishes two general classes of specification systems. The first one contains *tailored systems* which are not adaptable to user needs and requirements (e.g. proMod and PRADOS; see list of references). The second one is more like a *tool-box* containing more or less independent components (e.g. mbp-tool-system and S/E/TEC both described by Balzert [19]). In the latter system the individual tools can be adapted to the user's individual needs. There are several other possibilities to characterize and classify specification systems. One of them is the distinction between systems supporting either one special method per phase (e.g. proMod supporting Structured Analysis) or different methods per phase (sometimes also no method as in the case of EPOS, see list of references).

In the sequel, we summarize a few features useful in specification systems:

Methods

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

Languages

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

Tools

- Multi-user Database-System
- Tools for checking, retrieval and selection.

Abstract concepts

- Life cycle model
- Stepwise refinement.

4. Specification Languages and Methods: Examples

In this section, 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 languages and methods in detail.

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.

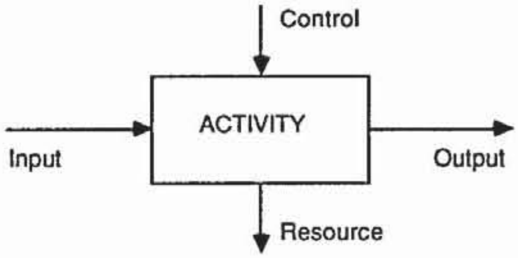


Fig. 4. SADT-Box.

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 [38], 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 and data in an actigram is also comprised in some datagram.

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. For the reason of complexity the language restricts the number of boxes per SADT-diagram to seven.

The Figure 4 shows a SADT-box with its typical components.

The three actigrams of Figs. 5–7 show an activity ("assist SADT *USERS*") at three different levels of refinement. Note that the last actigram refines an activity ("*CREATE KITS*") of the second diagram (Source: [36] from IGL, Paris).

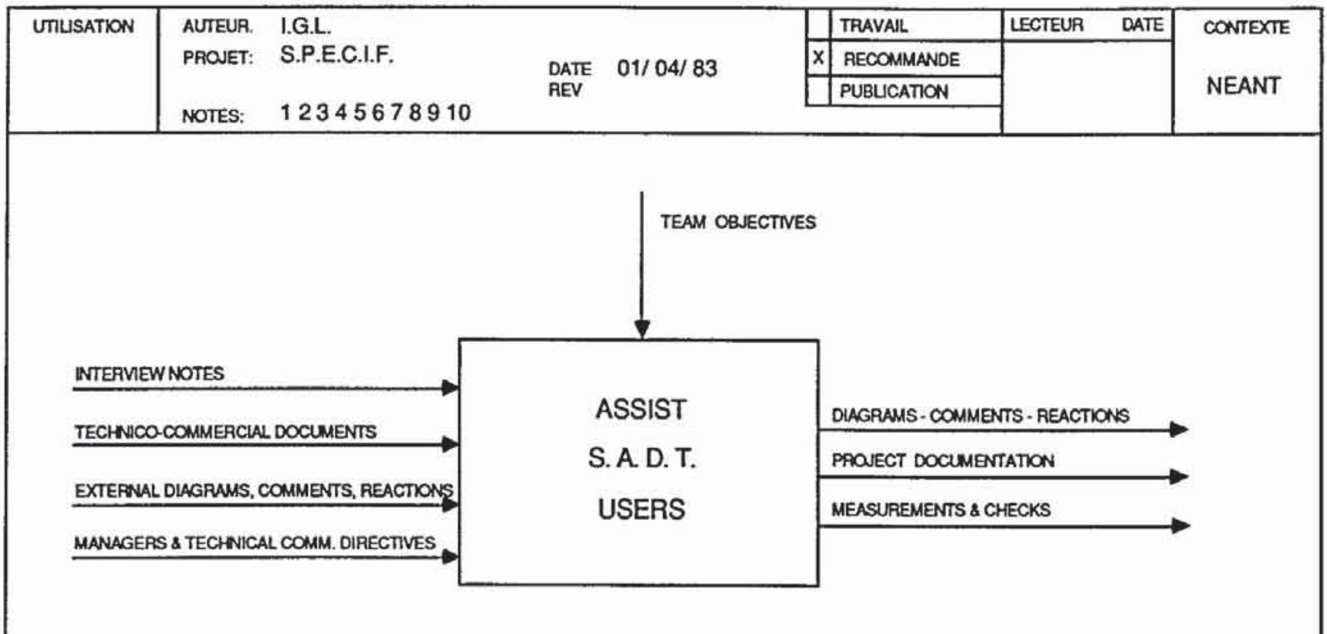


Fig. 5. Top-level Actigram of "ASSIST S.A.D.T. USERS"

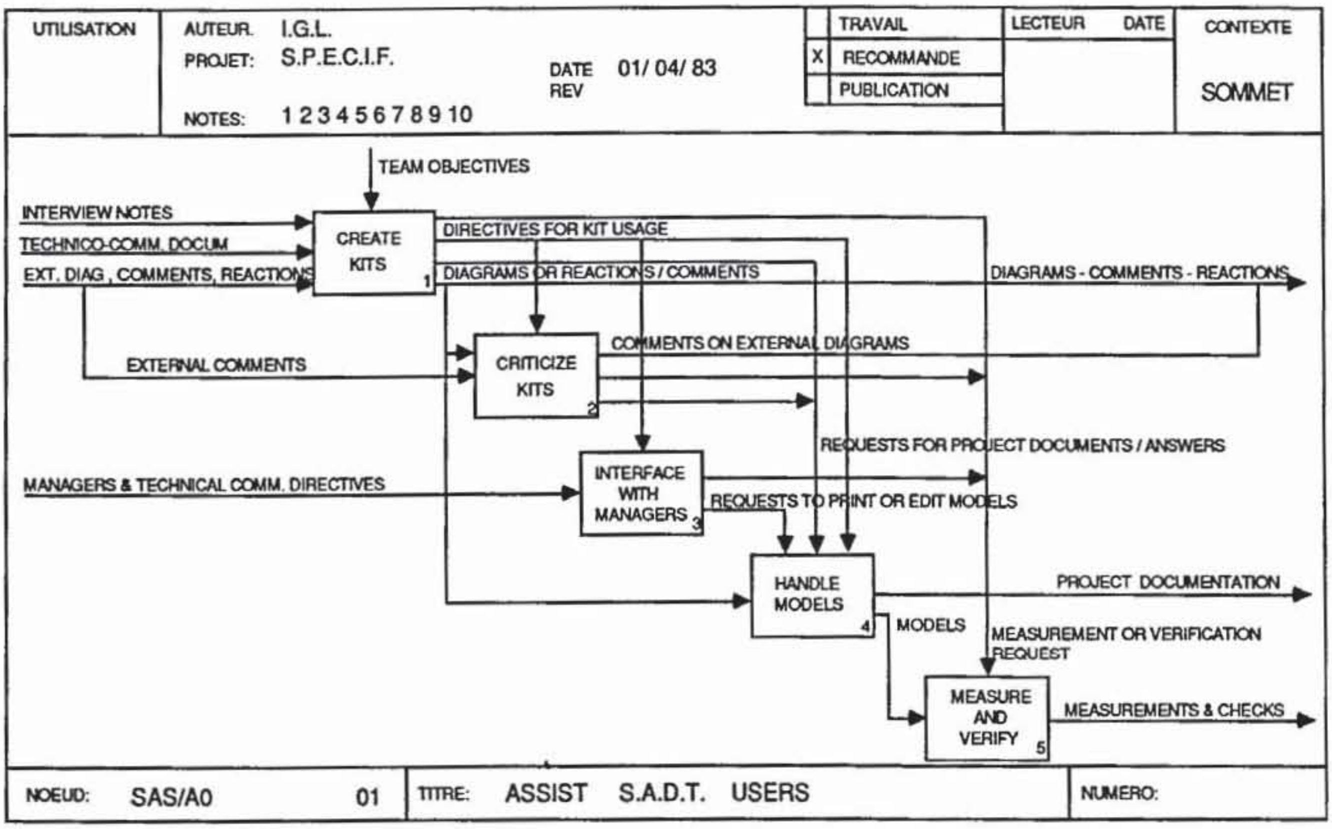


Fig. 6. Detailed Actigram of "ASSIST S.A.D.T. USERS"

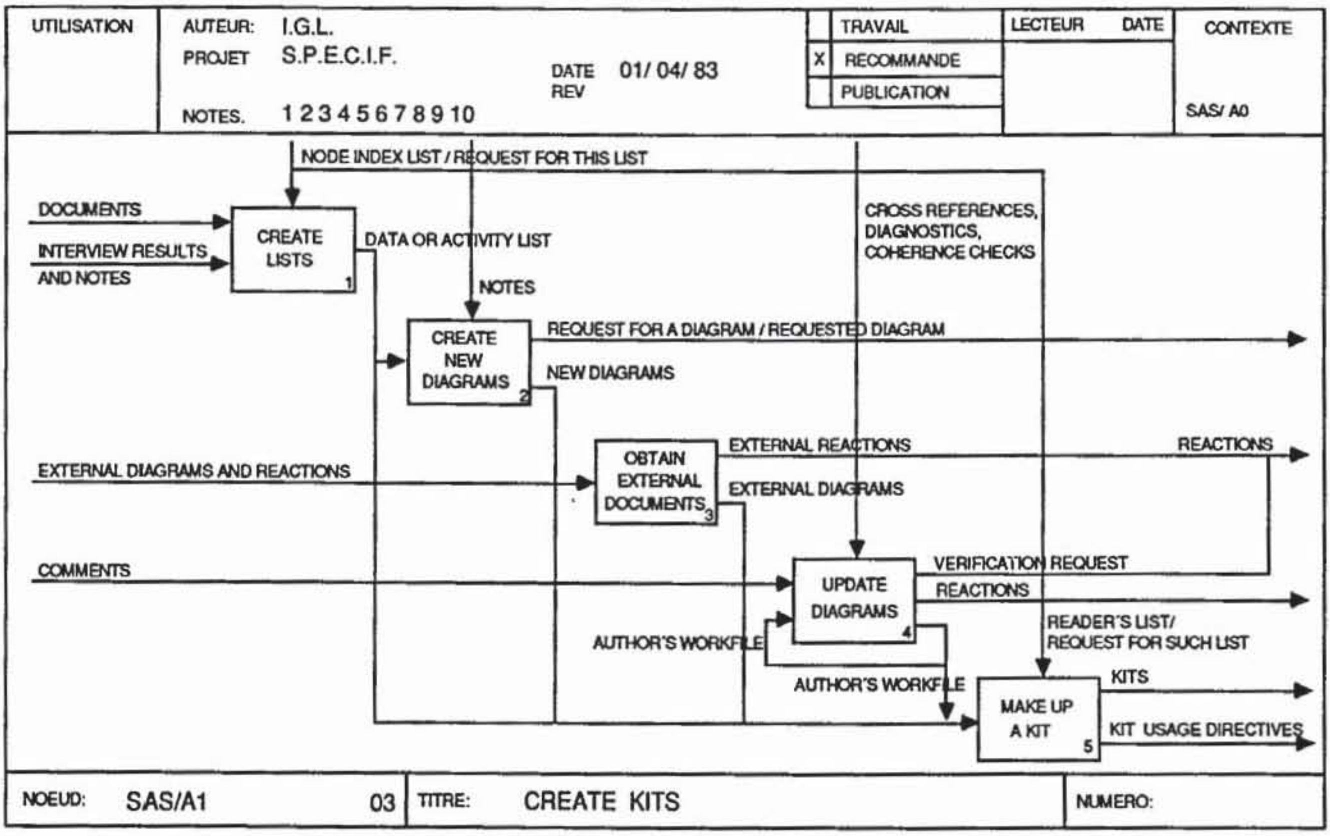


Fig. 7. Refinement of Activity "CREATE KITS"

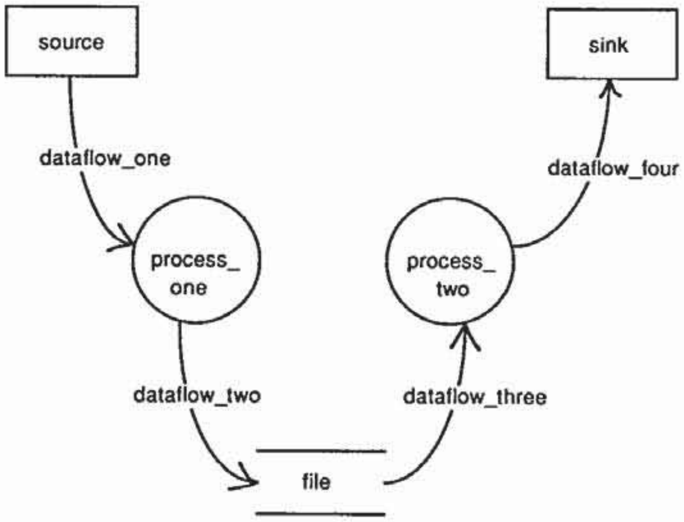


Fig. 8. Sample SA-DFD.

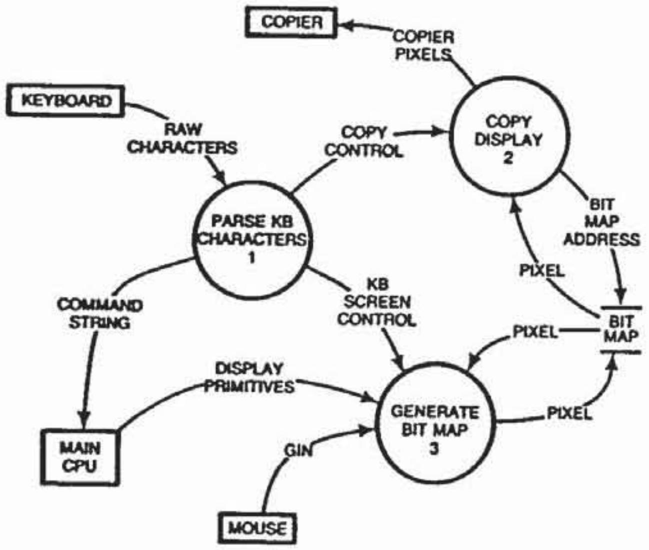
4.2. Structured Analysis (SA)

SA was developed by Yourdon and others (see [40]). Although the name is very similar to SADT, only the *data flow* as the central principle is common to both. It is used for analysis and both coarse and detailed design.

4.2.1. The Method

The method allows the user to model a system with *data-flow diagrams* (DFD's) consisting of *data*, and *processes* transforming the data. In other words, DFD's 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. In order to refine the structure of data there exist a *data dictionary* (DD). SA proposes a stepwise decomposition of data flow diagrams so that each process in the parent DFD is broken into several child DFD's. Consequently, several levels of DFD's emerge.

Now, let us have a closer look at SA. SA proposes two major steps. The first one is to develop a so-called *context diagram* (see Figure 9a) showing the system as connected to its

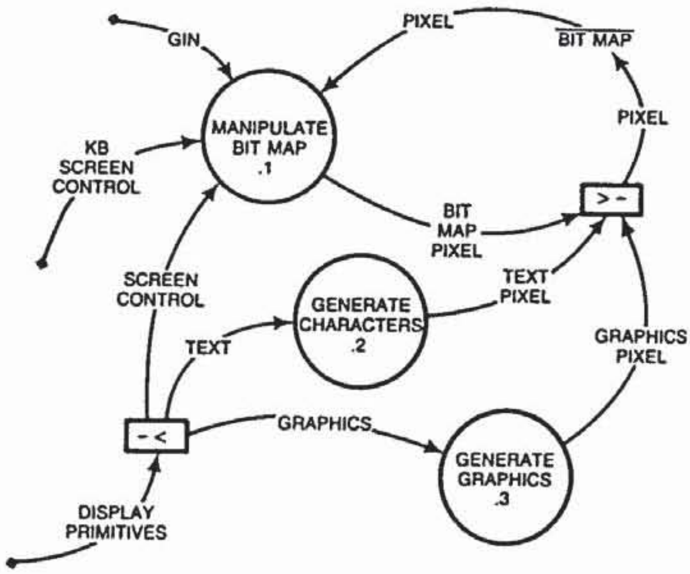


(a)

```
COPIER_PIXEL = PIXEL
BIT_MAP_ADDRESS = INTEGER
PIXEL = LOGICAL
TEXT = ASCII_CHAR
GRAPHICS = [POLYLINE | POLYMARKER | AREA FILL | GDP]
SCREEN_CONTROL = [SCROLL | ERASE | REVERSE | HORIZ_SCROLL]
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
```

(b)

Fig. 9. a) DFD for a Display Controller; b) DD for 9a.



(a)

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

(b)

Fig. 10. a) DFD for Generate Bit Map from Fig. 9a; b) Minispec for 10a.

environment. Hereby, the user defines the interface connecting the system and the environment in terms of sources and sinks of the environment, processes, data flows, and files. Please 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". This means, he describes each process of a DFD in more and more detail until he reaches processes which are atomic. Then, the user writes minispecs demonstrating the algorithmic structure of these atomic processes. Also, a data dictionary is installed containing the *structure* of the data. SA also gives a proposal how to name the items (processes, dataflows, files) in order to express meanings most clearly.

4.2.2. The Language

The sources and sinks belonging to the environment of the system to be described are shown as *boxes* on a data-flow diagram. Other symbols are *circles* representing processes, *arrows* representing data flows, and *bars* representing files (see Fig. 8).

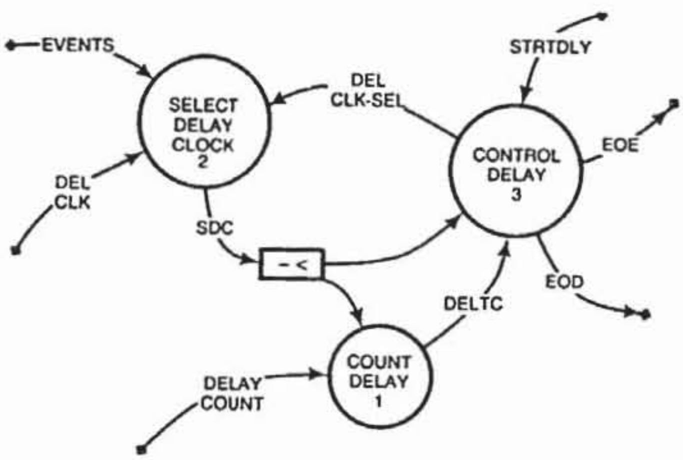


Fig. 12. DFD of Count Delays (from Fig. 11).

Please note that the first time a file is referenced in a DFD *two* bars are used (see Figure 9a, file "Bit Map") while further references to this file (in other DFD's) are denoted by *one* bar (see Figure 10a, 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 in Figs. 9–14 were taken

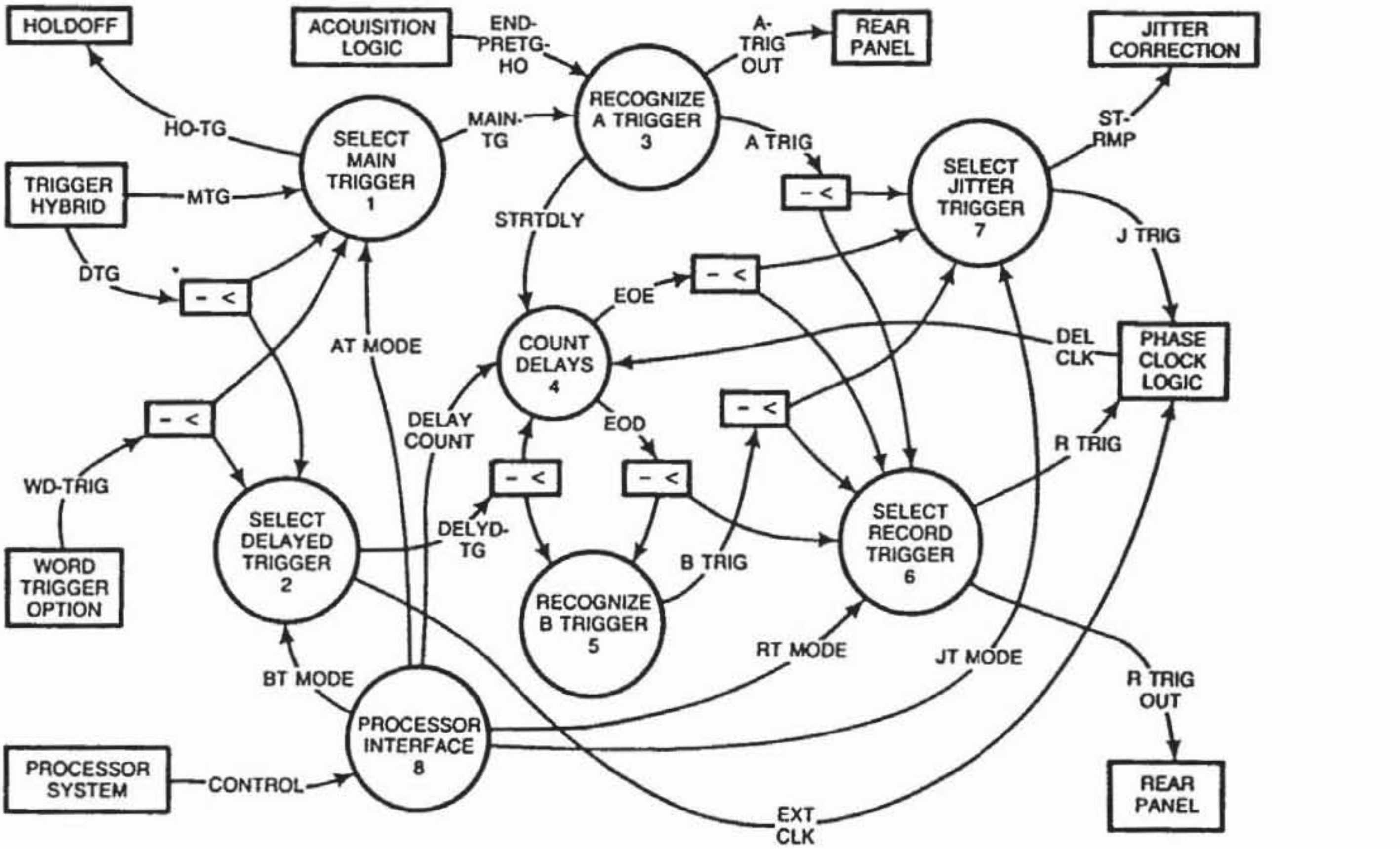


Fig. 11. Top Level DFD of a Trigger Gate Array.

```
MINISPEC 4.3

CIRCUIT ELEMENTS: 2FF2, 2FF3, 2FF4, 2G4, 2G5
OVERVIEW: THIS CIRCUIT IS A 3-FLIP-FLOP STATE MACHINE. 2FF2
          CONTROLS THE START OF COUNTING DELAY, 2FF3 SETS
          AT THE END OF EVENTS COUNT, AND 2FF4 SETS AT THE
          END OF THE TIME-DELAY COUNT. SPECIAL-CASE COUNTS
          OF NO EVENTS AND 1 EVENT ARE CONTROLLED BY LEVEL
          INPUTS SET BY THE PROCESSOR. THE INITIAL STATE
          OCCURS WHEN THE PROCESSOR STROBES RSTACQ. THIS
          CLEARS 2FF2, WHOSE QBAR OUTPUT CLEARS 2FF4. 2FF3
          IS CLEARED BY THE A TRIGGER FLIP-FLOP 1FF1. THE
          FIRST DCLK AFTER A TRIGGER WILL SET 2FF2 TO ENABLE
          THE DELAY COUNTER. IF ONEVNT = 1, 2FF3 WILL ALSO
          SET AT THIS TIME. DCLKS WILL BE COUNTED UNTIL DELTC
          = 1, CAUSING 2FF3 TO SET. WHEN EOE = 1, THE SELECT
          DELAY CLOCK LOGIC SWITCHES TO COUNTING DELAY BY
          TIME. THIS WILL CONTINUE UNTIL THE NEXT OCCUR-
          RENCE OF DELTC = 1, WHEN EOD = 1 WILL OCCUR. THE
          STATE MACHINE REMAINS IN THIS STATE UNTIL THE NEXT
LOGIC:    RSTACQ.
  ALL FLIP-FLOPS ARE RESET ASYNCHRONOUSLY BY PROCESSOR
  ACTION
    SET STRTDEL = 0 WHEN RSTACQ = 1
    SET EOD = 0 WHEN STRTDELB = 1
    SET EOE = 0 WHEN ATB = 1
  ALL FLIP-FLOPS WILL SET ON CONDITION ON THE RISING EDGE OF
  DCLK
    SET STRTDEL = 1 WHEN AT = 1 (RESETS ARE NOW REMOVED
                                 FROM 2FF3, 2FF4)
    SET EOE(N+1) = ONEVNT + DELTC + EOE + NOEVNTS
    SET EOD(N+1) = (EVDN + EOD)*(DELTC + EOD) = EVDN*DELTC
     + EOD
```

Fig. 13. Minispec of Control Delay (from Fig. 12).

from a paper on the Tektronix-tool [33]. They show data-flow diagrams, together with minispecs and information stored in the data dictionary.

4.3. Problem Statement Language (PSL)

PSL was developed at the University of Michigan by the ISDOS-project (*I*nformation *S*ystem *D*esign and *O*ptimization *S*ystem) in the 1970s. PSL primarily supports requirements analysis and documentation.

4.3.1. The Method

PSL is based on the *entity-relationship* approach first defined by Chen [21] 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.

Table 2

Entity-classes:	
REAL WORLD	
ENTITY	real world objects which are out of the 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

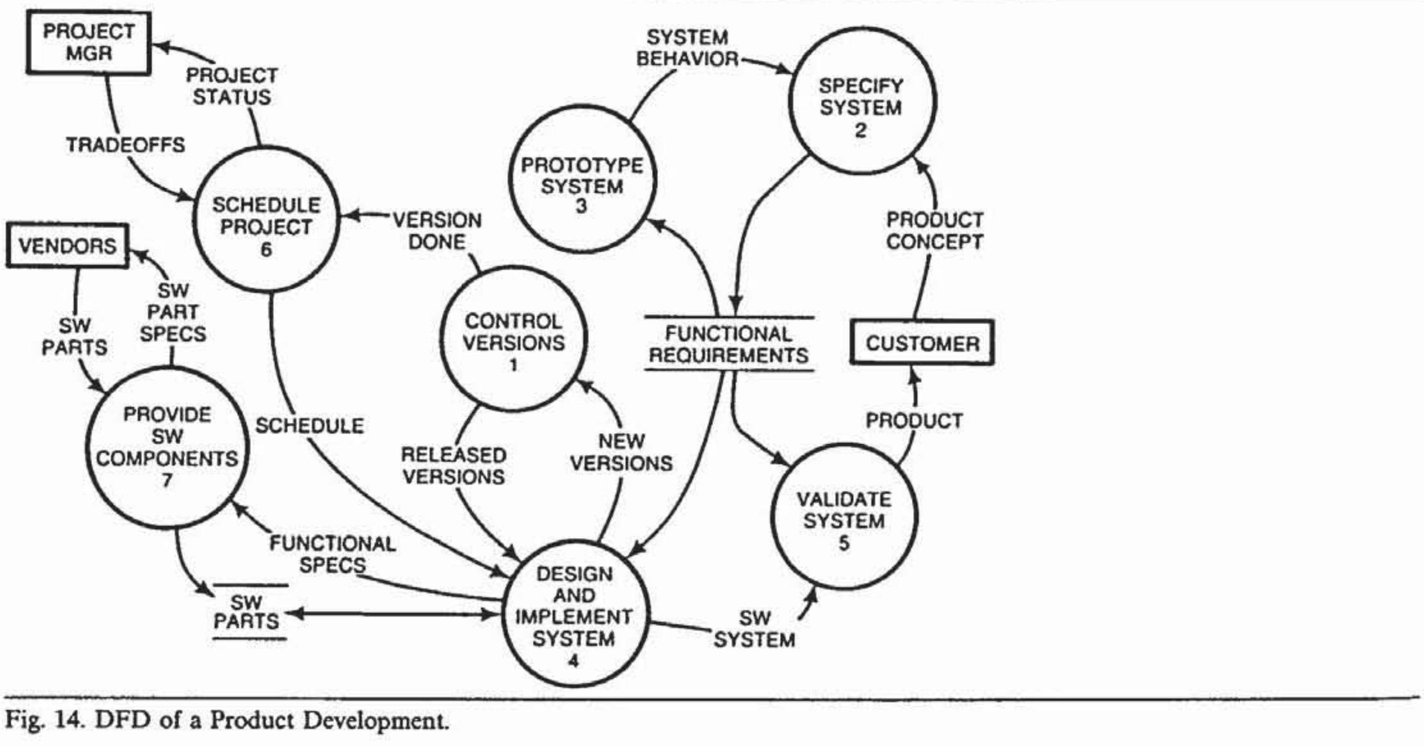


Fig. 14. DFD of a Product Development.

```
PSA Version A5.2R2M                         Jul 23, 1983  20:05:19      Page
                              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=ON
    NOEXPLANATION

   LINE  S T M T

      >/*  This is a set of PSL statements to define user views */
      >
      >/*  Here is the global users' view  */
      >
      >DEF ENTITY           Userviews;
      >   TKEY              'Global';
      >   SUBPARTS ARE      User-View-1,
      >                     User-View-2,
      >                     User-View-3,
      >                     User-View-4,
      >                     User-View-5,
      >                     User-View-6,
      >                     User-View-7;
      >   DESC;
      >This is a global view of a ship company.;
      >
      >
      >/*  ELEMENTs are declared */
      >
      >DEF ELE              Vessel,Cargo-Volume,Details,Port,Date-of-Arrival,
      >                     Date-of-Departure,Consignee,Container#,Size,
      >                     Shipping-Agent,Waybill#,
      >                     Delivery-Date,Contents,
      >                     Handling-Instructions;
      >
      >
      >/*  Here is the local users' view  */
      >
      >DEF ENTITY           User-View-1;
      >   TKEY              'V1';
      >   CSTS OF           View1-Ship;
      >   ATTR ARE          FREQUENCY-IS         100,
      >                     TIMING-REQUIREMENT   25;
      >   RPD IS            'E. Basar';
      >   DESC;
      >Information is stored about each ship, including
      >the volume of its cargo storage capacity.;
      >
      >
      >DEF ENTITY           User-View-2;
      >   TKEY              'V2';
      >   CSTS OF           View2-Ship,
      >                     View2-Ship-Port,
      >                     View2-Port;
      >   ATTR ARE          FREQUENCY-IS         100,
      >                     TIMING-REQUIREMENT   50;
      >   RPD IS            'E. Basar';
      >   DESC;
```

Fig. 15. PSL-Input Source Listing Page (No. 1).

```
PSA Version A5.2R2M                   Jul 23, 1983  20:05:19     Page       2
                            FSL/PSA - ISDOS - VM/CMS

                            IPSL Input Source Listing

  LINE  S T M T

      >A ship stops at many ports and it is necessary to
      >print out its itinerary.:
      >
      >
      >DEF ENTITY          User-View-3;
      >   TKEY             'V3';
      >   CSTS OF          View3-Consignee,
      >                    View3-Port,
      >                    View3-Ship,
      >                    View3-Container;
      >   ATTR ARE         FREQUENCY-IS          25,
      >                    TIMING-REQUIREMENT     7;
      >   RPD IS           'E. Basar';
      >   DESC;
      >Persons who ship goods are referred to as consignees.
      >Their goods must be crated or stored in shipping containers.
      >These are given a container identification number. A list
      >can be obtained, when requested, of what containers have
      >been sent by a consignee.;
      >
      >
      >DEF ENTITY          User-View-4;
      >   TKEY             'V4';
      >   CSTS OF          View4-Agent,
      >                    View4-Port,
      >                    View4-Container;
      >   ATTR ARE         FREQUENCY-IS         110,
      >                    TIMING-REQUIREMENT    75;
      >   RPD IS           'Chiang Wan';
      >   DESC;
      >The shipments are all handled by shipping agents. A
      >shipping-agent report must be generated, listing all
      >the containers that a given agent is handling and giving
      >their waybill numbers.;
      >
      >
      >DEF ENTITY          User-View-5;
      >   TKEY             'V5';
      >   CSTS OF          View5-Waybill,
      >                    View5-Port,
      >                    View5-Ship,
      >                    View5-Container;
      >   ATTR ARE         FREQUENCY-IS         100,
      >                    TIMING-REQUIREMENT    50;
      >   DESC;
      >A waybill related to a shipment of goods between two
      >ports on a specified vessel. The shipment may consist
      >of one or more containers.;
      >
      >
      >DEF ENTITY          User-View-6;
      >   TKEY             'V6';
      >   CSTS OF          View6-Ship,
```

Fig. 16. PSL-Input Source Listing Page (No. 2).

```
PSA Version A5.2R2M                        Jul 23, 1983  20:05:19      Page      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
    8 (GROUP)         3     View2-Ship (M-1)
    9 (ELEMENT)        4       Vessel
   10 (GROUP)         3     View2-Port (M-1)
   11 (ELEMENT)        4       Port
   12 (GROUP)        2   View2-Port
   13 (ELEMENT)       3     Port
   3* (ENTITY)      1 User-View-3
    1 (GROUP)        2   View3-Consignee
    2 (ELEMENT)       3     Consignee
    3 (GROUP)         3     View3-Container (M)
    4 (ELEMENT)        4       Container#
    5 (ELEMENT)        4       Date-of-Arrival
    6 (ELEMENT)        4       Shipping-Agent
    7 (GROUP)          4       View3-Port (1)
    8 (ELEMENT)         5         Port
    9 (GROUP)          4       View3-Ship (M-1)
   10 (ELEMENT)         5         Vessel
   11 (GROUP)        2   View3-Port
   12 (ELEMENT)       3     Port
   13 (GROUP)        2   View3-Ship
   14 (ELEMENT)       3     Vessel
   15 (GROUP)        2   View3-Container
   16 (ELEMENT)       3     Container#
   17 (ELEMENT)       3     Date-of-Arrival
   18 (ELEMENT)       3     Shipping-Agent
   19 (GROUP)         3     View3-Port (1)
   20 (ELEMENT)        4       Port
   21 (GROUP)         3     View3-Ship (M-1)
   22 (ELEMENT)        4       Vessel
   4* (ENTITY)      1 User-View-4
    1 (GROUP)        2   View4-Agent
    2 (ELEMENT)       3     Shipping-Agent
    3 (GROUP)         3     View4-Container (M)
    4 (ELEMENT)        4       Container#
    5 (ELEMENT)        4       Waybill#
    6 (ELEMENT)        4       Consignee
    7 (ELEMENT)        4       Vessel
```

Fig. 17. Report Showing a Tree-Structure by Indentation.

4.3.2. The Language

Different form 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 given in Table 2.

Figs. 15 and 16 show two pages of PSL-input source listing; the specification describes cargo-vessels and their organizational environment.

Two reports follow in Figs. 17 and 18. The first one shows a tree-structure (the hierarchical content-relation) by indentation. The second one shows part of the same information in a table. (Source: Material distributed by ISDOS, now META-Systems, Ann Arbor, Michigan).

4.4. Software Requirements Engineering Methodology (SREM)

SREM was developed by TRW since 1975. It supports the earlier phases (analysis, definition, verification, and validation of requirements) of the software development process and primarily addresses real-time applications.

4.4.1. The Method

SREM possesses two important features not present in other methods or languages for specification. First, it allows the stepwise development of specifications beginning with informal descriptions, and proceeding towards a specification

```
Version A5.2R2M                       Jul 23, 1983  20:05:19     Page      29
                        PSL/PSA - ISDOS - VM/CMS

                        Contents Comparison Report

Basic Contents Matrix

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 ---------- | *     *  |     *     |    * *|
-------------------------+---------+-----------+-------+
```

Fig. 18. Report Showing Part of the Same Information (of Fig. 17) in a Table.

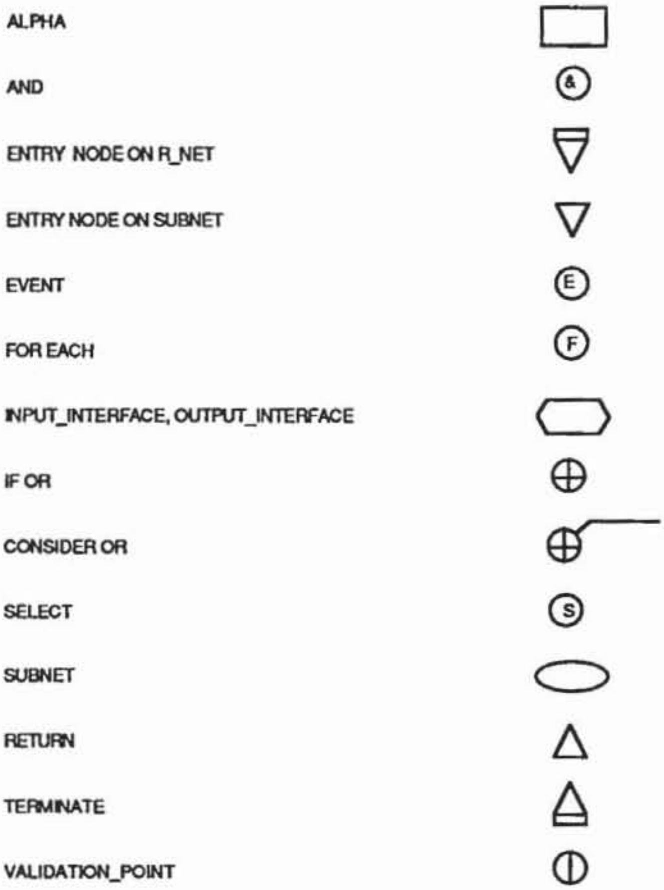


Fig. 19. Symbols of R-Nets.

which is more and more *formal*. Second, data on *performance* of a system can be formally included in the specification.

The method dictates the following eight steps:

(1) identifying the interface between the system and the environment and describing the data flows and the data-processing units inside the system;
(2) outlining the very first description of the system using either the graphical R-Net formalism (R-Net means requirements-net and is a stimulus-response network) or the linear language RSL (requirements statement language);
(3) completion and improvement of the RSL-specification developed so far; implementation of Pascal-procedures for so called *ALPHAs* (active components) in order to be able to simulate the *ALPHAs* (see step 5);
(4) addition of management informations, e.g. deadlines, milestones, needed tools, etc.;
(5) proof of syntactical correctness and simulation of dynamic behaviour; activation and evaluation of so called validation-points (serve as control points for performance analysis, e.g. response time) included in the system before;
(6) check if every requirement is fulfilled by the design;
(7) completion of validation conditions and refinement of functional validations developed in step 5;
(8) analytical feasibility study in order to prove that the current design is useful as a basis for a technical realization.

4.4.2. *The Language*

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

R-Nets are stimulus-response networks describ-

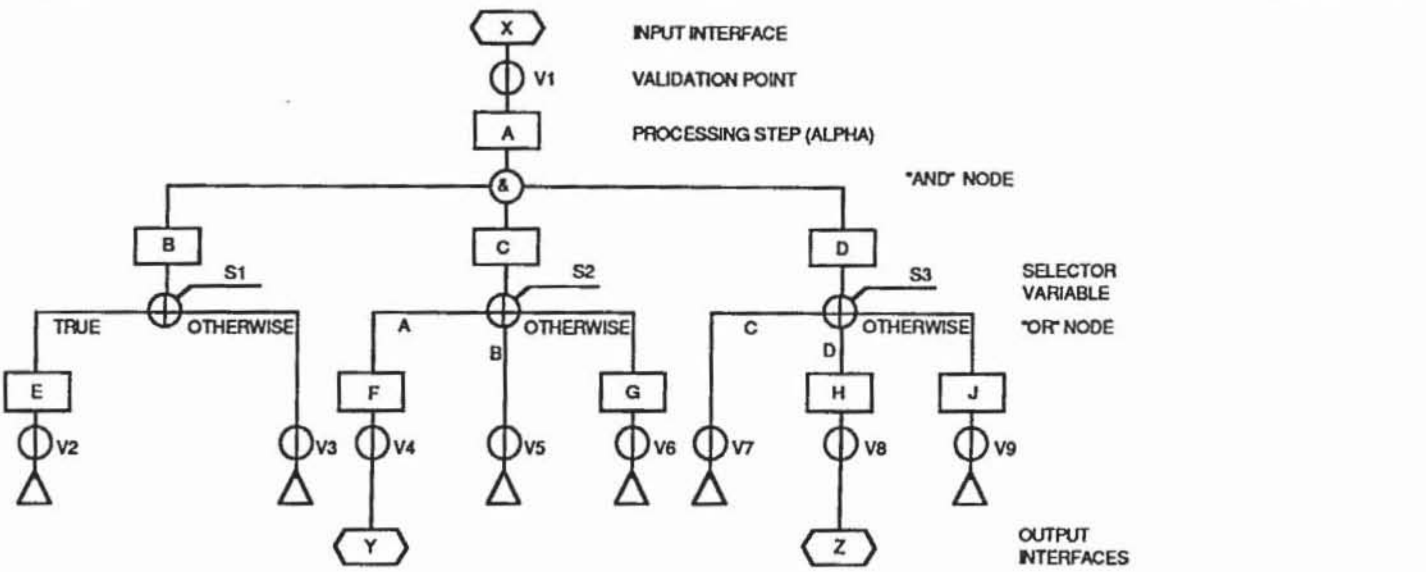


Fig. 20. Sample R-Net.

```
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.
```

Fig. 21a. RSL-Representation of Sample R-Net.

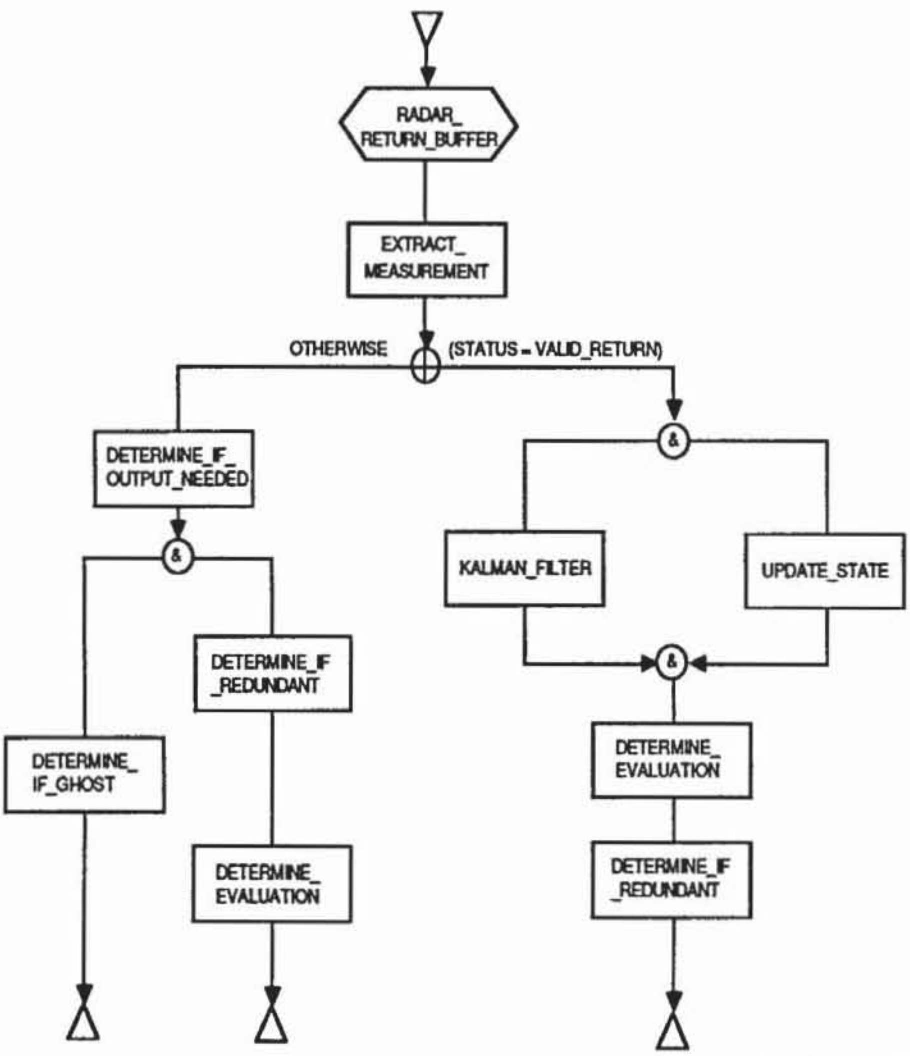


Fig. 21b. Flow Graph Representation of Sample R-Net.

ing reactions in a system evoked by events. An R-Net comprises nodes (*ALPHA*s and *SUBNET*s) and arcs connecting the nodes. While *ALPHA*s are functional specifications of processes, *SUBNET*s are specifications of processes at a lower level of hierarchy. A few operators (e.g. *AND, OR, FOR EACH*) allow the description of process control flow. Additionally, validation-points can be defined in order to obtain performance data.

In contrast, *RSL* is a textual specification language providing four primitive concepts:

(1) *Elements*: 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; *ALPHA*s stand for processes. Elements represent nouns in the language.

(2) *Relationships*: Relationships express logical links between Elements, e.g. ⟨data⟩ *INPUT TO* ⟨alpha⟩. They represent verbs in the language.

(3) *Attributes*: Attributes are used to complete the description of Elements, e.g. ⟨data⟩ *INITIAL VALUE* ⟨value⟩. They represent adjectives in the language.

(4) *Structures*: Structures are used to define the sequences of processing steps and represent R-Nets, *SUBNET*s, and *VALIDATION-PATH*s in terms of RSL-statements.

Figs. 19 and 20 show the symbols of R-Nets together with a sample R-Net. Fig. 21 demonstrates both the RSL-representation and the flow graph representation of a sample R-net. (Source: [31])

5. Management Aspects

There are (at least) two important management aspects.

First, the decision to use a specification system, and the choice of a particular product requires 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.

Second, 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 advantage of a specification system!

6. Conclusions

It is obviously possible to produce software (and systems) without any specification system.

Specification is not suited for every problem area. There are problems like developing user interfaces which call for 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 itself have standardized features.

Maintenance of specifications is not yet supported. This means when altering the specification the *user* has to find the implied modifications. In practice, there is still another problem with maintenance of specifications. The *program* is the only reference for modifications and not the specification. Therefore, the specification becomes obsolete.

Appendix: Bibliography on Specification and Specification Systems

Textbooks on Software Engineering

[1] R. Fairley, (1985): Software Engineering Concepts, McGraw-Hill Book Company, New York.

[2] Sommerville (1985): Software Engineering, Addison-Wesley Publ. Company, London, 2nd ed.

Life Cycle

[3] G.R. Gladden (1982): "Stop The Life-Cycle, I Want To Get Off," ACM SIGSOFT Software Engineering Notes, 7, 2, 35–39.

[4] M.M. Lehman (1980): "Programs, life cycles, and laws of software evolution," Proceedings of the IEEE, 68, 9, 1060–1076.

[5] J. Ludewig (1982): "Computer aided specification of process control software.," IEEE Computer, May 1982, 12–20.

[6] D.D. McCracken, and M.A. Jackson (1982): "Life Cycle Concept Considered Harmful," ACM SIGSOFT Software Engineering Notes, 7, 2, 29–32.

[7] W. Swartout and R. Balzer (1982): "On the inevitable intertwining of specification and implementation," Communications of the ACM, 25, 7, 438–440.

Fundamentals and Principles of Specification

[8] R. Balzer and N. Goldman (1979): "Principles of good software specification and their implications for specification languages," In: Proceedings of Specification of Reliable Software (SRS), IEEE Cat. No. 79 CH 1402-9C, pp. 58–67.

[9] B.W. Boehm (1976): "Software Engineering," IEEE Transactions on Computers, C-25, pp. 1226–1241.

[10] B.W. Boehm (1980): Software Engineering Economics. Prentice Hall, Englewood Cliffs, N.J.

[11] W.D. Brooks (1981): "Software Technology Payoff: Some statistical evidence," Journal of Systems and Software, 2, 3–9.

[12] W. Hesse, H. Keutgen, A.L. Luft and H.D. Rombach (1984): "Ein Begriffssystem für die Softwaretechnik," Informatik-Spektrum, 7, 4, S. 200–213.

[13] IEEE (1983): "Standard glossary of software engineering terminology," IEEE Std 729–1983.

[14] J. Kramer (ed.) (1982): "Glossary of terms," EWICS TC on Application Oriented Specification. Jeffrey Kramer, Imperial College, Dept. of Computing, 1980 Queen's Gate, GB-London SW7 2BZ.

[15] D.L. Parnas (1977): "The use of precise specifications in the development of software," In: Gilchrist, B. (ed.): Information Processing 77. North-Holland Publishing Company, Amsterdam, New York, Oxford, pp. 861–867.

[16] W.F. Racke and H.D. Rombach (1983): Methoden, Sprachen und Werkzeuge zur Software-Spezifikation. Universität Kaiserslautern, Fachbereich Informatik, Interner Bericht Nr. 66/83.

[17] P. Schnupp (1981): "Spezifikation für ein Spezifikationswerkzeug," In: Werkzeuge der Programmiertechnik, Springer-Verlag 1981, S. 75-100.

[18] M. Timm (1982): Grundlagen von Anforderungs- und Entwurfsspezifikationen im Prozess der Software-Entwicklung. GMD-Studien, Nr. 66, 82 S.

Surveys (Articles and Books)

[19] H. Balzert (1982): Die Entwicklung von Software-Systemen. Reihe Informatik/34, Bibliographisches Institut, Mannheim.

[20] G. Hommel (Hrsg.) (1980): Vergleich verschiedener Spezifikationsverfahren, am Beispiel einer Paketverteilanlage. KfK-PDV 186, Teile 1 und 2, Kernforschungszentrum Karlsruhe, BRD.

[21] P.P. Chen (1976): "The Entity-Relationship Model – Toward a Unified View of Data", ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, 9-36.

[22] L.L. Cheng (1978): Program design languages – an introduction. Report No. ESD-TR-77-324, Electronic Systems Division, Hanscom Air Force Base, MA 01731.

[23] Computer (1982): Special issue on application oriented specification, IEEE Computer, May 1982, 10–59.

[24] IEEE-SE (1977): Special collection on requirements analysis, IEEE Transactions on Software Engineering, SE-3, 2-84.

[25] J. Ludewig and W. Streng (1978): "Methods and tools for software specification and design – a survey," EWICS TC on Safety and Security, Paper No. 149, 23 Seiten.

[26] J. Ludewig (Hrsg.) (1983): Spezifikation von Realzeit-Systemen-Konzepte, Lösungen, Erfahrungen. 54. Tagung der Schweizerischen Gesellschaft für Automatik (SGA-ASSPA), Baden/Aargau, 1983–3-21.

[27] Y. Ohno (ed.) (1982): Requirements Engineering Environments, Proceedings of the International Symposium on current issues of Requirements Engineering Environments; Kyoto, Japan, September 20-21, 1982. NHPC, Amsterdam.

[28] D. Prentice (1981): "An analysis of software development environments," ACM SIGSOFT Software Engineering Notes, 6, No. 5, 19–27.

[29] C.V. Ramamoorthy and H.H. So (1977): "Survey of principles and techniques of software requirements and specifications," In: Software Engineering Techniques, Vol. 2, Infotech Intern. Ltd., Nicholson House, Maidenhead, Berkshire, England, pp. 265–318.

Particular Specification Methods and Systems

[30] M. Alford (1977): "A requirements engineering methodology for real time processing requirements," IEEE Transactions on Software Engineering, SE-3, 60-69. (on SREM).

[31] M. Alford (1980): "Software Requirements Engineering Methodology (SREM) at the Age of four," COMPSAC Conference 1980. (on SREM).

[32] H. Balzert (1985): Moderne Software-Entwicklungssysteme und Werkzeuge, Reihe Informatik/44, Bibliographisches Institut, Mannheim. (contains material on proMod, PRADOS and other systems; in German).

[33] R. Bell (1985): "Structured analysis aids in micro-computer system design," EDN, March 21, 1985, 251–257. (on Structured Analysis)

[34] J. Biewald, P. Göhner, R. Lauber and H. Schelling (1979): EPOS - a specification and design technique for computer controlled real-time automation systems. 4th International Conference on Software Engineering (ICSE) Munich, 1979, IEEE Cat. No. 79 CH 1479 - 9C, pp. 245–250.

[35] M. Hamilton and S. Zeldin (1976): "Higher Order Software – a methodology for defining software", IEEE Transactions on Software Engineering, SE-2, 9-32. (on (HOS)

[36] M. Lissandre, P. Lagier, A. Skalii, H. Massié (1984):

SPECIF – A specification assistance system. Institut de Génie Logiciel (IGL), Paris, France.
[37] J. Ludewig, M. Glinz, H.J. Huser, G. Matheis, H. Matheis and M.F. Schmidt (1985): "SPADES – A Specification and Design System and its Graphical Interface", 8th ICSE, IEEE CH2139-4/85/0000/0083, 83–89.
[38] D.T. Ross (1977): "Structured analysis (SA): A language for communicating ideas", IEEE Transactions on Software Engineering, SE-3, 16-34. (on SADT)
[39] D. Teichroew and E.A. Hershey III (1977): "PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems," IEEE Transactions on Software Engineering, SE-3, 41–48.
[40] E. Yourdon and L.L. Constantine (1979): Structured Design: Fundamentals of a disciplin of computer programs and systems design, Prentice Hall Inc., Englewood Cliffs, NJ.

Use of Programming Languages for Specifications, Prototyping

[41] S.J. Goldsack (ed.) (1985): Ada for specification: Possibilities and limitations, Cambridge University Press (for the Commission of the EC).
[42] B.W. Boehm, T.E. Gray and Th. Seewald (1984): "Prototyping versus Specifying: A multi-project experiment, 7th ICSE, Orlando, FL., March 1984, 473–484; also in IEEE Transactions on Software Engineering, SE-10 (1984), 290-303.