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.

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.

Keywords: Software Engineering, specification methods, specification languages.
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 deficiencies. 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 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>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>informal</td>
<td>not (precisely)</td>
<td>natural languages forms</td>
</tr>
<tr>
<td>formal</td>
<td>defined</td>
<td>pseudo-code (modern)</td>
</tr>
<tr>
<td>semi-formal</td>
<td>defined partially</td>
<td>programming languages</td>
</tr>
<tr>
<td>formatted</td>
<td>restricted (by forms)</td>
<td>not (precisely) defined</td>
</tr>
</tbody>
</table>

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.
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.
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-digram 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. 6. Detailed Actigram of “ASSIST S.A.D.T. USERS”

Fig. 7. Refinement of Activity "CREATE KITS"
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

Fig. 8. Sample SA-DFD.

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

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, data flows, 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).

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, 2GA, 2GS

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 STORES RSTACO. THIS CLEARS 2FF3, 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 ONEVENT = 1, 2FF3 WILL ALSO SET AT THIS TIME. DCLKS WILL BE COUNTED UNTIL DELTC = 1, CAUSING 2FF3 TO SET. WHEN EOD = 1, THE SELECT DELAY CLOCK LOGIC SWITCHES TO COUNTING DELAY BY TIME. THIS WILL CONTINUE UNTIL THE NEXT OCCURRENCE OF DELTC = 1, WHEN EOD = 1 WILL OCCUR. THE STATE MACHINE REMAINS IN THIS STATE UNTIL THE NEXT RSTACO.

LOGIC:
ALL FLIP-FLOPS ARE RESET ASYNCHRONOUSLY BY PROCESSOR ACTION:
SET STRTDEL = 0 WHEN RSTACO = 1
SET EOD = 0 WHEN STRTDEL = 1
SET EOE = 0 WHEN ATB = 1
ALL FLIP-FLOPS WILL SET ON CONDITION ON THE RISING EDGE OF DCLK
SET EOE(N+1) = ONEVENT + DELTC + EOD + NOEYNTS
SET EOE(N+1) = (EVDN + EOD)(DELTC + EOD) = EVOD + DELTC + EOD

Fig. 13. Minispec 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 (Information System Design and Optimization System) 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

<table>
<thead>
<tr>
<th>Entity-classes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL WORLD</td>
</tr>
<tr>
<td>ENTITY</td>
</tr>
<tr>
<td>real world objects which are out of the system</td>
</tr>
<tr>
<td>PROCESS</td>
</tr>
<tr>
<td>activities</td>
</tr>
<tr>
<td>INPUT</td>
</tr>
<tr>
<td>input data</td>
</tr>
<tr>
<td>SET</td>
</tr>
<tr>
<td>set of data elements</td>
</tr>
<tr>
<td>Relations:</td>
</tr>
<tr>
<td>GENERATES</td>
</tr>
<tr>
<td>e.g. (process) GENERATES (data)</td>
</tr>
<tr>
<td>RECEIVES</td>
</tr>
<tr>
<td>e.g. (process) RECEIVES (data)</td>
</tr>
<tr>
<td>UPDATES</td>
</tr>
<tr>
<td>e.g. (process) UPDATES (data)</td>
</tr>
<tr>
<td>CONSISTS</td>
</tr>
<tr>
<td>describes data structures; e.g. colour CONSISTS yellow, red, green, blue</td>
</tr>
</tbody>
</table>

Fig. 14. DFD of a Product Development.
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 STATEMENT

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';
48 > DESC;

Fig. 15. PSL-Input Source Listing Page (No. 1).
A ship stops at many ports and it is necessary to print out its itinerary.

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.

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.

A waybill related to a shipment of goods between two ports on a specified vessel. The shipment may consist of one or more containers.

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

Parameters: DB=VESSEL.DBF FILE=PSANAMES.PSAMEP NOCOMPLETENESS-CHECK
NOINDEX NOPUNCTED-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

Fig. 17. Report Showing a Tree-Structure by Indentation.
4.3.2. The Language
Different forms 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

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.

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

Fig. 18. Report Showing Part of the Same Information (of Fig. 17) in a Table.
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-
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
      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 (ALPHAs and SUB-NETs) and arcs connecting the nodes. While ALPHAs are functional specifications of processes, SUBNETs 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; ALPHAs 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, SUBNETs, and VALIDATION PATHs 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**

**Life Cycle**


**Fundamentals and Principles of Specification**


**Surveys (Articles and Books)**


**Particular Specification Methods and Systems**


SPECIF – A specification assistance system. Institut de Génie Logiciel (IGL), Paris, France.


Use of Programming Languages for Specifications, Prototyping
