Software Specification Methods and Attribute Grammars

GUENTER RIEDEWALD, PETER FORBRIG

1. Introduction

An important problem of modern computer science is the development of quality software. The necessary design of large systems in the 1980s requires other methods than the design of smaller systems in the 1960s and becomes more and more an engineering problem.

There are many different methods to cope with this problem. These methods range for example from data driven program development by Jackson [Jac 75] to information hiding or data encapsulation by Parnas [Par 72].

Computer science calls for programming development systems, which support the programming development process by the computer itself.

After a short suvery of the fundamentals of software specification we want to illustrate the use of attribute grammars in commercial data processing. It will be presented how the well known methods of data driven programming and data encapsulation, usually classified as contrary concepts, can be combined by using attribute grammars with abstract data types.

Such attribute grammars represent specifications in a clearly readable form. The grammatical definition formalism is used only to the necessary extend. Many implementation details are encapsulated in abstract data types.

2. Software Specification

2.1. Life Cycle Model

We will use the life cycle model for demonstrating the problems of software design in different stages of its development. Our point of view is demonstrated in Figure 1.

The arrows in Figure 1 represent possible relations between components of the life cycle.

The first problem in software design is to define the exact task of a programming system. In all other phases of development the corresponding specifications have to be compared and verified with this requirement specification.

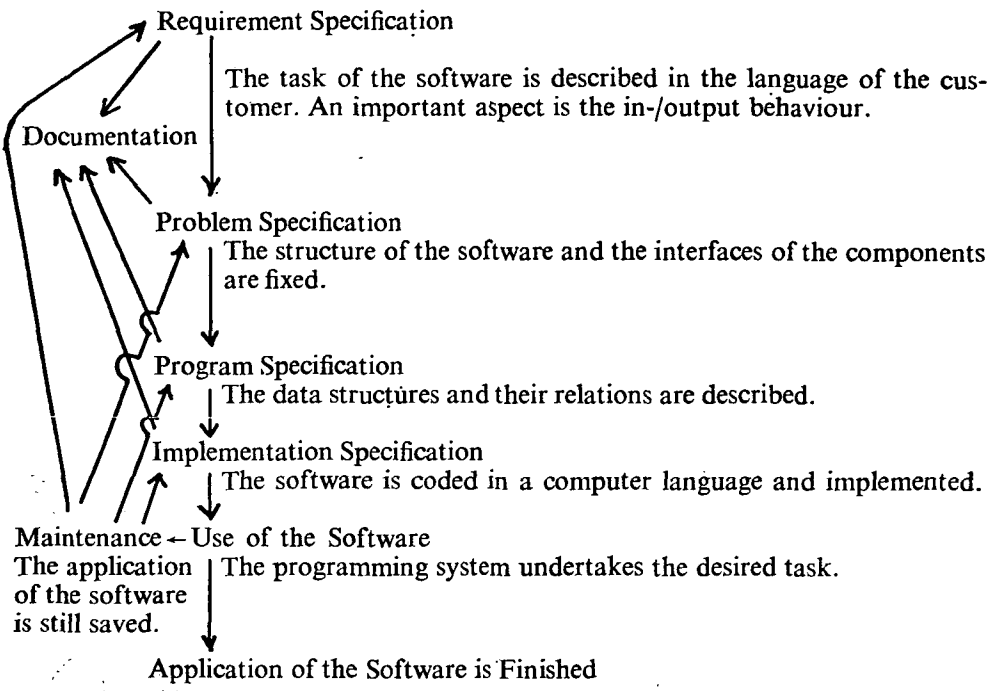

Figure 1. Life Cycle Model of Software

Figure 1 also demonstrates that the development of software is not finished after its implementation. Maintenance of software is of utmost importance. Experiences in software engineering have shown, that maintenance costs can be higher than all earlier development costs.

If programming systems are not developed in a good manner according to maintenance, the application of the software becomes more and more complicated.

According to Figure 1 software maintenance can influence all specifications of a project. The simplest modifications may have an influence on implementation only. More complex changes are even more difficult and it is possible, that not all specifications are updated in the right way.

In this case different descriptions contain contradictions and the application of the project is very difficult.

The same problems may occur during adaptation of software for further customers, which is necessary from the economic point of view.

Therefore, modern software engineering requires methods proving the different descriptions of software and tools generating one specification from the other automatically or in a dialogue with the sotware engineer.

2.2. Specification Methods

2.2.1. Fundamentals of Specification Methods

Nowadays, some methods well known from semantic definition of programming languages can be found in software specification reports. This is not surprising because an exact semantic definition of a programming language will also define the semantics of all its programs. Therefore, it is logical to use well known sematic definition methods for describing software.

Beside specification in natural languages, algebraic, logical and denotational specifications are the fundamentals of many methods.

To illustrate the principles of these methods we will use a simple example, the modified telegram problem. The original problem was already studied in [Heh 83], [Jac 75], [Noo 75] and [Rec 84].

2.2.2. Natural Language Specification

Specifications in natural languages are easily understood, whereas formal languages are more difficult to understand. But natural languages have the disadvantage, that they are ambiguous. Therefore it is very difficult to write down a precise specification in a natural language.

Natural Language Specification of a Simple Telegram Problem

A program is required to process a stream of telegrams. This stream is available as a sequence of words and spaces. The stream is terminated by the occurrence of the empty telegram. Each telegram is delimited by the symbol "$****$". The telegrams are to be processed to determine for each telegram the number of words with more than twelve and the number of words with less than twelve characters. The telegrams together with the statistics have to be stored on an outputfile by eliminating all but one space between the words. The longest possible word has twenty characters. For simplicity telegram streams containing telegrams with words longer than twenty characters are omitted.

2.2.3. Logical Specification

Algebraic specifications conceive programs as being abstract data types and therefore heterogeneous universal algebras. According to the static character of algebras the specification of a program means to define the structure of input and output data and the ralations between them. For this purpose sorts representing data types, operation symbols describing the "rough" structure of data and auxiliary operation symbols describing "details" of data are introduced.

Properties of operations assigned to operation symbols in concrete algebras, and therefore properties of data, are described by axioms (systems of term equations).

Auxiliary operation symbols and axioms are also used for describing the dependencies between input and output data.

We can get a shorter description using a method known from the definition of programming languages (see e.g. [Rie 85]). Then the input telegram stream is considered as a program. The concrete semantic meaning is the output telegram stream. In this case the structure of the input telegram streams is defined only. By axioms all input streams are grouped into equivalence classes. Each class represents the abstract semantic meaning of the elements of the class. Therefore the axioms must be such that a class contains all input telegram streams differing only by the number of spaces between the words. Furthermore a class must be defined containing all erroneous input telegram streams.

Fundamental work was done by Goguen and Thatcher [Gog 74] and also by Guttag [Gut 75]. The shown approaches are not the only ones. Other possible algebraic approaches are proposed for example by Guttag and Horning [Gut 78] using routines or by Mallgren [Mal 80] using event algebras.

Algebraic Specification of the Simple Telegram Problem Sorts:

C = Character Data Type
W = Word Data Type
WS = Word Sequence Data Type
T = Input Telegram Data Type
TS = Input Telegram Sequence Data Type
S = Input Telegram Stream Data Type
OT = Output Telegram Data Type
OTS = Output Telegram Sequence Data Type
OS = Output Telegram Stream Data Type
P = Program Data Type
N = Integer Data Type
B = Boolean Data Type

Operation Symbols

Formally only operation symbols are defined here. But our comments will already give an interpretation to increase the readability of the definitions.

char: $\rightarrow$ char
{char $\in \{A, \ldots, Z, a, \ldots, z, 0, \ldots, 9\}$
constructs the corresponding character.}

dig: $\rightarrow$ dig
{dig $\in \{0, \ldots, 9\}$}

bool: $\rightarrow$ bool
{bool $\in$ {true, false}}

sword: $C \rightarrow W$
{Constructs a simple word consisting of one character only.}

word: $W \times C \rightarrow W$
{Constructs a word by concatenating a word and a character.}

sseq: $W \rightarrow WS$

{Constructs a simple word sequence consisting of one word only.}

seq: $WS \times W \rightarrow WS$

{Constructs a word sequence by concatenating a word sequence, a space and a word.}

eseq: $WS \rightarrow WS$

{Constructs a word sequence by appending one space to a word sequence.}

tel: $WS \rightarrow T$

{Constructs from a word sequence a telegram by appending "∗∗∗∗".}

stseq: $T \rightarrow TS$

{Constructs a simple telegram sequence consisting of one telegram only.}

tseq: $TS \times T \rightarrow TS$

{Constructs a telegram sequence by appending one telegram to a telegram sequence.}

stream: $TS \rightarrow S$

{Constructs from a telegram stream an input stream by appending "∗∗∗∗", the empty telegram.}

outtel: $WS \times N \times N \rightarrow OT$

{Constructs an output telegram by composing a word sequence with the integers for short and long words.}

sotseq: $OT \rightarrow OTS$

{Constructs a simple output telegram sequence consisting only of one output telegram.}

outseq: $OTS \times OT \rightarrow OTS$

{Constructs an output telegram sequence by appending one output telegram to an output telegram sequence.}

outstream: $OTS \rightarrow OS$

{Constructs from an output telegram sequence an output telegram stream.}

prog: $S \times OS \rightarrow P$

{Composes an input telegram stream and an output telegram stream to a program.}

length: $W \rightarrow N$

{For a given word the number of characters is delivered.}

succ: $N \rightarrow N$

add: $N \times N \rightarrow N$

eq: $N \times N \rightarrow B$

le: $N \times N \rightarrow B$

{These are the well known successor, addition, equal and less or equal operators.}

Auxiliary operation symbols

isword: $W \rightarrow B$

isws: $WS \rightarrow B$

istel: $T \rightarrow B$

ists: $TS \rightarrow B$

iss: $S \rightarrow B$

isotel: $OT \rightarrow B$
isots: $OTS \rightarrow B$
isos: $OS \rightarrow B$
isprog: $P \rightarrow B$

{These operators deliver the value true if the corresponding word, word-sequence, telegram, telegram sequence, telegram stream, output telegram, output telegram sequence, output telegram stream or program are well formed; otherwise false is delivered.}

isderts: $TS \times OTS \rightarrow B$

{If the output telegram stream is derivable from the telegram stream true is delivered otherwise false}

isderws: $WS \times WS \rightarrow B$

{If the first wordsequence is derivable from the second one then true otherwise false is delivered.}

is: $W \times W \rightarrow B$

{If the two words are identical the operation delivers true otherwise false.}

isshort: $W \rightarrow B$

{If the number of characters of the word is less or equal twelve then true otherwise false is delivered.}

Axioms:

{X_S means a variable of sort S.}

isprog (stream (X_{TS}), outstream (X_{OTS})) = iss (stream $(X_S)_T$)
& isos (outstream (X_{OTS}))
& isderts (X_{TS}, X_{OTS})
{A program is well formed if the input telegram stream and the output telegram stream are well formed and the output telegram sequence is derivable from the input telegram sequence.}

iss (stream (X_{TS})) = its (X_{TS})
{A telegram stream is well formed if the corresponding telegram sequence is well formed.}

ists (tseq (X_{TS}, X_T)) = ists (X_{TS}) & istel (X_T)
ists (stseq (X_T)) = istel (X_T)
{A telegram sequence is well formed if it consists of a telegram sequence and a telegram and both are well formed. It is also well formed if it consists of one well formed telegram only.}

istel (tel (X_{WS})) = isws (X_{WS})
{A telegram is well formed if the corresponding word sequence is well formed.}

isws (eseq (X_{WS})) = isws (X_{WS})
isws (seq (X_{WS}, X_W)) = isws (X_{WS}) & isword (X_W)
isws (sseq (X_W)) = isword (X_W)
{A word sequence is well formed if it consists of a well formed word sequence followed by a space. It is also well formed if it consists of a word sequence and a word and both are well formed.
If it consists of a word only and this word is well formed the word sequence is well formed too.}

isword (X_W)=le (length (X_W), succ^{20} (0))
{A word is well formed if its length is less than or equal to twenty.}
length (sword (X_C))=succ (0)
length (word (X_W, X_C)=add (length $(X_W,)$ succ (0))
{The length of the simple word consisting only of one character is one and the length of a word consisting of a word followed by a character is the length of this word plus one.}
isos (outstream (X_{OTS}))=isots (X_{OTS})
{An output telegram stream is well formed if the corresponding output telegram sequence is well formed.}
isots (sotseq (X_{OT}))=isotel (X_{OT})
isots (otseq (X_{OTS}, X_{OT}))=isots (X_{OTS}) & isotel (X_{OT})
{An output telegram stream is well formed if it consists of one well formed output telegram only. It is also well formed if it consists of an output telegram sequence and an output telegram and both are well formed.}
isotel (outtel (sseq $(X_W,)$ X_N, Y_N)=isshort (X_W) & eq $(X$, succ (0))
& isword (X_W) & eq $(Y_N, 0)$
{If the output telegram consists of one short word only then the number of short words is one and the number of long words is zero.}
isotel (outtel (sseq (X_W), X_N, Y_N)=(¬isshort (X_W)) & isword (X_W)
& eq$(X_N, 0)$ & eq m $(Y_N$, succ (0))
{If the output telegram consists only of one word, which is a well formed word but not a short word, then the mumber of short words is zero and the number of long words is one.}
isotel (outtel (seq (X_{WS}, X_W), add $(X_N$, succ (0)), Y_N)=
isotel (outtel (X_{WS}, X_N, Y_N) & isshort (X_W) & isword (X_W)
{If an output telegram consists of a word sequence and a short word and the telegram of the word sequence is a well formed output telegram then the number of short words of the whole telegram is equal to the number of short words of this telegram plus one. The number of long words is the same in both telegrams.
isotel (outtel (seq (X_{WS}, X_W), X_N, add $(Y_N$, succ (0)))=
isotel (outtel (X_{WS}, X_N, Y_N) & (¬isshort (X_W))
& isword (X_W)
isotel (outtel (eseq (X_{WS}), X_N, Y_N))=false
{An output telegram cannot contain more than one space between the words.}
isotel (outtel (seq (X_{WS}, X_W), 0, 0))=false
{An output telegram composed of a word sequence and a word cannot have zero short and zero long words.}
isshort (X_W)=le (length (X_W), succ^{12} (0))
{A word is short if it has less than or equal to twelve characters.}
isderts (stseq (tel (X_{WS})), sotseq (outtel (Y_{WS}, X_N, Y_N))=
isderws (X_{WS}, Y_{WS})
isderts (tseq $(X_{TS}$, tel (X_{WS})), otseq $(X_{OTS}$, outtel $(Y_{WS}, X_N, Y_N)))$=
isderts (X_{TS}, X_{OTS}) & isderws (X_{WS}, Y_{WS})
isderts (tseq (X_{TS}, X_T), sotseq (X_{OT}))=false
isderts (stseq (X_T), otseq (X_{OTS}, X_{OT}))=false
{An output telegram stream is derivable from a telegram stream if both consists of one telegram only and the corresponding word sequences are derivable. It is

also derivable if both streams consist of a telegram stream and a word sequence and they are derivable correspondingly. But an output stream is not derivable from an input stream if one of them is a sequence and the other one is a simple sequence.}

isderws (sseq (X_W), sseq (Y_W)) = is (X_W, Y_W)
isderws (seq (X_{WS}, X_W), seq (Y_{WS}, Y_W)) = isderws (X_{WS}, Y_{WS}) & is (X_W, Y_W)
isderws (eseq (X_{WS}), Y_{WS}) = isderws (X_{WS}, Y_{WS})
isderws (seq (X_{WS}, X_W), sseq (Y_W) = false
isderws (sseq (X_W), seq (Y_{WS}, Y_W) = false

{An output word sequence is derivable from an input word sequence if both consist of one word only and these words are identical. It is also derivable if both sequences consist of a word sequence and a word and the word sequences are derivable and the words are identical. If an output word sequence is derivable from an input word sequence then the output sequence is also derivable from the input word sequence followed by a space. If one sequence is a simple sequence and the other is a sequence then the output sequence is not derivable from the input sequence.}

Because of readability the axioms for "is" are omitted. They can be built straightforward. "eq", "le", "succ" and "and" are assumed to be standard operations.

2.2.3. Logical Specification

Logical specifications are based on predicate calculus. Well known approaches are the axiomatic approach introduced by Hoare [Hoa 69] to define the semantics of programming languages and the logical programming using the programming language PROLOG [Kow 74].

Hoare's axiomatic approach

To define the semantics of a programming language Hoare uses specifications. A specification is a string of the form

$$\{A\}\, p\, \{B\},$$

where p is a part of a program, and A and B are formulas which can be interpreted as assertions. Therefore the above specification could be read as:

"If A is valid before execution of p and p is finished then B is valid after execution of p (partial correctness)."

To specify a programming language one needs a finite system of specifications consisting of axioms and rules of inference.

In the case of our simple telegram problem we consider a stream of input telegrams as a program. The semantics of this program is defined by the corresponding stream of output telegrams. Such a "program" consists of elements from the set

$$C = \{"A", \ldots, "Z", "a", \ldots, "z", "0", \ldots, "9", "\#", "****"\}$$

(# represents one space,
**** represents the end of a telegram)

and of the (invisible) concatenation operator which concatenates a part of an input stream with an element into a new part of an input stream.

For the definition of the semantics we need some auxiliary variables:

output — Part of the output stream corresponding to the treated part of the input stream,
length — Actual number of characters in the actually treated word,
short — Actual number of short words in the actually treated telegram,
long — Actual number of long words in the actually treated telegram.
telno — Number of already treated telegrams.

length $= -1$ means the last telegram of the input stream was treated. length $= 0$ means the end of a word was treated. The quintuple (output, length, short, long, telno) represents "evaluation" states of our "programs". That means a component of a given input stream of telegrams transforms a given quadruple into a new one. Particularly, a given input stream of telegrams transforms the quintuple (empty, 0, 0, 0, 0) into a quintuple with the sought output telegram stream as its first component.

In the following system of axioms and rules of inference we omit axioms and rules for formulas supposing that all formulas are well defined.

Axioms:

$$\{A\}c\{B\}$$

$$A = \text{output} = o \,\&\, \text{length} = l \,\&\, l \geqq 0$$

$$B = \text{output} = o.c \,\&\, \text{length} = 1+1$$

$$c \in \{"A", \ldots, "Z", "a", \ldots, "z", "0", \ldots, "9"\}$$

The last telegram has not been treated. Therefore the output stream is concatenated with the new element c. ("." means concatenation)

$$\{A1\} \# \{B1\}$$

$$\{A2\} \# \{B2\}$$

$$\{A3\} \# \{B3\}$$

$A1 = 12 < \text{length} \leqq 20$ & long $= l$ & output $= o$
$B1 = \text{length} = 0$ & long $= l+1$ & output $= o.\#$
$A2 = 0 < \text{length} \leqq 12$ & short $= s$ & output $= o$
$B2 = \text{length} = 0$ & short $= s+1$ & output $= o.\#$
$A3 = \text{length} = 0$
$B3 = \text{length} = 0$

These axioms count the number of short and long words in the actually treated telegram. The last axiom secures that only one space occurs between words of the output stream.

$$\{A1\} * * * * \{B1\}$$

$$\{A2\} * * * * \{B2\}$$

$A1$=(short>0 or long>0) & length=0 & output=o
& telno=t
$B1$=short=0 & long=0 & telno=t+1
& output=o. ∗∗∗∗. # .short. # .long. #
$A2$=short=0 & long=0 & length=0 & output=0
& telno notequal 0
$B2$=output=o. ∗∗∗∗. # .0. # .0. # & length=−1

In the first axiom the actual telegram is finished. For this telegram the number of short and long words is concatenated to the output stream.

In the second axiom the last telegram is treated and output contains the output telegram stream.

Rule of inference:

$$\frac{\{P\}p1\{Q1\},\ \{Q1\}p2\{Q\}}{\{P\}p1\,p2\{Q\}},$$

where $p1$ is a part of an input stream and

$$p2\in\{"A", \ldots, "Z", "a", \ldots, "z", "0", \ldots, "9", "\#", "\ast\ast\ast\ast"\}.$$

This rule enables the composition of specifications and thereby the construction of specifications for input streams of telegrams.

Specification of the simple telegram problem using predicate calculus

For the definition of the simple telegram problem we use now a finite system of Horn clauses (see e.g. [Loy 84]). A Horn clause is a string of the form

$$B \leftarrow A_1, \ldots, A_n, \qquad (*)$$

where B, $A_1, \ldots, A_n$ are atoms.

An atom consists of a n-ary predicate symbol followed by a list of n terms inserted in paranthesis. Let $x_1, \ldots, x_K$ be the only variables occurring in the terms of $B, A_1, \ldots, A_n$. Then $(*)$ means

$$(V x_1, \ldots, x_k)(A_1 \& \ldots \& A_n \Rightarrow B).$$

By usual interpretation of formulas we get:
For all values of the variables $x_1, \ldots, x_k$ such that all $\bar{A}_i$ are valid $\bar{B}$ is valid too ($\bar{A}_i$ and $\bar{B}$ are assertions arising from A_i and B respectively).

For the definition of facts one uses a special kind of Horn clauses:

$$B \leftarrow .$$

Horn clauses for the simple telegram problem (For simplicity we omit the definition of some clauses referring to natural numbers and the concatenation operation in terms.):

To achieve better readability of the clauses we will first give an interpretation of the atoms.

sum (X, Y, Z) — Z is the sum of X and Y.
greater (X, Y) — X is greater than Y.
lessequal (X, Y) — X is less than or equal to Y.
character (X) — X is a character.
word (X, L) — X is a word of length L.
telegram (X, Y) — Y is the output telegram corresponding to the input telegram X.
telegramstream (X, Y) [telstr (X, Y)] — Y is the output telegram stream corresponding to the input telegram steream.

character (X) ←. $X \in \{$"A", ..., "Z", "a", ..., "z", "0", ..., "9"$\}$
word $(X, 1)$ ← character (X).
word $(XY, L1)$ ← character (X), word (Y, L), sum $(L, 1, L1)$.
telegram $(\#X, Y)$ ← telegram (X, Y).
telegram $(X\#Y, X\#Z\#S\#L1\#)$ ← word $(X, L0)$,
tel $(Y, Z\#S\#L\#)$,
greater $(L0, 12)$,
lessequal $(L0, 20)$,
sum $(L, 1, L1)$.
telegram $(X\#Y, X\#Z\#S1\#L\#)$ ← word $(X, L0)$,
tel $(Y, Z\#S\#L\#)$,
lessequal $(L0, 12)$,
sum $(S, 1, S1)$.
tel $(****, ****\#0\#\ 0\#)$←.
tel (X, Y) ← telegram (X, Y).
telegramstr $(****, ****\#0\#0\#)$←.
telegramstr (XI, YO) ← telegram (X, Y),
telegramstr (I, O).
telegramstream (XI, YO) ← telegram (X, Y)
telegramstr (I, O).

For a given input telegram stream Ti the corresponding output telegram stream is determined (if it exists) beginning from the goal telegramstream (Ti, Y). This is done by constructing a prove for the goal with the Horn clauses. The variables of the Horn clauses are suitably substituted. The determined value of the variable Y in the goal is the sought output telegram stream.

2.2.5. Denotational Specification

The objective of denotational specification is to think of programs as being functions which transform input values into output values. However unlike a program which specifies how to compute the function, the denotational specifications merely indicate which function the program should compute.

The fundamentals of this theory were developed by Scott and Strachey [Sco 71] to define the semantics of programming languages. They were further developed by Stoy [Sto 77].

To specify our simple telegram problem we will use the meta-language of the Vienna Development Method. (see e.g. [Bjø 78])

Analogously to the axiomatic approach the starting point is to consider an input telegram stream as a program. The semantic meaning of this program is the corresponding output telegram stream. Now, the denotational approach requires the definition of syntactical and semantical domains and the definition of functions determining the semantic meaning of program parts. Furthermore, we need some functors for "pasting together" semantic functions.

Denotational Specification of the Simple Telegram Problem Syntactic Domains:

Program = Prog.
Prog::Stream Endword.
Stream = Tel | Telseq.
Tel::Wordseq Endword.
Telseq::Stream Tel.
Wordseq = Word | Words.
Word::Charstr Spaces.
Words::Wordseq Word.
Charstr = Character | Characters.
Character = $A|...|Z|a|...|z|0|...|9$.
Characters::Charstr Character.
Spaces = Space|Spaceseq.
Space = #.
Spaceseq::Spaceseq Space.
Endword = ∗∗∗∗.

Sematic Domains:
OUTTELSTREAM::OUTTEL*
OUTTEL::WORD* END SPACE INT SPACE INT SPACE
WORD::CHARACTER* SPACE
CHARACTER = {A, ..., Z, a, ..., z, 0, ..., 9}
INT = {0, 1, 2, 3, ...}
SPACE = #
END = ∗∗∗∗∗

Elaboration Functions:

type: eval-program:	Program	→	OUTTELSTREAM
type: eval-orog:	Prog	→	OUTTELSTREAM
type: eval-stream:	Stream	→	OUTTELSTREAM
type: eval-tel:	Tel	→	OUTTEL
type: eval-wordseq:	Wordseq	→	OUTTEL
type: eval-word:	Word	→	INT×INT
type: eval-charstream:	Charstream	→	INT
type: eval-characters:	Characters	→	INT

eval-program (p)∧eval-prog (p)
{The result of the function eval-program (p) is the result of the function eval-prog (p).}

eval-prog (mk-prog (*s*, *e*))$\underline{\wedge}$

let *v*1 = eval-stream (*s*)
  *v*2 = # ∗ ∗ ∗ ∗ ∗ # 0 # 0 #
in *v*1 . *v*2

{A program *p* consists of a stream *s* and an end word *e*. The result of eval-prog (*p*) is equivalent to the result of eval-stream (*s*) concatenated by the empty telegram.}

eval-stream (*st*)$\underline{\wedge}$

cases *st*: mk-tel (*ws*, *e*) → eval-wordseq (*ws*),
  mk-telseq (*s*, *t*) → **let** *v*1 = eval-stream (*s*),
    *v*2 = eval-tel (*t*)
  in *v*1 . *v*2

{If the stream *st* consists of a word sequence *ws* and the end word *e* the result of eval-stream (*st*) is the result of eval-wordseq (*ws*).
But if *st* consists of a stream *s* and a telegram *t* eval-stream (*st*) is equal to the concatenation of the results of eval-stream (*s*) and eval-tel (*t*).}

eval-tel (mk-tel (*ws*, *e*))$\underline{\wedge}$eval-wordseq (*ws*)

{A telegram consists of a word sequence *ws* and an end word *e*. The result of eval-tel (*t*) is equal to the result of eval-wordseq (*ws*).}

eval-wordseq (*ws*) $\underline{\wedge}$

cases *ws*: mk-word (*w*) → **let** (*x*, *y*) = eval-word (*w*)
    in *w* ∗ ∗ ∗ ∗ ∗ # *x* # *y* #
  mk-words (*sw*, *w*) → **let**
    (*u*, *v*) = eval-word (*w*),
    x ∗ ∗ ∗ ∗ ∗ # *y* # *z* # = eval-wordseq (*sw*)
    in *xw* ∗ ∗ ∗ ∗ ∗ # *u* + *y* # *v* + *z* #

{If the word sequence *ws* consists of one word *w* only the result of eval-wordseq (*ws*) is the composition of *w* and the result of eval-word (*w*).
If *ws* consists of a wordsequnce *sw* and a word *w* the result of eval-wordseq (*ws*) is a composition of the results of eval-word (*w*) and eval-wordseq (*sw*).}

eval-word (mk-charstr (*cs*, *s*))$\underline{\wedge}$

if eval-charstr (*cs*) ≦ 12
then (1, 0)
else
if eval-charstr (*cs*) ≦ 20
then (0, 1)
else undefined

{A word consists of a character stream *cs* and a space sequence *s*. The result of eval-word (*w*) is equal to (1, 0) or (0, 1) depending on the result of eval-chararstr (*cs*).}

eval-charstr (*cs*)$\underline{\wedge}$

cases *cs*: mk-character (*c*)→1,
mk-characters (*c*)→eval-characters (*c*)
{If the character stream *cs* consists of a character *c* the result of eval-charstr (*cs*) is equal to one.
If *cs* consists of characters *c* then the result of eval-charstr (*cs*) is equal to the result of eval-characters (*c*).}

eval-characters (mk-characters (*cs*, *c*))$\underline{\wedge}$

let x=eval-charstr (*cs*)
in $x+1$

{Characters *ca* consist of a character stream *cs* and a character *c*. The result of eval-characters (*ca*) is equal to the result of eval-charstr (*cs*) plus one.}

2.2.6. Relations Between Fundamentals of Problem, Program and Implementation Specification

A selection of some references related to the topic is given in Figure 2.

According to the methods for problem and program specification there are many tools for implementation specification, which are also influenced by mathematics. Figure 2 also presents a classification of some of these tools.

There is no fixed way from program specification to programming. The software engineer can choose all programming methods for every specification.

But modern methodologies of software engineering try to unify the descriptions during the whole life cycle. Especially the implementation specification attains a higher level of abstraction and has the form of program or problem specification.

There is a good success in logical programming (e.g. PROLOG), in programming by grammars (e.g. CDL, HFP) and in programming on a very high level (e.g. MODEL).

All modern methods have in common, that the software engineer is not confronted with all implementation details. In many cases he does not know them.

He can concentrate upon the main design principles. The details are generated automatically (artificial intelligence) or are already implemented (abstract data types).

In our opinion programming by grammar will be more important in future. Watt and Madsen [Wat 81] have shown for example, that algebraic, logical and denotational specification can be expressed by extended attribute grammars.

2.2.7. Grammatical Specification of the simplified telegram problem

First, we will informally introduce the notion of attribute grammars on the basis of grammars of syntactic functions [Rie 83].
An attribute grammar is a contextfree grammar

$$G = (V, T, S, P)$$

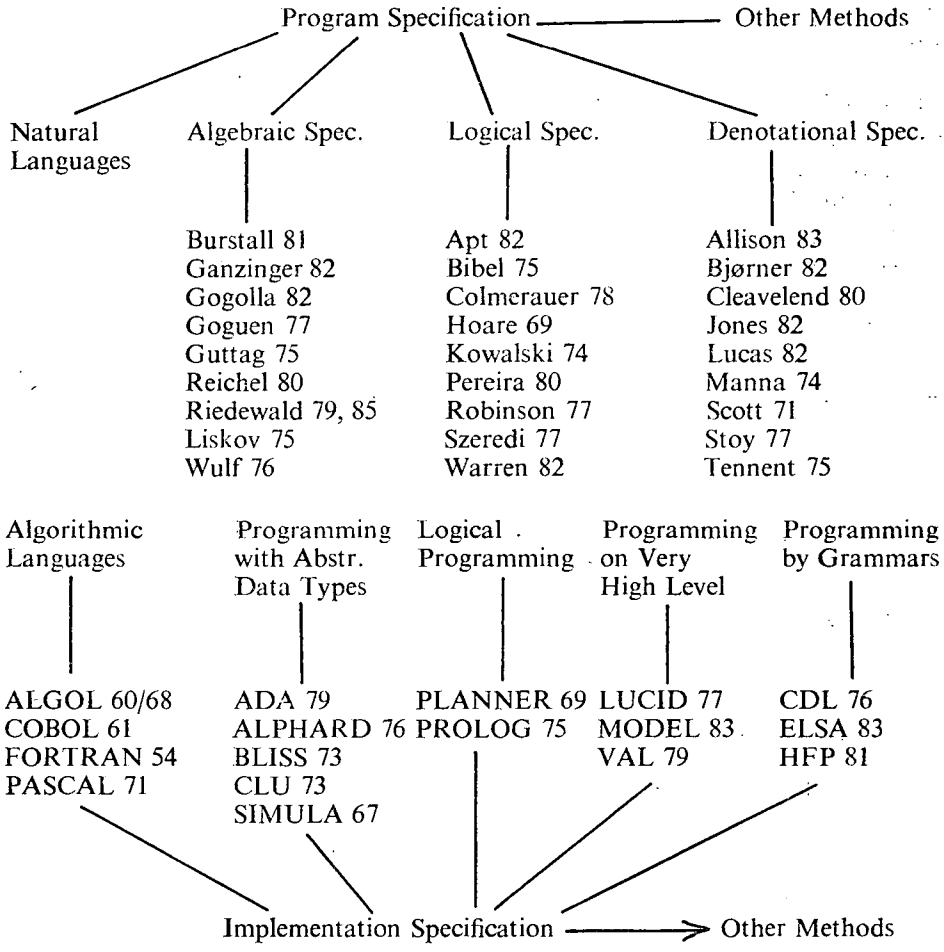

Figure 2. Classification of Problem, Program and Implementation Specification Methods and Tools

V-vocabulary, T-set of terminals, N-set of nonterminals
$V=N\cup T$, S-start element, P-set of production rules
augmented with parameters, auxiliary syntactic functions and semantic functions. Auxiliary syntactic functions are necessary to describe the static semantic.

The rules have the form

$$f(p_1^f, \ldots, p_{n_f}^f) ::= f_1(p_1^{f_1}, \ldots, p_{n_{f_1}}^{f_1}) \ldots f_r(p_1^{f_r}, \ldots, p_{n_{f_r}}^{f_r})$$

$$H_1(p_1^{H_1}, \ldots, p_{n_{H_1}}^{H_1}) \ldots H_k(p_1^{H_k}, \ldots, p_{n_{H_k}}^{H_k}).$$

$$f \in N, f_1, \ldots, f_r \in V,$$

$H_1, \ldots, H_k \in$ {auxiliary syntactic functions} $\cup$ {semantic functions}, $p_y^x \in$ set of parameters, $x, y, r, k \in$ set of integers.

The telegram problem can be specified in different ways by a grammar. The following two methods are possible:

1. The input and output telegram streams are described by parameters. The start symbol has the input telegram stream as input parameter and delivers the output telegram stream as the value of the output parameter.
2. The input telegram stream is described by a context-free grammar and this grammar is augmented by parameters and functions in such a way that the start element of the grammar delivers the output telegram stream as the value of the parameter of of the start symbol.

The first method would result in a grammar very similar to our specification using Horn clauses. Therefore we will omit this example here. The interested reader will very easily get such a grammar.

Let us demonstrate the second method in full detail.

Grammatical Specification of the Simple Telegram Problem (using the second method)

I. *Semantic functions*

— CAT2 ($\downarrow S1, \downarrow S2, \uparrow S$)
This function concatenates $S2$ to $S1$ and delivers S.
— CAT3 ($\downarrow S1, \downarrow S2, \downarrow S3, \uparrow S$)
This function delivers $S1.****.\#.S2.\#S3.\#$ in S.
— COUNT ($\downarrow L, \downarrow Long1, \downarrow Short1, \uparrow Long, \uparrow Short$)
IF $L<12$ THEN $Long := Long1$; $Short := Short1+1$
ELSE $Long := Long1+1$; $Short := Short1$
FI
— ADD ($\downarrow A, \downarrow B, \uparrow C$)
$C := A+B$

II. *Auxiliary Syntactic functions*

— OVERLENGTH ($\downarrow L$)
The actions of the parser are influenced by this function. The application of the corresponding rule is possible if L is less or equal to 20 only.

III. *Production rules*

1. Program ($\uparrow Outstream$)::=Telegramstream ($\uparrow Out$)
"#" Endsymbol
CAT3 ($\downarrow Out, \downarrow$ "0", $\downarrow$"0", $\uparrow Outstream$).
2. Telegramstream ($\uparrow O$)::=Telegram ($\uparrow O$).

3. Telegramstream (↑*O*)::=Telegramstream (↑*O*1)
   Telegram (↑*O*2)
   CAT2 (↓*O*1, ↓*O*2, ↑*O*).
4. Telegram (↑*O*)::= Wordsequence (↑*O*1, ↑*Short*, ↑*Long*)
   Endsymbol
   CAT3 (↓*O*1, ↓*Short*, ↓*Long*, ↑*O*).
5. Wordsequence (↑*O*, ↑*Short*, ↑*Long*)::=
   Word (↑*O*, ↑*Length*)
   OVERLENGTH (↓*Length*)
   COUNT (↓*Length*, ↓"0", ↓"0", ↑*Short*, ↑*Long*).
6. Wordsequence (↑*O*, ↑*Short*, ↑*Long*)::=
   Wordsequence (↑*O*1, ↑*Short*1, ↑*Long*1)
   Word (↑*O*2, ↑*Length*)
   OVERLENGTH (↓*Length*)
   COUNT (↓*Length*, ↓*Short*1, ↓*Long*1,
   ↑*Short*, ↑*Long*
   CAT2 (↓*O*1, ↓*O*2, ↑*O*).
7. Word (↑*Word*, ↑*L*)::= Charactersequence (↑*Word*, ↑*L*)
   Spacesequence.
8. Charactersequence (↑*C*, ↑"1")::=Character (↑*C*).
9. Charactersequence (↑*Char*, ↑*Length*)::=
   Charactersequence (↑*Char*1, ↑*Length*1)
   Character (↑*Char* 2)
   CAT2 (↓*Char*1, ↓*Char*2, ↑*Char*)
   ADD (↓*Length*1, ↓"1", ↑*Length*).
10. Character (↑"*A*")::="*A*".
    ...
35. Character (↑"*Z*")::="*Z*".
36. Character (↑"*a*")::="*a*".
    ...
51. Character (↑"*z*")::="*z*".
52. Character (↑"0")::="0".
    ...
62. Character (↑"9")::="9".
63 Endsymbol::="∗∗∗∗"..
64. Spacesequence::="#".
65. Spacesequence::=Spacesequence "#".

2.3. Programming with Production Rules

The relations of methods, which were developed independently for using production rules in programming, are the result of current research.

Figure 3 shows some interesting relations between attribute grammars and logical programming.

The world wide interest in logical programming and the relations of Figure 3 support our opinion to study applications of attribute grammars in software engineering.

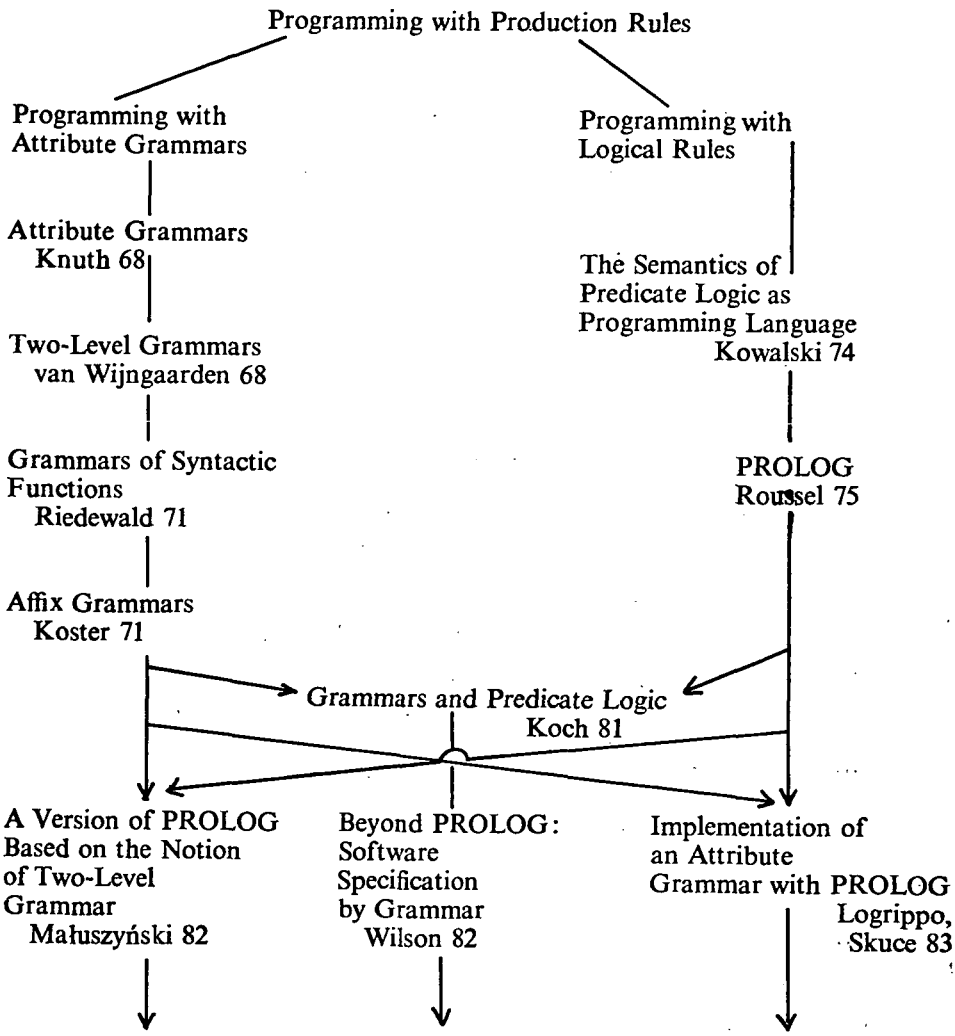

Figure 3. Some Relations Between Programming Methods Using Production Rules

2.4. Some Software Development Methods and Tools with Different Use of Data and Information

We want to discuss some other arguments supporting the application of attribute grammars in software development. Let us first have a look at some methods and tools supporting the use of data and information in different kinds. Figure 4 is an attempt to classify some methods and tools. Such tools can also be used in another way, of course, but they were mainly designed for this purpose.

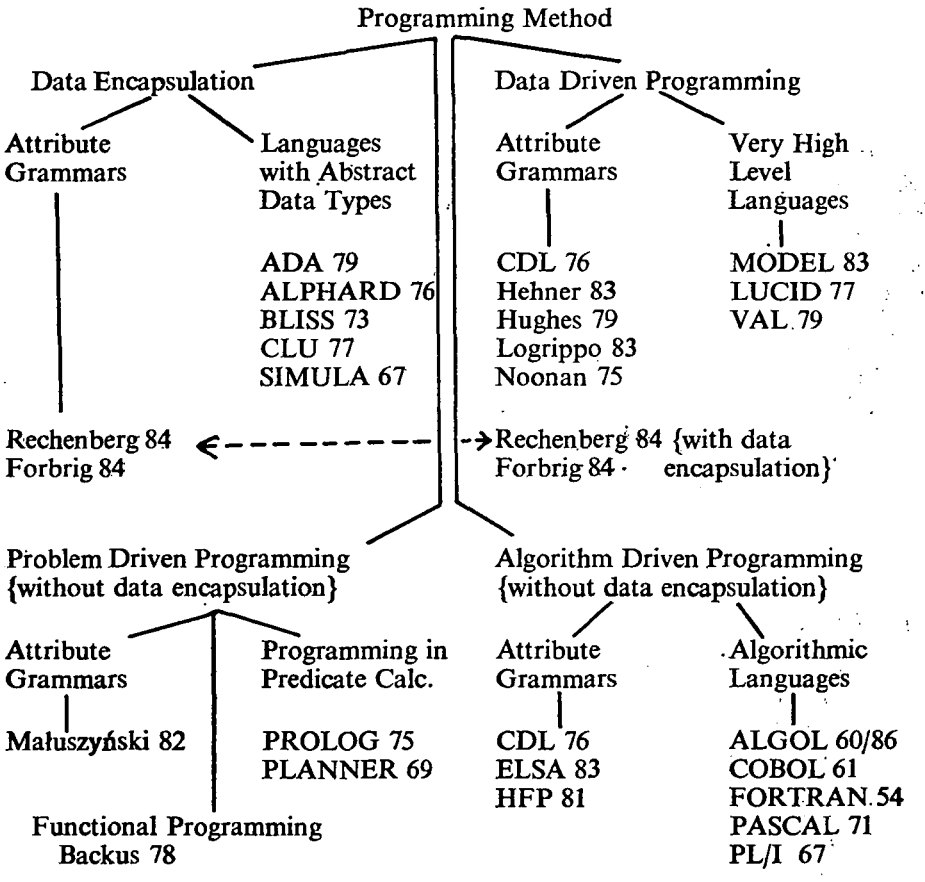

Figure 4. Classification of some Programming Methods and Tools

The following results can be obtained from Figure 4:

1. Attribute grammars are applied in all classified fields.
2. The programming language CDL can be used in a data driven and algorithm driven way.
3. Attribute grammars can be combined with data encapsulation.
4. Attribute grammars can be used to combine data driven programming and data encapsulation.

The method of data driven programming by attribute grammars with abstract data types was discussed in Rostock in [For 84 a]. Some similar results can be found in [Rec 84]. Rechenberg proposes attribute grammars mainly as a tool for program specification. The implementation is suggested by top-down programming.

We will mainly use attribute grammars as input for a translator writing system. Examples of data driven data processing using attribute grammars with abstract data types will be presented in the following section.

2.5. Data Driven Programming with Abstract Data Types

2.5.1. Attribute Grammars and Abstract Data Types

We will extend the definition of an attribute grammar by abstract data types. The context-free grammar is not only augemented with parameters, semantic functions and syntactic functions but also with functions of abstract data types.

That means

$H_1, \ldots, H_k \in$ {semantic functions} $\cup$ {syntactic functions} $\cup$ {functions of abstract data types}.

These grammars are more effective to implement, because not all information has to be transfered a long way via parameters.

In our opinion attribute grammars with abstract data types are better to read and write. They can better be maintained.

2.5.2. Examples

2.5.2.1. Grammatical Specification of the Simple Telegram Problem

I. *Abstract data types*

* File of output telegram stream with statistics
— OPEN-OUTFILE, CLOSE-OUTFILE
These functions open and close the file.
— OUTWORD (↓*Word*, ↓*Length*)
The "*Word*" with given length is stored on the file.
— OUTCOUNT (↓*Short*, ↓*Long*)
The number of long and short records of the current telegram are stored on the file according to the specification. (∗∗∗∗. # .*Short*. # *Long*. #)

II. *Semantic functions*

We use the semantic functions of our example in section 2.2.7.

III. *Aixiliary syntactic functions*

We use the syntactic functions of the example in section 2.2.7.

IV. *Production rules*

1. Program::=Begin Stream CLOSE-OUTFILE.
2. Begin::=OPEN-OUTFILE.
3. Stream::=Telegramstream. "#" Endsymbol
OUTCOUNT (↓"0", ↓"0").
4. Telegramstream::=Telegram.
5. Telegramstream::=Telegramstream Telegram.

6. Telegram::=Wordsequence (↑*Short*, ↑*Long*)
   Endsymbol
   OUTCOUNT (↓*Short*, ↓*Long*).
7. Wordsequence (↑*Short*, ↑*Long*)::=Word (↑*Word*, ↑*L*)
   OVERLENGTH (↓*L*)
   COUNT (↓*L*, ↓"0", ↓"0", ↑*Short*, ↑*Long*)
   OUTWORD (↓*Word*, ↓*L*).
8. Wordsequence (↑*Short*, ↑*Long*)::=
   Wordsequence (↑*Short*1, ↑*Long*1)
   Word (↑*Word*, ↑*L*)
   OVERLENGTH (↓*L*)
   COUNT (↓*L*, ↓*Short*1, ↓*Long*1, ↑*Short*, ↑*Long*)
   OUTWORD (↓*Word*, ↓*L*).
9. Word (↑*Word*, ↑*L*)::=Charactersequence (↑*Word*, ↑*L*)
   Spacesequence.
10. Charactersequence (↑*C*, ↑"1")::=Character (↑*C*).
11. Charactersequence (↑*C*, ↑*L*)::=
    Charactersequence (↑*C*1, ↑*L*1)
    Character (↑*C*2)
    CAT2 (↓*C*1, ↓*C*2, ↑*C*)
    ADD (↓*L*1, ↓"1", ↑*L*).
12. Character (↑"A")::="A".
    ...
37. Character (↑"*Z*")::="*Z*".
38. Character (↑"*a*")::="*a*".
    ...
53. Character (↑"*z*")::="*z*".
54. Character (↑"0") :="0".
    ...
64. Character (↑"9")::="9".
65. Endsymbol::="∗∗∗∗".
66. Spacesequence ::="#".
67. Spacesequence ::=Spacesequence "#".

If standard technices are used as subprograms for lexical analysis only the first eight production rules of the grammar are necessary.

2.5.2.2. Grammatical Specification of a Very Little Commercial Project

The following task has to be fulfilled by a computer: A special master file contains data of all wage-earners of an enterprise. Another file contains monthly data of working time and wages. These two files have to be used to produce pay slips, to remit the money through the bank and to report about working time. There is a lot of possibilities of monthly data. Therefore, every item has a key and the file contains only items different from zero. According to the four kinds of taxes used in the GDR the total sum on the pay slip is broken up into four groups. This very little commercial project can be descibed by the following attribute grammar.

I. *Abstract data types*

a) Master file with the functions:
- — OPEN-MASTER-FILE, CLOSE-MASTER-FILE
  These functions open and close the file.
- — MASTER-DATA (↓*Number*, ↑*Group*, ↑*Place*, ↑*Bank*)
  This function delivers for a given number of a wage-earner his number of the bank account, his working place and his group of professional classification. The master file data of this worker are prepared open for other functions.
- — MASTER-WAGES (↑*Money*)
  For the current wage-earner the money per hour is delivered from the master file.

b) File of working time statistics:
- — OPEN-TIME, CLOSE-TIME
- — TIME-BEGIN (↓*Group*, ↓*Place*)
  For a given group and working place the entry of statistical data is prepared.
- — TIME-KEY (↓*Key*, ↓*Hours*)
  For a group and working place fixed above the given hours are added according to the key.

c) File of data for the bank:
- — OPEN-BANK, CLOSE-BANK
- — BANK-REMIT (↓*Bank*, ↓*Amount*)
  The amount is transfered to the given bank account.

d) File of pay slips:
- — OPEN-PAY-SLIP, CLOSE-PAY-SLIP
- — PAY-SLIP-BEGIN (↓*Number*, ↓*Place*)
  An entry of data is prepared for number and place.
- — PAY-SLIP (↓*Am*1, ↓*Am*2, ↓*Am*3, ↓*Am*4, ↓*Sum*)
  The given data are entered on the file.

II. *Semantic functions*

- — ADD2 (↓*S*1, ↓*S*2, ↑*Sum*)
  $Sum := S1 + S2$
- — ADD (↓*S*1, ↓*S*2, ↓*S*3, ↓*S*4, ↑*Sum*)
  $Sum := S1 + S2 + S3 + S4$
- — MULT (↓*F*1, ↓*F*2, ↑*Product*)
  $Product := F1 * F2$
- — DEC (↓*D*1, ↓*D*2, ↓*D*3, ↓*D*4, ↓*D*5, ↑*Value*)
  $Value := (((D1 * 10 + D2) * 10 + D3) * 10 + D4) * 10 + D5$

III. *Auxiliary syntactic functions*

Auxiliary syntactic functions are not necessary for our example.

IV. *Production rules*

1. Project-run::=Begin Records CLOSE-MASTER-FILE
   CLOSE-TIME CLOSE-BANK CLOSE-PAY-SLIP.
2. Begin::=OPEN-MASTER-FILE OPEN-TIME OPEN-BANK
   OPEN-PAY-SLIP.
3. Records::=Record.
4. Records::=Records Record.
5. Record::=Head (↑*Bank*) Items (↑*Amount*)
   BANK-REMIT (↓*Bank*, ↓*Amount*).
6. Head (↑*Bank*)::="NO" Earn-No (↑*Number*)
   MASTER-DATA (↓*Number*, ↑*Group*, ↑*Place*, ↑*Bank*)
   TIME-BEGIN (↓*Group*, ↓*Place*)
   PAY-SLIP-BEGIN (↓*Number*, ↓*Place*).
7. Items (↑*Amount*)::=Amount1 (↑*Am*1) Amount2 (↑*Am*2)
   Amount3 (↑*Am*3) Amount4 (↑*Am*4)
   ADD4(↓*Am*1, ↓*Am*2, ↓*Am*3, ↓*Am*4, ↑*Sum*)
   PAY-SLIP (↓*Am*1, ↓*Am*2, ↓*Am*3, ↓*Am*4,
   ↓*Sum*).
8. Amount1 (↑*Am*1)::=Amounts1 (↑*Am*1).
9. Amount1 (↑"0")::=.
10. Amounts1 (↑*Am*1)::=*Am*1 (↑*Am*1).
11. Amounts1 (↑*Am*1)::=Amounts1 (↑*Am*2) *Am*1 (↑*Am*3)
    ADD2 (↓*Am*2, ↓*Am*3, ↑*Am*1).
12. Am1 (↑*Am*1)::="H01" Hours (↑*H*) "M01" Money (↑*M*)
    MULT (↓*H*, ↓*M*, ↑*Am*1) TIME-KEY (↓"01", ↓*H*).
13. Am1 (↑*Am*1)::="H01" Hours (↑*H*) MASTER-WAGES (↑*M*)
    MULT (↓*H*, ↓*M*, ↑*Am*1) TIME-KEY (↓"01", ↓*H*).
14. Am1 (↑*Am*1)::="FM01" Money (↑*Am*1).
15. Am1 (↑*Am*1)::="FM02" Money (↑*Am*1).
16. Earn-No (↑*Number*)::=Number5 (↑*Number*).
17. Hours (↑*H*)::=Number5 (↑*H*).
18. Money (↑*M*)::=Number5 (↑*M*).
19. Number5 (↑*V*)::=Digit (↓*D*1) Digit (↓*D*2) Digit (↓*D*3)
    Digit (↓*D*4) Digit (↓*D*5)
    DEC (↓*D*1, ↓*D*2, ↓*D*3, ↓*D*4, ↓*D*5, ↑*V*).
20. Digit (↑"0")::="0".

    ...

30. Digit (↑"9")= ::"9".

With respect to simplicity the rules of Amount2, Amount3 and Amount4 were omitted. They can be formulated similarly to the rules of Amount1.

According to rule 1 and 2 the project run consists of opening all abstract data types, interpreting a sequence of records and closing all abstract data types.

Every record has head data and items (rule 5).

The head data consist of key "NO" followed by the number of a wage-earner (rule 6). With the help of this number, data are obtained from the master file and the entry of data for statistics and the pay slip are prepared.

The items consist of four groups (rule 7). Every group can be a sequence of data (rule 11). The empty sequence is possible (rule 9).

If there are data about hours and money, multiplication is performed and the hours are reported for statistics (rule 11).

If there are only hours the money per hour is taken from the master file (rule 13). It is also possible to get money per month (rule 14, 15).

Everybody familiar with attribute grammars can easily get this information from the grammar. Therefore, it is an exact document of the project and it supports the implementation.

3. Summary

After a short survey of the fundamentals of software engineering we have discussed some classifications of methods and tools. As a result, the combination of data driven programming and data encapsulation, usually classified as contrary concepts, was developed by using attribute grammars.

This method was demonstrated by a very little commercial data processing system. The advantages of the method presented can be summarized as follows.

1. Attribute grammars are a good document for design and implementation.
2. Modularization is supported.
3. Maintenance can be performed relative easily and locally.
4. Syntactic analysis of data is automated and the software engineer can concentrate upon the main principles of his system.
5. Grammars can already be tested at very early development phases and the completeness of the system can be checked.
6. Simulations can be performed without total implementation of all functions.
7. Developed projects are broken up into many parts in a natural manner, which can run in parallel.
8. Functions have not to be designed in the same manner. A system of existing modules can be composed by using this method.
9. The method supports the use and design of so called knowledge bases (e.g. as abstract data types).
10. Syntactic analysis algorithm in translator writing systems will be much more effective than most hand written algorithms.
11. Automatic error recovery methods can be used (e.g. [For 84 b]).

Of course, this method is not intended to be applied to all problems of software engineering. The application of data driven programming, however, is very well supported by a grammar. We think this method to be useful especially in the field of commercial data processing.

Only a short list of references can be given here. A more complete list with about 300 references related to the topic of software specification can be obtained from the authors.

WILHEM PICK UNIVERSITAET ROSTOCK
SEKTION INFORMATIK
DDR 2500 ROSTOCK
ALBERT EINSTEIN STRASSE 21

References

[Aba 82] ABAFFY, J., KRAFFT, W., XHELF: An Aid for Developing Computer Programs, Proc. of the Conference on System Theoretical Aspects in Computer Science, Hungary, 1982.

[Bac 78] BACKUS, J., Can Programming be Liberated from the von Neumann Style? Comm. of the ACM 21 (8) 1978.

[Bjø 78] BJØRNER, D., JONES, C. B., The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science, Vol. 61, 1978.

[Cle 80] CLEAVELAND, J. C., Mathematical Specification, SIGPLAN Notices, 15 (12) 1980.

[Clo 81] CLOCKSIN, W. F., MELLISH, C. S., Programming in PROLOG, Springer Verlag, 1981.

[Col 81] COLEMAN, D., HUGHES, W., POWELL, M. S., A Method for the Syntax Directed Design of Multiprograms, IEEE Trans. on Software Eng., 7 (2) 1981.

[Des 83] DESPEYROUX, J., An Algebraic Specification of a PASCAL Complier, SIGPLAN Notices, 18 (12) 1983.

[Fib 84] FIBY, R., MOLNAR, S., WEIGL, I., Is the Idealised Logic Programming Feasible? Proc. IMYCS 84, Smolenice, CSSR, 1984.

[For 84a] FORBRIG, P., Attributierte Grammatiken und Softwarespezifikation, Seminar attr. Gr., Rostock 84.

[For 84b] FORBRIG, P., A New Error Recovery Method for Optimized LR Parsers, Proc. IMYCS 84, Smolenice.

[For 85] FORBRIG, P., Kombination der datengesteuerten Programmierung nach Jackson mit der Methode der Datenabstraktion nach Parnas, Rostock, Rep. 7/85.

[Gog 74] GOGUEN, J. A., THATCHER, J. W., Initial Algebra Semantics, IEEE Symp. on Switching and Autom. 74.

[Gut 75] GUTTAG, J. V., The Specifications and Application to Programming of Abstract Data Types, University of Toronto, Report CSRG-59, 1975.

[Gut 78] GUTTAG, J. V., HORNING, J. J., The Algebraic Specification of Abstract Data Types, Acta Informatica 10 (1) 1978.

[Gyi 83] GYIMÓTHY, T., SIMON, E., MAKAY, A., An Implementation of HLP, Acta Cybernetica, 3 (6) 1983.

[Heh 83] HEHNER, E. C. R., SILVERBERG, B. A., Programming with Grammars: An Exercise in Methodology-Directed language Design The Computer J. 26 (3) 1983.

[Hug 79] HUGHES, J. W., A Formalization and Explication of the Michael Jackson Method of Program Design, Software Practice & Experience, 9 (2) 1979.

[Hoa 69] HOARE, C. A. R., An Axiomatic Basis for Computer Programming, Comm. of the ACM, 12 (10) 1969.

[Jac 75] JACKSON, M., Principles of Program Design, Academic Press, 1975.

[Kat 81] KATAYAMA, T., HFP: A Hierarchical and Functional Programming Based on Attribute Grammars, Proc. 5th Int. Conf. on Software Engineering, 1981.

[Knu 82] KNUTH, E., NEUHOLD, E. J., Specification and Design of Software Systems, Proc. of the Conference of Operating Systems, Hungary, 1982.

[Kow 74] KOWALSKI, R. A., Predicate Logic as a Programming Language, Information Processing 74, North Holland, 1974.

[Lae 84] LAEMMEL, U., Specification of Dialogue Systems Using Attributed Grammars, Proc. IMYCS 84.

[Lin 83] LINDSEY, C. H., ELSA — An Extensible Programming System, IFIP — TC 2, Dresden, 1983.

[Log 83] LOGRIPPO, L., SKUCE, D. R., File Structures, Program Structures and Attribute Grammars, IEEE Trans. on Software Engineering, 9 (3) 1983.

[Loy 84] LOYD, J. W., Foundations of Logic Programming, Springer Verlag, Heidelberg—New York—Tokio 1984.

[Mad 80] MADSEN, O. L., On Defining Semantics by Means of Extended Attribute Grammars, Lecture Notes in Computer Science, Vol. 94, 1980, p. 259—300.

[Mal 80] MALLGREN, W. R., Formal Specification of Graphical Data Types, University of Washington, Technical Reprt No. 80—04—04, 1980.

[Mał 82] MAŁUSZYNSKI, J., NILSSON, J. F. A Version of PROLOG Based on the Notion of Two-Level Grammars, International PROLOG Workshop, Linkoeping, 1982.

[Noo 75] NOONAN, R. E., Structured Programming and Formal Specification, IEEE Trans. on S. Eng., 1 (4) 1975.

[Par 72] PARNAS, D. L., On Criteria to be Used in Decomposing Systems into Modules, Comm. ACM, 15 (12) 1972.

[Rec 84] RECHENBERG, P., Attributierte Grammatiken als Methode der Softwaretechnik, El. Rechenanl., 26 (3) 84.

[Rie 83] RIEDEWALD, G., MALUSZYNSKI, J., DEMBINSKI, P., Formale Beschreibung von Programmiersprachen, Akademie Verlag Berlin, 1983. also Oldenbourg Verlag, Wien—Muenchen, 1983.

[Rie 85] RIEDEWALD, G., Ein Modell fuer Programmiersprachen und Compiler auf der Basis universeller Algebren, Elektronischen Informationsverarbeitung und Kybernetik, 21 (3) 1985.

[Sco 71] SCOTT, D., STRACHEY, C., Towards a Mathematical Semantics for Computer Languages, Proc. of the Symp. on Computer and Automata 1971.

[Sto 77] STOY, J. E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press 1977.

[Sze 77] SZEREDI, P., PROLOG: A Very High Level Language Based on Predicate Logic, Proc. 2nd Hungarian Computer Science Conference, 1977.

[Wat 79] WATT, D. A., MADSEN, O. L., Extended Attribute Grammars, Aarhus Univ., Rep. DAIMI PB-105, 1979.

[Watt 83] WATT, D. A., MADSEN, O. L., Extended Attribute Grammars, The Computer Journal, 26 (2) 1983.

[Wil 82] WILSON, W. W., Beyond PROLOG: Software Specification by Grammars, SIGPLAN Notices, 17 (9) 1982.

(Received Dec. 18, 1985)