T U M

I N S T I T U T F Ü R I N F O R M A T I K

Pragmatic and Formal Specification of System Properties by Tables

Manfred Broy

TUM–I9802
Januar 98

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-01-I9802-300/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©1998

Druck: Institut für Informatik der
Technischen Universität München

Pragmatic and Formal Specification of System Properties by Tables*)

Manfred Broy

SysLab
Fakultät für Informatik
Technische Universität München

A picture is worth a 1000 words

Folklore

It takes a long time to understand a 1000 words

Engineering experience

Abstract

We suggest table techniques as a pragmatic description formalism for a both precise and readable specification of systems, their interfaces, as well as their functional properties. We define a formal semantics for these tables by translating them into logical formulas. Tables are considered easier to read, to comprehend, or at least easier to communicate than large logical formulas. However, we consider formulas to be better suited for a manipulation by logical calculi to derive properties. By translating tables into formulas of predicate logic and vice versa, we provide a bridge between the conciseness of readable suggestive specifications and the preciseness of mathematical methods in software and systems engineering.

1. Introduction

Studies in the development of embedded software systems[1)] show that the requirements capture is one of the most decisive and critical issues in system development. There are at least three sources of troubles in the task of requirements engineering:

*) This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen" and the industrial research project SysLab sponsored by Siemens Nixdorf and by the DFG under the Leibniz program.

- requirements are not completely and not accurately identified and understood by the application experts and/or not properly communicated to the requirements engineers,
- requirements are not correctly documented, although they have been completely and accurately identified and understood,
- requirements are correctly documented, using informal techniques, but not properly interpreted and conceived by the system designers and implementers.

All three situations lead to serious errors and into costly troubles in the software development process. As it is well-known, bugs in the requirements tend to be very expensive, especially if they are captured very late in the development process such that much redevelopment is indispensable.

Of course, errors cannot be fully avoided when developing complex information processing systems. However, well worked out specification methods with adequate methods and techniques for analysing requirements and by this guaranteeing the quality and correctness of the specified requirements can help to avoid some pitfalls and to obtain a better control over the development process and the functional quality of the produced documents.

There are a number of properties that a practically helpful requirements engineering and documentation method should have. These are, among others:

- preciseness,
- conciseness,
- understandability,
- expressiveness,
- generality,
- tractability with respect to the development method and process,
- modularity.

Obviously, some of these properties are in a mutual conflict. Therefore we have to find acceptable compromises when designing or choosing a specification method.

In this paper we concentrate mainly on the issues of understandability, conciseness, and preciseness. The other goals, we hope, are taken care of by the development methods and the mathematical system models on which our description techniques are based.

It is well recognised that one of the most significant effects of applying formal techniques in software development consists in their potentials to improve the quality and precision of specifications. For large and complex systems such specifications have to be well-structured and modular. They have to be abstract concentrating on the interface description - also called the *black box behaviour* of systems - not taking into account implementation details or low level considerations. They have to be precise in the sense that they specify the system properties unambiguously.

The main purposes and advantages of using specification techniques in the system development process are fourfold:

- requirements engineering: specification techniques help the requirements engineer in the process of clarifying the required system properties,
- communication medium: specifications are a basis for the communication between the application experts, the requirements engineers, the system designers and implementers,

[1)] Embedded software systems are dedicated computing and control systems embedded into a technical environment.

- implementation: abstract descriptions of particular design and implementation ideas provide guidelines in system realization,
- documentation: specifications document the system properties for the use and reuse of a system and its components.

A good specification technique provides a helpful mental model (semantic model) of a system, a notation (formalism) for the concrete representation of specifications, and methods to reason about and to refine specifications.

One of the striking arguments against mathematical and logical formalisms for the specification of software requirements are difficulties with their understandability due to their technical complexity and sophistication. However, these arguments often are based on unjustified common sense and lack proper justification by experimental data. On the contrary, experiments have shown that graphical formalisms are not necessarily easier to understand than textual or mathematical ones (see [Petre 95]). It is very interesting to observe that the problems that readers have with diagrams are to a large extend similarly to those they have with logical formulas. If diagrams get very large, they do no longer provide helpful graphical patterns but rather provide a large amount of informations in a fairly unstructured way. It is not clear for a reader of a document which nodes to start with when studying a diagram and in which order to work through it in a systematic manner.

This indicates that graphical documentation techniques do not help, per se, to make specifications better readable and understandable, but need a careful design of their layout. Large formulas as well as large diagrams are difficult to deal with. If the information contained in the formulas and diagrams exhibits a particular structure it might be much more appropriate to gather this information outside the formula or the diagram and to present it in a table. This table can be well-combined with diagrams. They help to avoid an overloading of the diagrams by formulas and text. Moreover, tables allow for a very regular structure to present a large amount of information.

Tables are a well-kown concept in many disciplines including mathematics and engineering. There they are and have been used since a long time to provide well-structured descriptions for large amounts of informations. For instance, engineers often use tables and diagrams for specifying the behaviour of mechanical or electrical devices.

Of course, specifications are mainly thought to be helpful for documentation and communication. Therefore comprehensibility, readability, and understandability are first class goals of specification methods. Formal specifications are often considered to be hard to understand and difficult to deal with. This applies, in particular, as long as engineers are not familiar with predicate logic. Formulas of predicate logic can get rather long when complex large systems are to be described. Therefore the appropriate structuring of such formulas is crucial for their practical use. However, even with a consequent structuring and a well designed layout, logical formulas may be hard to read and to comprehend, especially if long lists of case distinctions have to be expressed. In such situations, tables are especially helpful as we will demonstrate. We concentrate in the following, in particular, on tables for specifying the behaviour of components. For this purpose, we understand tables as a systematic, more readable representation of large logical formulas with some regular structure.

Diagrams complement tables in helping to illustrate behaviours, but often do not describe behaviour precisely enough. Nevertheless, diagrams are used in many approaches to software engineering. However, preciseness and understandability need not be contradictory.

Our goal is to achieve both of them. Therefore we also show how tables can be represented by diagrams or be used in connection with diagrams.

Tables and table-directed notations have been advocated many times before in computing science. Prominent examples were decision tables in the 70`s. Less important were perhaps Nassi-Schneidermann style tables, an attempt to organize program text in a table-oriented style. Significant work on tables has been done by the group of David Panas, beginning with the remarkable work of [Heninger et al. 78] on the A-7E aircraft. This work has been continued in several directions exploring many ways of using tables in software development. What we present in the following does not actually go beyond that work. Our goal is to provide ideas for the use of tables for particular formal methods including algebraic specifications and stream-based models for reactive systems.

The main goal of this paper is to provide simple table and diagram techniques for the better understandability of mathematical specification techniques. These techniques are especially helpful when used to support axiomatic specifications. Our concrete goal is the support of the algebraic specification language SPECTRUM and of the functional system specification and development technique FOCUS, for which a rich mathematical theory and powerful methods for the specification, refinement, and verification are available (see [FOCUS 92] and [SPECTRUM 91]). The techniques to support logical formalisms by tables carry over, of course, to most of the other formal methods that work with logical formulas.

We do not advocate to use only tables or diagrams in specifications. We rather suggest to use a fine mixture of textual explanations, mathematical techniques, tables, and diagrams in well-chosen combinations side by side. We do not explain the mathematics of the formulas that we represent by the tables in great detail. We mainly see formulas as syntactic entities that can be represented by the tables and diagrams if they have a regular structure.

The paper is organised as follows. We start with a short section 2 that introduces and recapitulates some basic notions from algebraic specifications that are helpful when writing formulas in multisorted predicate logics. Then we have a short look at value tables in section 3. Section 4 introduces some instances of term tables and discusses their semantics. In section 5 this concept is generalised to tables that carry predicates. Section 6 introduces a general concept for representing formulas by tables. Section 7 discusses quantifiers and abbreviations in tables. Section 8 treats specific tables for the specification of interactive systems. Section 9 contains concluding remarks with a short discussion of the relationship of tables to diagrams.

2. Logical Basis

We are convinced that logics are appropriate as a firm basis of system specification. Whatever we write down in a table or in a diagram can also be expressed by a logical formula as long as the table and the diagram have a precise interpretation. The logical formula is perhaps less readable and more difficult to understand for untrained persons, however, it provides all the advantages of formal theories, such as a precise syntax and semantics as well

as a logical calculus for the formal manipulation, analysis, transformation, and deduction[1]. It also allows us to give a precise definition how different description methods are combined and related to form a comprehensive system description. In this section, we introduce the basic concepts for our logical formulas that are rather common and general.

Since we are convinced that type information[2] is a helpful idea for the better structuring and easier comprehension of system descriptions we work with a sorted (typed) higher-order predicate logic as a mathematical basis of specifications. In fact, throughout this paper we mainly use classical first order predicate logic. In particular, we use the following primitives for writing terms and formulas:

- *sorts* from a set S of sorts (including functional sorts),
- *constants* from a set F of function symbols and constants,
- *a sort assignment* for the constants by a mapping srt: $F \rightarrow S$,
- a set X of *identifiers*.

The sorts in S and function symbols in F with their associated sorts together form what is commonly called a *signature* $\Sigma = (S, F)$. Given a signature, we can form terms and formulas of propositional logic. Given identifiers, we can also form terms with free variables and formulas of predicate logics. Assigning carrier sets to sorts as well as elements and functions to the symbols in F, we can form algebras of a particular signature and use them to interpret terms. We assume that these concepts are familiar to the reader (see [Wirsing 90] for an extensive introduction).

Whenever we use an identifier in the following, we fix its sort before. So we can always assume that the function srt assigning sorts to constants can be extended to a function

$$\text{srt:} \quad X \cup F \rightarrow S$$

assigning sorts also to variables. We assume that the set of sorts S contains functional sorts and polymorphic sorts (like in [SPECTRUM 91]).

3. Value Tables

Tables have a long tradition in mathematics and engineering. By a table, a relation between values is represented. Therefore tables can be used to describe relations and - as a special case - to describe functions. In their simplest form tables contain only denotations of values as entries. Such tables are called *value tables*.

In a value table all entries are constants denoting values. We give two simple examples of value tables, namely Boolean tables defining connectives of propositional logic such as logical "or".

---

[1] We are aware of work that attemps to apply transformations and deductions directly to table representations and diagrams. However, even in this case it may be useful to relate these manipulations to the manipulation of logical foumulas.

[2] We follow the tradition of algebraic specifications and speak of sorts instead of types.

Tab 3.1 Value Table of the Logical Connective “or“

x	y	x v y
0	0	0
0	1	1
1	0	1
1	1	1

Tab 3.2 Value Table of the Logical Connective “or“ in Matrix Form

x v y	0	1
0	0	1
1	1	1

Value tables seem to have an obvious meaning. But even for value tables different semantic interpretations might exist. Since value tables are a special case of term tables for which we give a semantics in the following section we do not have to give any semantics or formal translation for value tables.

Nota bene, value tables help only when the domain of the functions that are specified is finite and, in particular, rather small. However, also for functions on infinite domains value tables can be used. Then, of course, only a finite set of argument/result pairs can be listed. Traditionally, in mathematics value tables have been used to provide the values of arithmetic and trigonometric functions such as logarithm, sinus, cosinus etc. Today, many of them have been replaced by pocket calculators. Nevertheless, still a lot of value tables are in use in everyday life such as tax tables etc.

4. Term Tables

An obvious step to generalise value tables are term tables. In term tables, we permit terms and variables as entries in addition to constants. A good example for a term table is the conditional expression. The conditional expression can of course be defined by a value table (provided the domain of values is finite), in a case distinction, or by logical formulas. A term table as given in Tab 4.1 is perhaps even more suggestive.

Tab 4.1 Term Table for the Conditional **if_then_else_fi**

b	**if** b **then** x **else** y **fi**
1	x
0	y
$\perp$	$\perp$

Value tables have a more or less obvious semantics. This also holds for simple term tables. For more sophisticated term tables this does no longer hold. Often, the more sophisticated term tables do not have an obvious interpretation, since there may exist several possible interpretations. Therefore we give a translation of term tables into logical formulas.

Both value tables and term tables can be understood as a representation of a possibly large logical formula that is in a simple disjunctive normal form.

Tab 4.2 A Scheme for a Term Table with m Lines and n Columns

x_1	...	x_n
...		...
t_1^i		t_n^i
...		...

Let x_1, ..., x_n be variables and t_1^i , ..., t_n^i be terms such that the sorts of the terms t_j^i and the identifiers x_j coincide. Tab 4.2 represents a large formula of predicate logic. We associate with it the formal meaning given by the following logical formula:

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} x_j = t_j^i$$

Here - as usual in predicate logic - all variables that occur are understood to be implicitly universally quantified. Aiming at this logical interpretation of a table, we speak of a *disjunctive table*.

Consider the example of the if-then-else-table given in Tab 4.1. By its interpretation as a disjunctive table we obtain the following logical formula

(b = 1 ∧ **if** b **then** x **else** y **fi** = x) ∨
(b = 0 ∧ **if** b **then** x **else** y **fi** = y) ∨
(b = ⊥ ∧ **if** b **then** x **else** y **fi** = ⊥)

This example shows already that the table is more concise (shorter and clearer structured) than the logical formula since in the formula certain large terms have to be repeated several times.

Disjunctive tables allow us to write only formulas with a very simple logical structure as tables. More common than formulas in disjunctive form are logical implications as they are used in conditional equations for writing algebraic specifications. Such conditional equations can be captured by implicative tables. Let us consider a simple example.

Example: Specification of a storage cell
We give a specification of an interactive storage cell that stores a data message from a set of messages M = Data ∪ {®} and returns it upon request. A request is indicated by the signal ®. The specification can be given by a data flow node as shown in Fig. 4.1, which indicates the communication channels, and by a state transition table given in Tab 4.3 (let d, e ∈ Data).

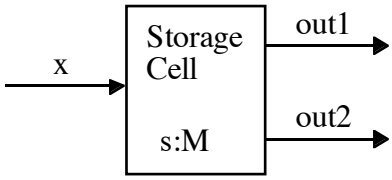

Fig. 4.1 Component with one Input Channel x, two Output Channels out1 and out2, and State s

The table describes a state automaton. By $i \in M$ we denote the input message on channel x, by $s \in Data$ the current state of the automaton. The input i stimulates a state transformation and some output. By state(s, i) we describe the successor state, by $out_1(s, i)$ the output on the first output channel, by $out_2(s, i)$ the output on the second output channel.

Tab 4.3 Implicative Table of a State Transition System with Input i on Channel x and State s (for all elements d, e ∈ Data)

i	s	state(s, i)	$out_1(s, i)$	$out_2(s, i)$
e	d	e	®	d
®	d	d	d	®

The functions out_1 and out_2 are output functions for the cell in state s obtaining input i. The function state gives the new state after the input i has been received in state s. The vertical double line separates the input for a state transition from its output. We will come back to the specification of interactive components in section 8. □

As explained, the interpretation of a table by a disjunctive formula is rather inflexible. Disjunctive tables require that all cases are covered. This is sometime too narrow. A more flexible concept of a table is obtained by an implicative table with a general form as it is given in Tab. 4.4.

Tab 4.4 Schematic Form of an Implicative Table

x_1	...	x_c	x_{c+1}	...	x_n
... t^i_1 ...	...	... t^i_c ...	... t^i_{c+1} ...	...	... t^i_n ...

Tab 4.4 has two blocks of columns separated by a vertical double line which separates the premises from the conclusions. The first block denotes premises, the second one conclusions. It has the semantic interpretation given by the following logical formula (again $x_1, ..., x_n$ are identifiers and $t^i_1, ..., t^i_n$ are terms):

$$\bigwedge_{i=1}^{m} ((\bigwedge_{j=1}^{c} x_j = t^i_j) \Rightarrow (\bigwedge_{j=c+1}^{n} x_j = t^i_j))$$

If a table provides a complete case distinction, the implicative and the disjunctive interpretation of the table are logically equivalent (for a proof see the appendix). Note that the implicative interpretation requires that the table is divided into two blocks, the conditions and the conclusions, that are - according to our convention - separated by the vertical double line.

According to the translation scheme for implicative tables given above for our example represented in Tab 4.3 we obtain the following specifying formulas as expressed by the vertical double line between the second and the third column (let e, d ∈ Data):

$$i = e \;\wedge\; s = d \;\Rightarrow\; state(s, i) = e \;\wedge\; out_1(s, i) = ® \;\wedge\; out_2(s, i) = d$$
$$i = ® \;\wedge\; s = d \;\Rightarrow\; state(s, i) = d \;\wedge\; out_1(s, i) = d \;\wedge\; out_2(s, i) = ®$$

These formulas show very clearly that the table exactly captures the relevant entries taken from the two conditional equations.

The vertical double line does not only show where the implication sign is to be placed. In our example it also separates input from output. This is very helpful for the understanding from a conceptual and from a methodological point of view.

The well-known definition of a function by cases is written in mathematics generally as follows

$$f(x) = \begin{cases} t_1 & \text{if } c_1 \\ t_2 & \text{if } c_2 \end{cases}$$

It can be written in the form of a disjunctive table as shown in Tab 4.5a or by an implicative table as given by table Tab 4.5b.

Tab 4.5a Disjunctive Table Describing a Complete Case Distinction

true	f(x)
c_1	t_1
c_2	t_2

Tab 4.5b Implicative Table for the Case Distinction

true	f(x)
c_1	t_1
c_2	t_2

In spite of the semantic equality between disjunctive and implicative tables for complete case distinctions, we advocate to use rather implicative tables than disjunctive tables. They provide more structure since they separate assumptions (conditions) from conclusions.

We have already mentioned the work of the group of Parnas on tables in software development. The tables of Parnas (see [Parnas 92]) in their most basic form can be represented in our approach, too. Parnas suggests to use tables to represent formulas of the form:

$$\bigwedge_{i=1}^{n} (C_i \Rightarrow \bigwedge_{j=1}^{m} v_j' = E_j^i)$$

by tables of the form shown in Tab 4.6.

Tab 4.6 Parnas Table

	C_1	...	C_n
...	...	...	...
v_j'	E_j^1	...	E_j^n
...	...	...	...

By such a table Parnas represents specifications of programs (statements) that work with the program variables v_1, ..., v_n. By v_1', ..., v_n' the values of the variables after the execution of the specified statements are denoted.

In our approach we may represent Tab 4.6 in a straightforward way by the transposed implicative table shown in Tab 4.7.

Tab 4.7 Parnas Table as an Implicative Table

true	v_1'	...	v_m'
C_1	E_1^1	...	E_m^1
...	...		...
C_n	E_1^n	...	E_m^n

This demonstrates that in its most simple form Parnas tables are a special case of implicative tables. This is a first hint on the generality and expressiveness of our table concept.

5. Syntax and Semantics of Term Tables

In this section we deal with a more general syntactic form of term tables and their semantics in more detail. A table consists of a heading line, a number of lines, and a number of columns. Also for the lines, headings may be provided that may serve as comments. We speak of a leading column. Every line and every column share an entry that is a term.

The tables introduced so far represent equational formulas. Tables are now generalised by allowing arbitrary terms and predicates as entries. Their semantics is formally defined in the following.

Often we want to talk about conditions in a table. They can easily be included into tables by allowing the headings and the entries of a table to be arbitrary terms. If the heading of the column j is the term h_j of sort $s \in S$ and the entry (in a line) for this column is a predicate P for elements of sort s then $P(h_j)$ is the proposition associated with that entry in place of the equation we are usually dealing with.

A disjunctive table represents the following logical formula (where the E_{ij} denote logical formulas that are obtained from the table entries)

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} E_{ij}$$

An implicative table represents the following logical formula

$$\bigwedge_{i=1}^{m} \left(\left(\bigwedge_{j=1}^{c} E_{ij} \right) \Rightarrow \left(\bigwedge_{j=c+1}^{n} E_{ij} \right) \right)$$

The formula E_{ij} is called the basic proposition of the entry in line i and column j. It is defined as follows.

We distinguish two kinds of entries:

(1) The heading entry h_j of the jth column is of the same sort as the term t_{ij} that is the table entry in column j and line i. Then E_{ij} stands for the equation

$$h_j = t_{ij}$$

(2) The heading entry h_j of column is a term of sort s and the table entry P_{ij} in column j and line i is a predicate for elements of sort s. Then E_{ij} stands for the formula

$$P_{ij}(h_j)$$

In this case we also use formulas as entries in which h_j occurs freely in the case h_j is a variable.

In the case of the example of the storage cell above a table with predicates reads as shown in Tab 5.1.

Tab 5.1 Transition Systems with Predicates as Entries

i	s	state(s, i)	$out_1(s, i)$	$out_2(s, i)$
isData	isData	i	®	s
®	isData	s	s	®

Here isData is a predicate

$$\text{isData:} \quad M \rightarrow \text{Bool}$$

that is specified by the following equation

$$\text{isData}(z) = (z \in \text{Data})$$

Often for this kind of tables, predicates have to be introduced additionally. This yields some overhead, but may help very much in the structuring of a specification and thus in its understanding provided the predicates and their names are well-chosen.

6. Schematic Formulas and their Representation by Tables

To provide sophisticated techniques for translating arbitrary formulas from predicate logic into tables is perhaps an overkill. Tables are only helpful as means of abbreviation and structuring if they are easy to comprehend and if a set of logical formulas is to be represented that are structurally similar.

Example: Specification of a sender in the alternating bit protocol
As an example we specify a component that serves as the sender in the alternating bit protocol example following the FOCUS approach (see [Focus 92]). The component can be graphically shown by a data flow node (see [Broy 93a]) as given in Fig 6.1.

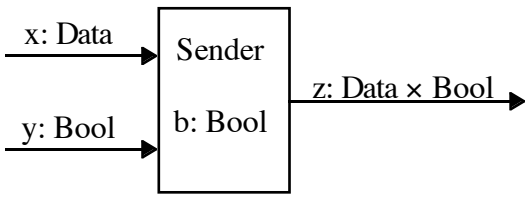

Fig. 6.1 Sender as a Data Flow Node

We are particularly interested in the syntactic representation of formulas by tables. Therefore we do not give an extensive explanation of the meaning of the example formulas represented by the tables. However, it is helpful to give some brief hints. In the following x, y, x' are identifiers of the sort stream of Data or stream of Bool, as indicated in Fig 6.1. A stream is a finite or infinite sequence of elements. The identifier b is an attribute of the sort Bool. The functions ft and rt operate on streams and deliver the first element of a stream and the rest of a stream resp. The behaviour of the sender is captured by instances of the logical formula of the form

Send(b)(x, y) = (ft.x', b')^Send(b')(x', rt.y)

with particular choices for the values b' and x' that depend on the actual message ft.y. These choices are instantiated as specified by the table 6.1. The table shows the two instantiations of the formula above for the cases "matching acknowledgement" (Positive Ack) and "not matching acknowledgement" (Negative Ack). □

Tab 6.1 Table of the Instances for the Sender of the Alternating Bit Protocol

	ft.y	b'	x'
Positive Ack	b	¬b	rt.x
Negative Ack	¬b	b	x

In the example chosen above we only work with two instantiations of the schematic formula and therefore with a table that has only two lines. More realistic protocol examples have to be represented by tables that contain many more lines an columns. This usage of a table allows us to separate the schematic form of a set of formulas from their instantiations.

As it is shown by our simple example, in specifications we sometimes have to write a sequence of formulas of the same structure that are all equal apart from certain significant subexpressions. Assume we work with a formula of the form

$$\bigwedge_{i=1}^{m} E(t_1^i,...,t_n^i)$$

where $E(x_1, ..., x_n)$ is an arbitrary maybe large logical formula where t_1^i , ..., t_n^i are terms of the sort $s_1, ..., s_n$. Let $x_1, ..., x_n$ be fresh variables. Logically equivalent, we work with a formula of the form

$$\bigwedge_{i=1}^{m} \left(\bigwedge_{j=1}^{n} x_j = t_j^i \right) \Rightarrow E(x_1, ..., x_n)$$

We represent this formula by a table as schematically shown by the Tab 6.2.

Tab 6.2 Table for Representing a Schematic Formula and its Instances

$E(x_1, ... , x_n)$		
x_1	...	x_n
...		...
t_1^i	...	t_n^i
...		...

This form of tables for representing arbitrary schematic formulas gives us more flexibility in specifications. We can choose the schematic form of the formulas freely and describe their instantiations by the entries of the table.

The simple transition of the formula given above does not work properly if $E(x_1, ..., x_n)$ is a formula that contains quantifiers and some of the x_j occur in the range of the quantifiers and the terms t_j^i contain occurrences of identifiers that are bound by these quantifiers. Then we specify the meaning of the table by replacing the identifiers x_j *textually* by the terms t_j^i. In this case the terms tij replace the identifiers in E(x1, ... , xn) by straightforward substitutions without renaming bound identifiers. This purely syntactic substitution is in contrast to logical substitution or to applications of functions in l-notation where locally bound identifiers are renamed by a-convention to avoid name-clashes. However, we should keep in mind that the

identifiers x1, ..., xn are not introduced for reasons of deduction as this is the case in predicate logic but as simple textual abbreviations.
A note on abbreviations:
carefully chosen abbreviations help a lot to improve the readability of formulas. For abbreviations, it is important to distinguish between textual abbreviations, the semantics of which is explained by simple textual substitutions, and parameterized abbreviations, which introduce functions with arguments. The difference between these two becomes crucial when working with bound identifiers. Consider the following formula that is again taken from a FOCUS specification (we come back to this formula) in section 7:

$$\forall\, d \in D: f.\langle d\rangle = \langle\rangle \wedge \forall\, x \in M^{\omega}: \exists\, f': B.f' \wedge f(d\hat{}\circledR\hat{}x) = d\hat{}f'.x$$

Using *textual abbreviations* we may write for this formula

$$\forall\, d \in D: \text{Case1} \wedge \text{Case2}$$

where Case1 and Case2 are textual abbreviations standing for the following formulas as shown below:

$$\text{Case1} \equiv (f.\langle d\rangle = \langle\rangle)$$

$$\text{Case2} \equiv (\forall\, x \in M^{\omega}: \exists\, f': B.f' \wedge f(d\hat{}\circledR\hat{}x) = d\hat{}f'.x)$$

Using *parameterized specifications* of Case1 and Case2 we would write

$$\forall\, d \in D: \text{Case1}(d) \wedge \text{Case2}(d)$$

Case1 and Case 2 are predicates specified as follows:

$$\text{Case1}(d) \equiv (f.\langle d\rangle = \langle\rangle)$$

$$\text{Case2}(d) \equiv (\forall\, x \in M^{\omega}: \exists\, f': B.f.' \wedge f(d\hat{}\circledR\hat{}x) = d\hat{}f'.x)$$

Both concepts of dealing with abbreviations have advantages and disadvantages. Both seem helpful in their own way. Therefore it seems good to have both concepts available for a flexible specification formalism. End_of_Note

Of course, all the tables introduced so far can be expressed by the technique of giving a number of instances for a schematic formula.

Example: Specification of a Routing Cell
A routing cell has two input and two output lines. On one line it receives messages and forwards it on one of its output lines, depending on the value on its second Boolean input line called y. It acts as a switch. It can also be specified by the table technique for schematic formulas. The syntactic interface of the cell is shown as a data flow node in Fig. 6.2.

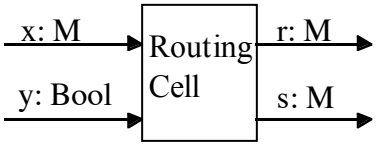

Fig 6.2 Routing Cell as a Data Flow Node

The table Tab 6.3 fixes the instantiations of the given schematic formula where m is of sort M. Here the entry "-" in the table expresses that the respective output is empty.

Tab 6.3 Table of the Formula Specifying a Routing Cell

Cell ‹ x:m ‹ y:b = r:c ‹ s:e ‹ Cell		
b	c	e
true	m	–
false	–	m

In the sample formula of Tab 6.3 we use a notation for the algebraic specification of interactive systems as introduced and explained in detail in [Broy 96a]. □

By the technique of schematic tables we achieve a valuable flexibility for writing conjugation of instances of arbitrary logical formulas as tables. We can always provide a formula scheme that is instantiated by the lines provided by the table.

7. Readability of Formulas and Structuring of Tables

In our approach each table represents a formula in predicate logic. Formulas containing quantifiers can be written as tables by using the techniques of section 6 for representing schematic formulas which of course may contain quantifiers.

Since each table can be seen as a specific representation of a logical formula we may combine tables like logical formulas. Of course, to form formulas with tables as subexpressions seems to be difficult, especially when tables are very large. However, we have the possibilities to introduce names for tables. Then it is easy to build formulas with tables by referring to their names.

Another option of representing nested formulas by tables is the nesting of tables exactly as the logical formulas are nested. In this section we follow this second line of thought. Our goal is a general concept for the representation of large complex logical formulas by hierarchically nested structured tables.

Often the terms or formulas we want to use as entries in tables get very large such that they hardly fit into the columns of a table. Then tables get unreadable. Sometimes the same large formula occurs in many entries. In both cases it is appropriate to work with well-chosen abbreviations. Abbreviations by parameterized specifications and textual abbreviations are a simple and well-known concept in predicate logic and computing science as mentioned above.

The readability of a logical formula is often difficult according to the use of local variables bound by quantifiers especially when a number of nested quantifiers occurs. For a given signature $\Sigma = (S, F)$ that defines a set of sorted names we may write a logical formula E. For such a formula we may require, in principle, that at most the symbols in F occur as free identifiers. This forces us to introduce quantifiers for all other identifiers that are used as bound local variables.

A convention that allows us to leave out and therefore to save the explicit writing of some universal quantifiers is the universal closure. If there occurs an identifier in a formula E that is not contained in the signature F then it is understood as (implicitly) universally quantified. Its sort then has to be deduced from the context. However, often it improves the readability to write quantifiers explicitly.

The general scheme of representing quantifiers in tables is as follows. For the formula

$$\forall\ x \in M: P(x)$$

we write a table as shown in Tab 7.2. Then we can apply a table along the lines of using tables for schematic formulas to represent the formula P(x).

Tab 7.2 Universal Quantification in a Table

∀ x: M:
P(x)

For the formula

$$\exists\ x \in M: P(x)$$

we write the table of the form given in Tab. 7.3.

Tab 7.3 Existential Quantification in a Table

∃ x: M:
P(x)

In spite of all the possibilities of writing formulas in a more structured style, nested quantification remains a concept that makes formulas more difficult to write, read, understand, and manipulate. Sometime it is possible to replace existential quantification by declarations. This option is explained in detail in the following.

Existential quantification is closely related to the declaration of identifiers and their binding and vice versa. Whenever we write a declaration

Let x = t **in** e

or

e **where** x = t

what we mean can be expressed logically (provided e is a Boolean expression) by the formula:

$$\exists\ x: e \wedge x = t$$

Note that this formula is equivalent to the formula

$$\forall\ x: x = t \Rightarrow e$$

provided we have the validity of the formula ∃ x: x = t. The proposition ∃ x: x = t is trivially true, if every term t has a value[1]. So it is true in classical equational logic.

Therefore declarations may always be understood as an abbreviation for certain formulas in predicate logic. Clearly, in a declaration x = t we generally assume that the formula

$$\exists\, x : x = t$$

is valid. This shows a remarkable difference between the roles the subformula e and that of the subformula x = t in the formula

$$\exists\, x : e \wedge x = t.$$

The subformula x = t can be seen to be just an auxiliary construction that can be always satisfied and serves as an auxiliary construct for the formulation of e. If the declaration is not recursive and thus x does not occur in t then the formula

$$\exists\, x : e \wedge x = t$$

expresses the same as the formula

$$e[t/x].$$

In these cases we may break a formula in pieces by structuring quantifiers into declarations. This idea of a structured specification of existential quantification can be generalized to formulas of the form

$$\exists\, x : e \wedge B(x)$$

provided the proposition

$$\exists\, x : B(x) \qquad (*)$$

holds. In contrast to the equation x = t that we used in the place of the subformula B(x) above the subformula B(x) does not necessarily identify the element x uniquely. Then we may read the formula as the proposition

e where x is arbitrary such that B(x) holds

provided, the logical value of e does not depend on the choice of x. Expressed in logical terms we require

$$B(x) \wedge B(x') \Rightarrow (e \equiv e[x'/x]) \qquad (**)$$

Given the conditions (*) and (**) the formula ∃ x : e ∧ B(x) is moreover equivalent to the formula[2]

$$\forall\, x: B(x) \Rightarrow e.$$

---

[1] Of course the situation gets more complicated if there are expressions t without a value (such as in the presence of partial function) or if x = t may have several solutions for x (such as in the case of recursive declarations).

[2] As my colleague Wolfgang Naraschewski pointed out to me there is a relationship to dependent types in type theory; there we express ∃ x: e ∧ B(x) as Σ x: e × B(x)

If the requirement described above is valid, we write the proposition ∃ x: t ∧ B(x) rather in the tabular form given in Tab 7.4.

Tab 7.4 Representation of the Formula ∃ x : e ∧ B(x)

e
where x : B(x).

Here we require as a well-founded condition that (*) and (**) hold.

We know how to specify the behaviour of interactive components in principle by logical formulas in FOCUS (see [FOCUS 92]). For complex components, these formulas can get fairly large and unreadable. Therefore we need a better way of structuring them to make them easier to read and understand. One possibility is to use good formatting concepts and conventions (see [Lamport 93] for an example) for formulas. In addition, the introduction of well-chosen auxiliary declarations may help to shorten the formulas occurring in tables and to provide additional structure (see [Spies 94] for an extended example).

A schematic table is used to abbreviate a sequence of implications connected by logical conjunction. Sometimes in the premise or in the conclusion local quantifiers are used.

Example: Quantification in FOCUS
Using the techniques of FOCUS (cf. [Broy 94]) we work with specifications defining sets of functions via predicates like the following one. Here the predicate B is specified that characterizes functions on streams:

$$B.f \equiv \forall\, d \in D: f.\langle d\rangle = \langle\rangle \wedge \forall\, x \in M^{\omega}: \exists\, f': B.f' \wedge f(d\hat{}\circledR\hat{}x) = d\hat{}f'.x$$

This specification essentially expresses that a function f mapping streams of messages to streams of messages characterizes a one element buffer. The formula indicates that f produces the empty output stream on the input stream that carries exactly one data element and that it produces the data element d as output on the input stream that starts with the data message d followed by the request signal ® followed by the stream x. Then the output d is followed by the stream that f produces on the input stream x. The nested quantification is rather horrible here and difficult to read and to understand. A table specification as shown in Tab 7.5 could provide more structure. This is a first example of a hierarchically structured table.

Tab 7.5 Table of a Formula Specifying a Queue

$B \equiv (f: M^{\omega} \rightarrow M^{\omega}): \exists\, f': B.f' \wedge$	
$\forall\, d \in D, x \in M^{\omega}: f.i = o$	
i	o
‹d›	‹›
d^®^x	d^f'.x

Here existential and universal quantification is written in a more structured style, which makes the formula hopefully better readable. □

We may include quantification into the heading of tables describing formulas. This allows us also to give tables for formulas with quantifiers as shown in Tab. 7.5.

The readability of a formula critically depends on its layout. Especially, if a formula is large the layout should be carefully chosen. The difference between a table and a diagram may become more or less unimportant.

Often for large, uniformly structured formulas it is better to choose the following layout (advocated by [Lamport 93])

∧	t_1
∧	t_2
∧	t_3

instead of

$$t_1 \wedge t_2 \wedge t_3.$$

Then the structure of the formulas gets clearer and more uniform such that the formulas get more readable. We may even use a style like in OCCAM and write only the first instance of the operator and leave out all the others.

In a similar style we better write

	t_1
⇒	t_2

for large formulas t_1 and t_2 instead of the logical formula

$$t_1 \Rightarrow t_2$$

to make the formula more readable.

8. Interaction Tables

The description of the behaviour of reactive systems is an important but difficult task. As we have shown in the examples before, also for this task tables are useful. In this section, we consider a specific situation which is typical for protocols in telecommunication systems. To illustrate the issue we use again a one element buffer as a simple example which is now lossy, however.

Example: Unrealiable Buffer
We work with the following sets of input and output messages (assume a set of data elements D):

$$M = D \cup \{®\}$$

$$N = D \cup \{®, @\}$$

The behaviour of the unrealiable buffer is described by a stream processing function (M^ω denotes the set of finite or infinite streams which are sequences of elements over M)

$$f: M^{\omega} \rightarrow N^{\omega}$$

The lossy buffer can be specified by a predicate depending on the state of the buffer

$$Q: M \rightarrow ((M^{\omega} \rightarrow N^{\omega}) \rightarrow Bool)$$

The basic properties of the buffer can be described by a table (let $d1, d2 \in D$), as it is shown in Tab 8.1 (an explanation of the meaning of this notation is found in [Broy 96a]). Roughly speaking σ denotes the current state, σ' the successor state, a the input message and r the output message. □

Tab 8.1 State Transition System Described by a Table

$Q(\sigma) \langle a \Leftarrow r \langle Q(\sigma')$			
σ	a	σ'	r
d1	d2	®	®
d1	®	®	®
d1	®	®	d1
®	d2	®	®
®	d2	d2	@
®	®	®	®

The formula expressed by the table is rather weak and too liberal to capture the behavior of the lossy buffer. It only expresses which output is possible for the input a, but not which output is forbidden. We rather want to understand the table as listing all possible transitions. The formula that we actually want to express by the Tab 8.1 reads as follows

$$\bigwedge_{j=1}^{n} (Q(s_j) \langle a_j \equiv \bigvee_{i=1}^{m} r_j^i \langle Q(s_j^i))$$

For a precise explanation of the meaning of this formula see [Broy 96a]. This example shows that it is not so simple that we can work with a table for a schematic formula since here the structure of the formula is slightly more complicated. Nevertheless, it is possible to write the formula in the form of a table.

We represent this formula by an interaction table as shown in its general form in Tab 8.2. This table is very particularly structured as suggested by the formula to be represented. This example demonstrates that the technique of introducing tables for schematic formulas does not always work if we have a specific nesting of iterations in formulas. Then we should not hesitate to introduce other techniques for writing tables. The meaning of Tab. 8.2 is given by the following formula

$$\bigwedge_{j=1}^{n} (s = s_j \wedge a = a_j \Rightarrow Q(s)\langle a \equiv (\exists\, r,s' : r\langle Q(s') \wedge \bigvee_{i=1}^{m} s' = s_j^i \wedge r = r_j^i))$$

which can be simplified to

$$\bigwedge_{j=1}^{n} Q(s_j) \langle a_j \equiv \bigvee_{i=1}^{m} r_j^i \langle Q(s_j^i)$$

For an extensive discussion and an explanation of its meaning see [Broy 96a].

Tab 8.2 Interaction Table

Q (s) ‹ a		r ‹ Q(s')	
s	a	s '	r
…	…	…	…
s_j	a_j	… s_j^i …	… r_j^i …
…	…	…	…

This table description does not express any fairness properties in general. Such properties can be added by equations. For instance, we may in addition to the properties expressed in the table require that - if we try long enough - each message gets through. This can be expressed by the following formula:

$$Q(\sigma).f \wedge (\sigma \in D \neq m \in D) \Rightarrow \exists\, i \in Nat: f.m^i \neq ®^i$$

The conclusion can also be expressed by the more simple formula

$$f.m^\infty \neq ®^\infty$$

Another way to express this is by the equations

$$Q(d).f \Rightarrow \exists\, i \in Nat: f.®^i = ®^i\hat{}d$$

$$Q(®).f \Rightarrow \exists\, i \in Nat: f.d^i = ®^i\hat{}@$$

These equations can again be represented by a table, of course, as it is shown in Tab 8.3. Here we assume $d \in D$.

Tab 8.3 Fairness Conditions Represented by a Table for a Schematic Formula

$Q(\sigma).f \Rightarrow \exists\, i \in Nat: f.m^{i+1} = ®^i\hat{}a$		
σ	m	a
d	®	d
®	d	@

Again this table represents a logical formula, or more precisely a collection of similar formulas.

Note that we can freely combine tables and formulas since tables are only seen as syntactic sugar for formulas. It is one additional advantage of our technique that it allows us to translate the information given by tables into formulas of predicate logic such that we can use tables and formulas side by side with a clear definition how they work together.

9. Combining Tables with Diagrams

There are many possibilities to combine tables with diagrams. Diagrams may, like tables, be helpful to support the quick understanding of large specifications. Each table represents a finite relation between terms. It can be rewritten into a diagram. Vice versa each diagram that is a directed labelled graph can be represented by a table.

A diagram is a labelled graph consisting of nodes and arcs. In computer science many variations of diagrams are used. Often they differ only in the form of the symbols used to represent their nodes. Of course, we cannot and want not treat all kinds of diagrams found in computer science in this section in a systematic way. Therefore, we demonstrate the combination of tables with diagrams only for simple state transition diagrams. For them the nodes represent states and the arcs represent transitions.

A diagram is a labelled graph with labels for the nodes and for the arcs. Every node and arc may carry several labels such as names and values. The translation of a diagram into a table and vice versa can be described by fixing the way an arc is represented by a table.

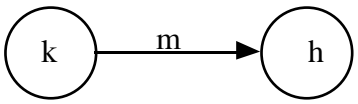

Fig. 9.1 An Arc with its Two Nodes

We can then associate an entry in the table with an arc as shown in Fig. 9.1 of the form of a line in the table with the following entries:

k	m	h

If this translation scheme has been fixed, each diagram can be translated into a table and vice versa. Note, however, that we can freely choose which columns are used to represent arcs and which are representing nodes. This may lead to quite different options for diagrams to represent tables.

A general way to translate graphs into diagrams is sketched in the following. Assume a directed graph with labelled arcs and labelled nodes. We may easily translate the graph into a table with three columns. Then each arc corresponds to a line with the entry in the first column being the source node label, the entry in the second column being the arc label and the entry in the third column being the target node label. State transition diagrams are discussed in detail in [Broy 96b].

10. Concluding Remarks

Specifications that are readable but nevertheless precise are a valuable concept in system development. We hope that we managed to demonstrate that readability and preciseness are not a contradiction. On the contrary, a vague, imprecise specification is hardly understandable. Of course, often complications in formulas make the understanding more difficult, however, simplifications should correspond to abstraction and not to vagueness. A well-chosen structure and layout may help a lot to keep formulas readable.

A well-chosen balance between explanatory text, formulas, tables, and diagrams is a must for useful specification methods. Such a structuring of specifications requires insights into the application area and an understanding for the specification methods and goals. Here we are only at a beginning in software engineering. Much more experience is needed in how to apply specification techniques in practical projects. Furthermore a scientific basis is necessary that allows us to integrate the various description formalisms. This paper is intended as an attempt to contribute to this basis.

The techniques of specifications used in practical applications in systems engineering often are not precisely defined. Typical examples are statecharts or SDL (see [ITU 93]). In both cases many proposal for a semantics exist while a formal reference semantics is missing. Obviously the concepts of such languages allow for a wide variety of semantc interpretations. Experiments show that such languages lead to quite different interpretations of descriptions even by the experts. This is a hint that the chosen description concepts are too complex. Therefore simple, suggestive concepts with a straightforward semantic interpretation of graphical description formalism and tables are needed that are suggestive and often more familiar to most users. However, if one does not understand the logical connectives, it is unlikely that one understands the meaning of the diagrams and tables in all its consequences.

Acknowledgement

This paper has been influenced by discussions we had during the design of [Grapes 90]. I am grateful to Wolfgang Naraschewski, Jan Philipps, Katharina Spies, and David Parnas for carefully reading versions of this paper and providing feedback.

Appendix: Disjoint, complete case distinctions

We consider a finite set of cases p_i and a finite set of consequences c_i, $1 \le i \le n$. We speak of a complete, disjoint case distinction if the following two propositions hold

$$\bigvee_{i=1}^{n} p_i \qquad \{\text{completeness}\}$$

$$\bigwedge_{i=1}^{n} \bigwedge_{\substack{j=1 \\ j \neq i}}^{n} (p_i \Rightarrow \neg\, p_j) \qquad \{\text{disjointness}\}$$

Under these conditions we can show that the following two propositions are equivalent

$\bigvee_{i=1}^{n} (p_i \wedge c_i)$ {disjunctive normal form}

$\bigwedge_{i=1}^{n} (p_i \Rightarrow c_i)$ {implicative form}

We first prove the equivalence for n = 2. We start the deduction processes with the implicative form:

$(p_1 \Rightarrow c_1) \wedge (p_2 \Rightarrow c_2)$

$\equiv$ {definition of implication}

$(\neg p_1 \vee c_1) \wedge (\neg p_2 \vee c_2)$

$\equiv$ {distributive law}

$(\neg p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge c_2) \vee (\neg p_2 \wedge c_1) \wedge (c_1 \wedge c_2)$

$\equiv$ {disjointness, completeness imply $\neg p_1 \equiv p_2$}

$(p_2 \wedge c_2) \vee (p_1 \wedge c_2) \vee (c_1 \wedge c_2)$

$\equiv$ {disjointness, completeness imply $(c_1 \wedge c_2) \Rightarrow (p_1 \wedge c_1) \vee (p_2 \wedge c_2)$}

$(p_1 \wedge c_1) \vee (p_2 \wedge c_2)$

We prove the equivalence for n > 2 by induction as follows:

$\bigwedge_{i=1}^{n+1} (p_i \Rightarrow c_i)$

$\equiv$ {definition of iterated conjunction and completeness of case distinction}

$((\bigvee_{i=1}^{n} p_i) \Rightarrow \bigwedge_{i=1}^{n} (p_i \Rightarrow c_i)) \wedge (p_{n+1} \Rightarrow c_{n+1})$

$\equiv$ {by the equivalence for n = 2 and induction}

$(\bigvee_{i=1}^{n} p_i \wedge \bigvee_{i=1}^{n} (p_i \wedge c_i)) \vee (p_{n+1} \wedge c_{n+1})$

$\equiv$ {since $\bigvee_{i=1}^{n} (p_i \wedge c_i) \Rightarrow \bigvee_{i=1}^{n} p_i$ by absorption}

$\bigvee_{i=1}^{n+1} (p_i \wedge c_i)$

The proved equivalence justifies the equivalence of disjunctive and implicative tables in the case of complete, disjoint case distinctions. □

References

[Broy 91]
M. Broy: Towards a Formal Foundation of the Specification and Description Language SDL, FAC, 1991, 3, 32-57

[Broy 93a]
Broy, M.: (Inter-)Action Refinement: The Easy Way. In: Broy, M. (ed.): Program Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118, pp. 121-158, Berlin, Heidelberg, New York: Springer 1993

[Broy 93b]
M. Broy: Functional Specification of Time Sensitive Communicating Systems. REX Workshop. ACM Transactions on Software Engineering and Methodology 2:1, Januar 1993, 1-46

[Broy 94]
M. Broy: Specification and Refinement of a Buffer of Length One. Marktoberdorf Summer School 1994

[Broy 96a]
M. Broy: Algebraic Specification of Reactive Systems. M. Nivat, M. Wirsing (eds): Algebraic Methodology and Software Technology. 5th International Conference, AMAST '96, Lecture Notes of Computer Science 1101, Heidelberg: Springer 1996, 487-503

[Broy 96b]
M. Broy: The specification of System Components by State Transition Diagrams. Unpublished Manuscript 1996

[FOCUS 92]
M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, R. Weber: The Design of Distributed Systems - an Introduction to FOCUS. Technische Universität München, Institut für Informatik, TUM-I9203, Januar 1992

[Fuchs 93]
M. Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung; Technische Universität München, Institut für Informatik, TUM-INFO, SFB-Bericht 342/1/93, Januar 1993

[Grapes 90]
GRAPES-Referenzmanual, DOMINO, Integrierte Verfahrenstechnik. Siemens AG, Bereich Daten-und Informationstechnik 1990

[Heninger 78]
K. Heninger, J. Kallander, D. L. Parnas, J. Shore: Software Requirements for the A-7E Aircraft. NRL Report 3876, November 1978

[Heninger 80]
K. L. Heninger: Specifying Software Requirements for Complex Systems: New Techniques and Their Application, IEEE Transactions Sortware Engineering, Vol. SE-6, No 1, Januar 1980

[ITU 93]
CCITT, Recommendation Z.100, Specification and Description Language (SDL), ITU-T, 1993

[Lamport 93]
L. Lamport: How to Write a Long Formula. DIGITAL Systems Research Center, SRC 119, December 1993

[Parnas 92]
D. L. Parnas: Tabular Representation of Relations. CRL Report #260, McMaster University, September 1992

[Parnas 93]
D. L. Parnas: Predicate Logic for Software Engineering, IEEE Transactions on Software Engineering, Vol 19, No. 9, September 1993

[Petre 95]
M. Petre: Why Looking Isn't Always Seeing. Comm. ACM 38:6, June 1995, 33-44

[SPECTRUM 91]
M. Broy, Ch. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, K. Stolen: The Requirement and Design Specification Language SPECTRUM. Technische Universität München, Institut für Informatik, TUM-I9140, Oktober 1991

[Spies 94]
K. Spies: Funktionale Spezifikation eines Kommunikationsprotokolls, Technische Universität München, Institut für Informatik, TUM-I9414, Mai 1994

[Wirsing 90]
M. Wirsing: Algebraic Specification. Handbook of Theoretical Computer Science, Vol. B, Amsterdam: North Holland 1990, 675-788