PUC-RJ - Departamento de Informática

Series : Monografias em Ciência da Computação

Nº 4 / 88 June 1988

Series Editor : Paulo A. S. Veloso

A PROBLEM - THEORETIC ANALYSIS AND (META -) MODEL OF THE SOFTWARE DEVELOPMENT PROCESS

Paulo A. S. Veloso +

Armando M. Haeberer *

+ Pontifícia Universidade Católica, Dept. Informática
22453 Rio de Janeiro RJ ; BRAZIL

* ESLAI : Escuela Superior Latinoamericana de Informática
PO Box 3193 , 1000 Buenos Aires ; ARGENTINA

Research partly sponsored by the ETHOS project of the Argentinian-Brazilian Program of Research and Advanced Studies in Computer Science and by FINEP.

In charge of publications

Rosane T. L. Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC/RJ - Depto. de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

ABSTRACT

The software development process is analyzed from a problem-theoretic viewpoint and a precise (meta-)model is proposed. This provides a uniform conceptual structure for understanding this process by clarifying its semantics and indicates some important requirements on formalisms purporting to represent it.

The software development process goes from informal descriptions to efficient programs, which are formal and executable, via precise specifications. The first phase, of formalization, is similar to theory construction in natural science and is validated as such. The process of program construction obtains an efficient program from a precise specification and has as central step the construction of solutions for problems. Auxiliary steps, capturing divide-and-conquer and reduction ideas, consist of specification translations, with refinements and decompositions, and program translations, with recombinations and extensions.

This meta-model provides a formalization for a large portion of software development process, leaving another portion as heuristics, which appears to be an essential ingredient in any widely applicable method. This analysis also sugests the non-existence of a single canonical step for the entire process as well as obstacles to its complete formalization or automation.

Key words : software development, program construction, problem solving, problem theory, formal model, formal specifications, program transformations, stepwise refinement, decomposition, abstraction, correctness, validation, testing, formal systems.

RESUMO

Analisa-se o processo de desenvolvimento de programas de um ponto de vista de teoria de problemas, propondo-se um (meta-)modelo preciso. Isto fornece uma estrutura conceitual uniforme para o entendimento deste processo, através da elucidação de sua semântica, além de indicar importantes restrições sobre formalismos que pretendam representá-lo.

O processo de desenvolvimento de programas vai de descrições informais a programas eficientes, que são formais e executáveis, através de especificações precisas. A primeira fase, de formalização, é similar à construção de teorias em ciência natural, sendo como tal validada. O processo de construção de programas obtém um programa eficiente a partir de uma especificação precisa, tendo como passo central a construção de soluções para problemas. Passos auxiliares, envolvendo idéias de divisão-e-conquista e redução, consistem de traduções de especificações, com refinamentos e decomposições, e de programas, com recombinações e extensões.

Este meta-modelo fornece uma formalização para uma boa parte do processo de desenvolvimento de programas, deixando outra parte como heurística, o que parece ser um ingrediente essencial em qualquer método de grande aplicabilidade. Esta análise também sugere a não-existência de um único passo canônico para todo o processo, além de obstáculos a sua completa formalização ou automação.

Palavras chave : desenvolvimento de programas, construção de programas, resolução de problemas, teoria de problemas, modelo formal, especificações formais, transformações em programas, refinamentos sucessivos, decomposição, abstração, corretude, validação, teste, sistemas formais.

ACKNOWLEDGEMENTS

Research reported herein has been partly sponsored by the ETHOS project of the Argentinian-Brazilian Program of Research and Advanced Studies in Computer Science and by the Brazilian agency FINEP. Helpful conversations with Gabriel Baum, Thomas S. E. Maibaum, Alejandro Rios, Władysław M. Turski, and Sheila R. M. Veloso are gratefully acknowledged.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This paper analyzes the software development process from a problem-theoretic viewpoint and proposes a meta-model for it. The need for a formal model of the software development process has been felt for some time. Such a model is needed in order to understand the deep nature of the process as well as for the design and implementation of environments to support software development [TM]. Such a need has been felt in the ETHOS project.

ETHOS is an acronym (in Portuguese and Spanish) for Software-Oriented Heuristic Work-Station, which is the kernel of a joint Brazilian-Argentinian research/development project. The product of this project is to be a meta-environment for knowledge-based software development. The environment designer shall provide this meta-environment with information concerning a software development method and an application domain. It will then generate an environment where a software engineer can develop a program for a particular application in this domain by using the given method [TM].

The above idea of environment generator, to be instantiated to particular environments, raises some fundamental questions. For instance, how are these methodologies to be described; are there perhaps primitives for such task ? Also, what is the software development model underlying this process ? Here, we propose some answers to these questions, arrived at by means of a problem-theoretic analysis of the software process.

The structure of this paper is as follows. In the next

section we examine some ideas on software development, which motivate a first, simplified view of our meta-model. Section 3 presents some ideas concerning problems and solutions, as well as some operations on problems and corresponding ones on solutions. Section 4 uses these problem-theoretic concepts to clarify linguistic transformations and outlines a problem-theoretic view of divide-and-conquer in order to explain decomposition and recombination with formal systems, these ideas being then put together to provide a more detailed view of the meta-model for program construction. Then, sections 5 deals with the the entire software development process, emphasizing the similarity of the requirement formalization phase with the hypothetico-deductive model of natural science, especially in what concerns program testing and specification validation. Finally, section 6 presents some concluding remarks on this "rational reconstruction" of the programming process. In a first reading, the reader may choose to postpone some problem-theoretic details, proceeding to items 4.4 and 5.4 immediately after section 3.1.

2. THE SIMPLIFIED " INVERTED U " META-MODEL

In this section we examine some ideas on the software development process and on program construction by means of transformations, which motivate a first, simplified view of our " inverted U " meta-model. Item 2.1 sketches a model of the software process [LB,LST] and a method for program construction [BW,BP], whereas item 2.2 outlines our simplified meta-model and its phases and describes the transformations it involves.

2.1 Preliminaries

Lehman's PW model [LB] views the software development process as an inverted V consisting of abstraction and reification phases. The former starts from the application concept, a first verbalization of the problem, and obtains a specification by abstracting away from some irrelevant details. The latter starts from the specification and obtains a program by adding some details concerning the particular system where it is to be implemented. Thus, both the application concept and the final program can be viewed as representational models, in distinct languages, of the specification [Tk].

As noted by Lehman, both abstraction and reification are complex enough to be divided into subphases involving decompositions and recombinations. Also, as emphasized by Turski and Maibaum [MT], the formal parts of the development process can be regarded as iterations of a canonical step, whose kernel amounts to linguistic transformations of representational models. It is suggested that one can exercise creativity either in the

design of the target language or in the transformation itself.

The CIP method of program construction by transformations [BN] provides a good illustration for some ideas. This method starts from a specification of the problem in first-order predicate logic, which gives rise to a first declarative version. The latter is successively transformed into more refined versions, until a functional, applicative version can be obtained. By means of further transformations other applicative versions are obtained and eventually transformed into iterative versions, which can be still further transformed. All such transformations are effected by means of rules, previously proven correct, which guarantees the correctness of the final program.

```
*
oper bin_log ( n : int ) ----> int
   pre  n > 0
      that k : int such that ( 2↑k < n & n ≤ 2↑(k+1) )
end_oper  {  original problem specification  }

oper bin_log ( n : int ) ----> int
   pre  n > 0
      that k : int such that ( ⌊ n/2 ⌋ < 2↑k ≤ n )
end_oper  {  non-algorithmic formulation  }
```

Fig. 2.1 : Two declarative versions of a problem specification : integer part of base-two logarithm of a positive natural.

As an illustration of this method of program construction by

transformations, consider the problem of computing the integer part of the base-two logarithm of a positive natural [BP]. Some of the versions obtained in constructing a program to solve this problem are shown in figs. 2.1 and 2.2.

```
*
oper bin_log ( n : int ) ----> int
   pre  n > 0
     if n = 1
          then 0
          else bin_log ( n div 2 ) + 1
     end_if
end_oper  {  applicative version  }
```

```
oper bin_log ( n : int ) ----> int
   { pre  n > 0 }
     var i , j : int ;
       i := n ; j := 0 ;
       while i > 1 do
         i := i div 2 ; j := j+1
       end_while ;
       return j
end_oper  {  iterative version  }
```

Fig. 2.2 : Two algorithmic formulations of a program : for the integer part of base-two logarithm of a positive natural.

Thus, the above method views program construction as a series of linguistic transformations, and one may regard the

application of a rule from the catalogue as its canonical step. Two points, however, should be stressed. First, this method has a formal part, embodied in the catalogue of rules, formally proven correct, and a heuristic part consisting of the selection of the rules to be applied at each step. It appears that most, virtually all, methods have these two parts. Second, the various declarative versions, illustrated in fig. 2.1, are basically problem specifications, whereas the other ones, illustrated in fig. 2.2, are algorithmic descriptions of a solution for this problem; as such they are fundamentally different in nature. We contend that the passage from a declarative problem specification, as in fig. 2.1, to an algorithmic description of a solution, as in fig. 2.2, is the crucial step in program development, and in problem solving, one that is not explicated merely by linguistic transformations. The latter, together with decompositions, may be regarded as preparatory steps for the central one of solution construction.

2.2 A First View of the Meta-model

One can say that there are two major kinds of languages, informal and precise ones, the latter being further subdivided into declarative and algorithmic ones. Thus, the process of software development may be viewed as starting from an informal description of a problem, which is made first precise and then algorithmically executable. The first phase, the formalization of an informal verbalization, involves activities that can be epistemologically described as "interpretation" and "elucidation". As such, it cannot be proven correct; it can only

be validated by processes akin both to program testing and to the traditional hypothetico-deductive method in natural science [He].

Thus, software development may be regarded as consisting of two major processes, namely formalization and program construction. The former goes from an from an informal application concept to a precise specification, and is similar to the process of theory formation in the natural sciences. The latter goes from a precise specification to an (efficiently) executable program, and can be thought of as a mathematical activity since it deals with precise, formal objects [BP]. This view of the software development process is depicted in fig. 2.3.

*

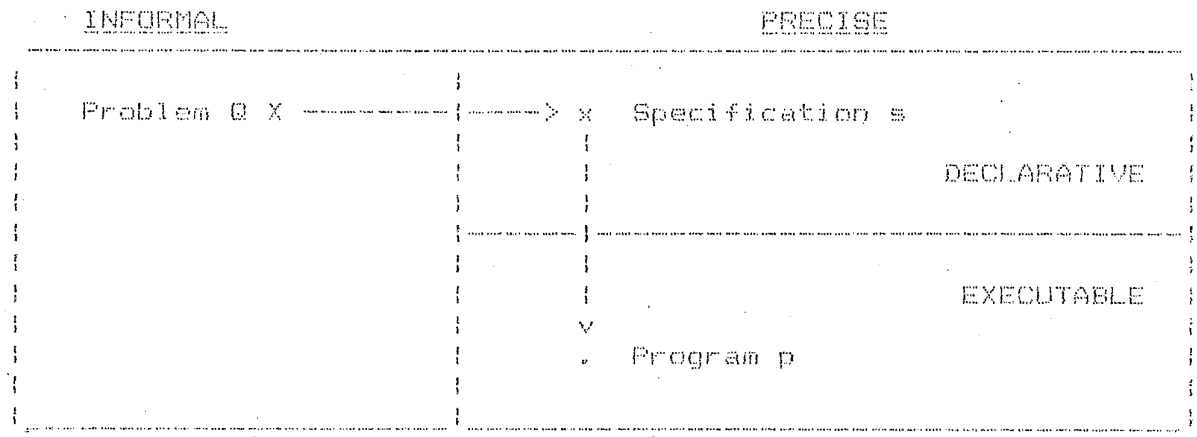


Fig. 2.3 : Software development
as formalization followed by program construction.

The (simplified) " inverted U " meta-model views program construction as consisting of three kinds of transformations :

α : specification refinement with decomposition;

δ : construction of feasible algorithmic solutions;

β : program extension with recombination.

These transformations are formal and verification lemmas have been established that ensure their correctness (see section 4). They can also be validated by means of tests (see section 5).

*

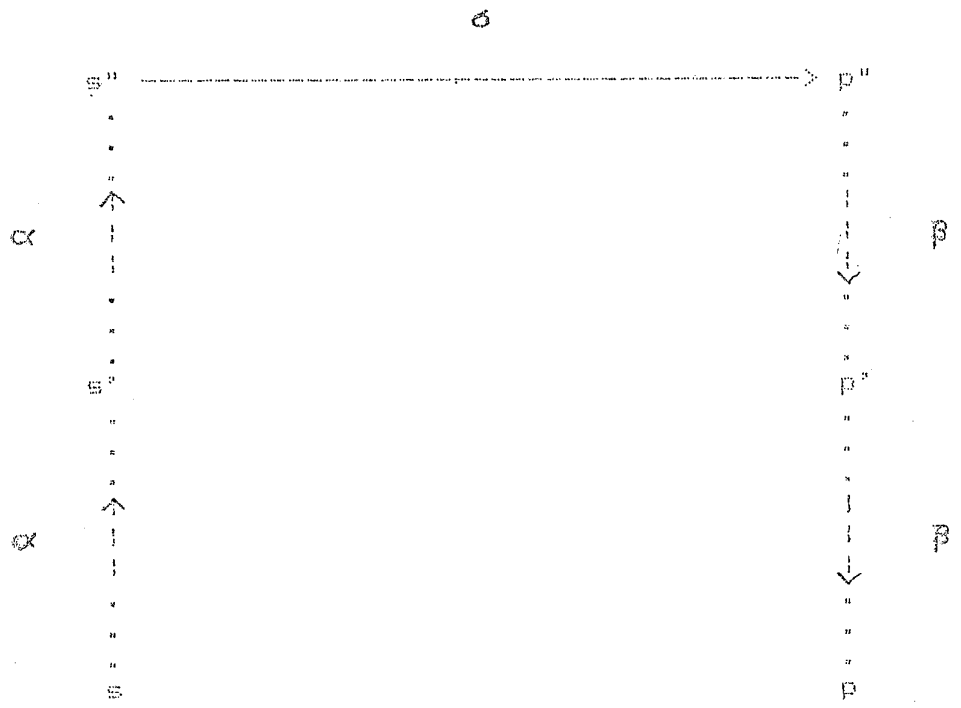


Fig. 2.4 : Inverted U meta-model for program construction (simplified version).

The whole process of program construction is modelled as starting with a first formal specification s at the lower left of the inverted U. It then climbs up the left leg with a series of specification transformations s ... -- α --> s' -- α --> ... s" involving refinement and decomposition. Then, it traverses the top curve by constructing solutions for the component (sub)problems s" -- σ --> p" . Finally, it goes down the right leg via a series of program transformations involving extension

and recombination p" ... — β --> p' — β --> ... p . This simplified inverted U meta-model for program construction is illustrated in fig. 2.4.

The process of program construction obtains a program p to solve a problem specified by s . Thus, it can be described by the regular-like expression $\alpha^* . \sigma . \beta^*$. Now, the process of program development obtains a program p to solve a problem Q . So, if we denote the formalization phase by ϕ , then this process can be similarly described by $\phi . \alpha^* . \sigma . \beta^*$. In the sequel we shall elaborate on these points. This simplified inverted U meta-model for the software development process is illustrated in fig. 2.5.

*

Fig. 2.5 : Inverted U meta-model for the software development process (simplified version).

3. BASIC PROBLEM-THEORETIC CONCEPTS

Program development aims at solving problems. This section presents some basic problem-theoretic concepts, to be used throughout the rest of the paper. In item 3.1 we define the fundamental concepts of problem and solution, by rendering precise some intuitive ideas suggested by Polya [Po] and formalize our δ transformation or solution construction. Then, item 3.2 outlines some operations on problems, accompanied by corresponding ones on solutions, paving the way for our analysis of the divide-and-conquer strategy for problem solving.

3.1 Problems and Solutions

Some intuitive ideas of Polya's [Po] can be captured by a precise definition of problem as a mathematical structure consisting of data, results, requirement, and instances of interest [V80]. A solution for a problem is basically a function assigning results to data so as to satisfy the requirement for the instances of interest. This provides a framework for study of problem-solving methods like reduction and divide-and-conquer [VV]. It will be seen that this formulation, suggested by the use of liberal specifications for abstract data types in program development [VPM], bears some resemblance to ideas used in program verification [Ma] and in recursion theory [Ro].

A problem [V84] is a two-sorted structure Q consisting of nonempty domains D and R, of input data and output results, respectively, together with a binary relation q from D to R, the requirement, and a subset I of D, the set of instances of

interest. A restricted function is a partial function F from D to R together with a restriction J contained in the domain of F . We shall leave 'restricted' implicit and use simply F to denote such functions whenever convenient and safe. A solution for problem Q is a (restricted) function F from D to R whose restriction J includes I and such that for every d ∈ I < d , F(d) > ∈ q . We use F ≺ Q to denote that F solves Q , i. e. I ⊆ J and F | I ⊆ q .

Call problem Q viable iff for every d in I there exists r in R such that < d , r > ∈ q , which can be expressed by the formula (∀ d : D) [d ∈ I ——> (∃ r : R) q(d , r)] .
Then, it is easily seen that the solutions for Q are basically the Skolem functions [En,Ma,Sh] of this formula.

We often describe solutions by programs expressed in an algorithmic language A with semantics ν , in that ν assigns to each program p in A the (restricted) function ν[p] computed by p (its restriction coming from the expressed precondition of p). Similarly, problems will be described in a declarative language N with semantic function μ assigning to each problem specification s in N the problem μ[s] specified by s .

Now, we can precisely describe the σ transformation of solution (Skolem function) construction : we say that program p is a σ transformation of problem specification s (which we denote by s —— σ ——> p) iff the function computed by p is a solution for the problem Q specified by s . Thus, the correctness criterion for s —— σ ——> p is ν[p] ≺ μ[s] . A pictorial illustration for this correctness criterion is provided in fig.

3.1.

*

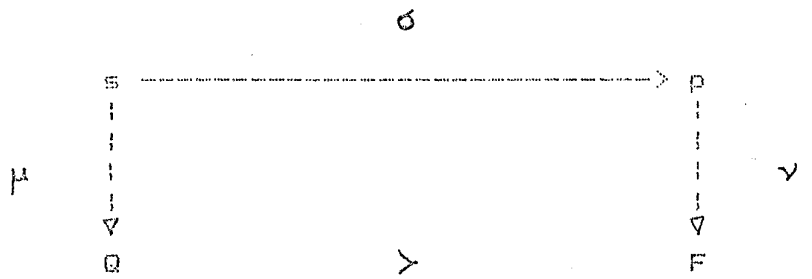


Fig. 3.1 : Solution construction transformation σ :
correctness criterion

We can now clarify why the left and right legs of the inverted U are different in nature. A problem specification describes the problem requirement as a binary relation, a set of ordered pairs of data and results (see fig. 2.1). On the other hand, a program computes a function, it describes an algorithm, as in fig. 2.2, that actually obtains a result from a data. This is where their difference lies, which makes solution construction the crucial step in software development, and indeed in problem solving.

3.2 Some Operations on Problems and Solutions

A fundamental strategy in program development, and indeed in problem solving in general, is the so called divide-and-conquer paradigm [HS,VV], where a problem is decomposed into smaller subproblems, whose solutions are recombined into a solution for the original problem. In our problem-theoretic framework, decomposition amounts to expressing the given problem as the

result of some operations on component problems, whereas recombination is effected by accompanying operations on solutions. We shall now illustrate some simple cases of this paradigm (for more details see [HBVJ), leaving the case of "abstraction" for the next section.

We call a problem Q the sum of problems Q' and Q'', denoted by $Q = Q' \dagger Q''$, iff $D = D' \cup D''$, $R = R' \cup R''$, $q = q' \cup q''$, and $I = I' \cup I''$. Now, we would like to combine a solution F' for Q' and a solution F'' for Q'' into a solution $F' \dagger F''$ for $Q \dagger Q''$. This function $F' \dagger F''$ is basically the union of F' and F'' with some choice, given by a subset of the intersection of D' and D'', when both happen to be defined. We then have the sum lemma : if $F' \prec Q'$ and $F'' \prec Q''$ then $F' \dagger F'' \prec Q' \dagger Q''$, which is illustrated in fig. 3.2.

*

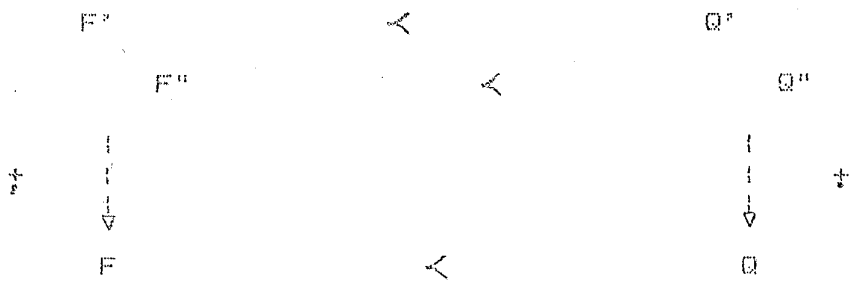


Fig. 3.2 : Sum lemma.

We call a problem Q the product of problems Q' and Q'', denoted by $Q = Q' \ddagger Q''$, iff $D = D'$, $R = R''$, $q = q' \circ q''$ (the relative product), and $I = I'$. Now, define function $F' \ddagger F''$ as the composite so that $F' \ddagger F''(d) = F''(F'(d))$. Thus, we have the following product lemma : if $R' \subseteq D''$ and $I'q' \subseteq I''$,

then F' ; F" ≺ Q' ; Q" whenever F' ≺ Q' and F" ≺ Q" (here I'q' is the image of I' under q'). This lemma is illustrated in fig. 3.3.

*

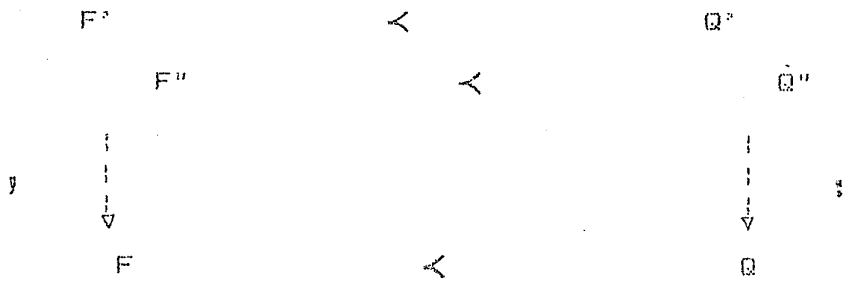


Fig. 3.3 : Product lemma.

In general, one may have other operations on problems, most of them accompanied by corresponding ones on solutions. The goal is having a decomposition theorem like : if Q is expressed as a "polynomial" T(Q_1 ,..., Q_n) satisfying appropriate conditions, then, given solutions F_j for Q_j (j = 1 ,..., n) , the corresponding "polynomial" t(F_1 ,..., F_n) expresses a solution for Q , as illustrated in fig. 3.4.

*

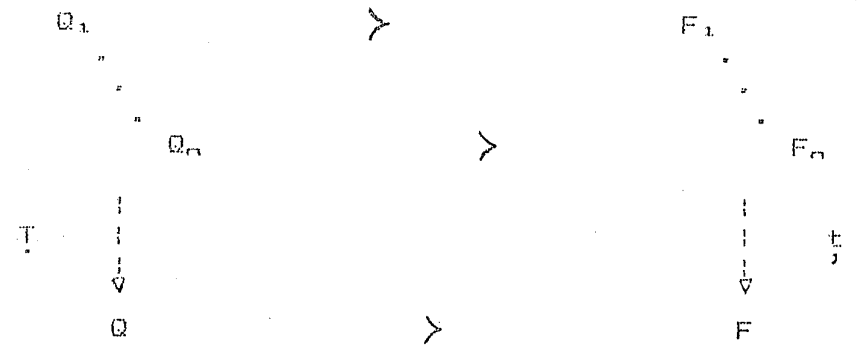


Fig. 3.4 : Decomposition goal.

The above correspondence between "polynomial" expressions for

problems and solutions is the basic idea of Jackson's method for program development [Ja] and will be clarified in the sequel, together with the, more general, divide-and-conquer strategy for problem solving.

4. PROGRAM CONSTRUCTION : ANALYSIS AND META-MODEL

In this section we use problem-theoretic concepts to analyze and explicate the various phases of the process of program construction. In item 4.1 we clarify linguistic transformations and in item 4.2 we outline a problem-theoretic view of divide-and-conquer, which is used in item 4.3 to explain decomposition and recombination with formal systems. Those ideas are put together in item 4.4 to provide a more detailed view of our meta-model for program construction.

4.1 Linguistic Transformations

As mentioned before, we have two kinds of linguistic transformations, namely on programs and on problem specifications. At first sight, such transformations would be expected to preserve meaning. But, it often happens that a transformed program is more general than the original one, which is perfectly acceptable. A similar phenomenon occurs with specification transformations.

Consider programs p and p' in languages, respectively, A and A' , perhaps the same. We say that p' is an extension of p , denoted by p --- ξ ---> p' , iff the function computed by p' is an extension of the function computed by p , in that the former may have a larger restriction but they agree on the smaller one. Thus, the correctness criterion for the extension transformation p --- ξ ---> p' is $J \subseteq J'$ and $F' \mid J = F$ where $\nu[\ p\ ]$ consists of F and J and similarly for $\nu'[\ p'\ ]$. This correctness criterion for c transformations is illustrated in fig. 4.1.

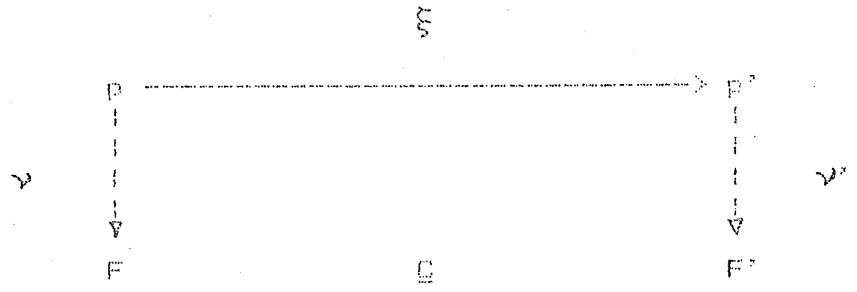


Fig. 4.1 : Program extension transformation ξ : correctness criterion

Consider problem specifications s and s' in languages, respectively, N and N', perhaps the same. We say that s' is a refinement of s , denoted by s --- η ---> s' iff every solution for the problem specified by s' is a solution for the problem specified by s . Thus, one can see that a correctness criterion for the transformation s --- η ---> s' is $\mu'[\,s'\,] \supseteq \mu[\,s\,]$. Here by $Q' \sqsubseteq Q$ we mean that $D = D'$, $I \subseteq I'$, $R' \subseteq R$, and $q' \mid I \subseteq q$; for then, $F \prec Q$ whenever $F \prec Q'$. Fig. 4.2 illustrates this correctness criterion for n transformations.

*

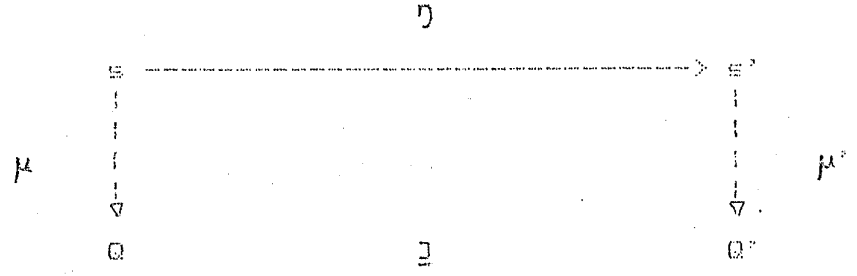


Fig. 4.2 : Specification refinement transformation η : correctness criterion

We mentioned Turski's observation that there is some room for creativity in the linguistic transformations. The above correctness criteria make explicit a constraint on this creativity : both representational models are to be equivalent, up to some small extension or refinement.

4.2 Abstraction and Parameterization

The idea of corresponding operations on problems and on solutions is similar to the analogy between structuring constructs for data and for control [Ho], exploited in Jackson's method [Ja]. So, it approximates the idea of stepwise refinement. Other powerful concepts employed in program development are abstraction and parameterization [Bo], which are best dealt with in linguistic terms.

Consider a specification $s(X_1 , \ldots , X_n)$ with variables ranging over problems. It does not by itself specify a problem, but if we instantiate each problem variable X_j to a variable-free specification s_j describing a problem $Q_j = \mu[s_j]$, then we will have $s(s_1 , \ldots , s_n)$ specifying a problem $Q(Q_1 , \ldots , Q_n)$. So, we may regard this specification as parameterized. Thus, solutions for such a specification should be parameterized as well. So, consider a parameterized program $p(x_1 , \ldots , x_n)$ with variables ranging over programs. Again, if we instantiate each program variable x_j to a variable-free program p_j computing the function $F_j = \nu[p_j]$, then $p(p_1 , \ldots , p_n)$ will compute a function $F(F_1 , \ldots , F_n)$. We can generalize our previous δ transformation to solving with parameters, which we shall denote

by $s(X_1 , \ldots , X_n) \text{ -- } \sigma \text{ --> } p(x_1 , \ldots , x_n)$, by means of the natural correctness criterion that whenever $s_j \text{ -- } \sigma \text{ --> } p_j$, for $j = 1 , \ldots , n$, then $s(s_1 , \ldots , s_n) \text{ -- } \sigma \text{ --> } p(p_1 , \ldots , p_n)$. This correctness criterion for the σ transformation of solving with parameters is illustrated in fig. 4.3.

*

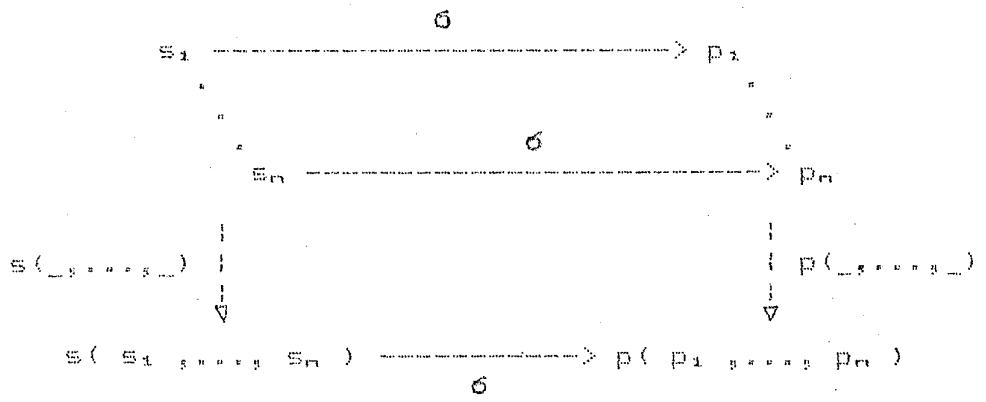


Fig. 4.3 : Solving with parameters : correctness criterion for $s(X_1 , \ldots , X_n) \text{ -- } \sigma \text{ --> } p(x_1 , \ldots , x_n)$.

Notice that we may view the program variables of a parameterized program as invocations to procedures still to be defined. Thus, it resembles a program on an abstract data type [LZ]. Similarly, expressing a given problem as $Q(Q_1 , \ldots , Q_n)$ involves a process of abstraction. The correctness criteria and theorems tell when and how one can obtain a solution for the original problem from solutions for the component ones. This is the formalized part of a program construction method. They do not tell how to express the given problem in terms of component ones. Such design decisions are guided by the heuristic part of the method. In the sequel we shall elaborate on this similarity between solving with parameters and programming with abstract

data types.

4.3 Decomposition and Recombination

In the declarative language N we have operations on specifications, like $\dotplus$ and $,$, to be interpreted as operations on problems. On the solution side, the algorithmic language A should have operations on programs, like $\dotplus$ and $;$, corresponding to those on specifications. But, these are not the only means available to decompose specifications and to recombine programs.

First, let us consider the declarative language N . We say that problem specification s is transformed by decomposition into a parameterized specification $s_0(X_1 , \ldots , X_n)$ together with n specifications $s_1 , \ldots , s_n$ iff s is the result $s_0(s_1 , \ldots , s_n)$ of substituting $s_1 , \ldots , s_n$ for $X_1 , \ldots , X_n$, respectively, in $s_0(X_1 , \ldots , X_n)$. For this δ transformation, which will be denoted by $s \text{ --- } \delta \text{ --->} s_0(X_1 , \ldots , X_n) / s_1 , \ldots , s_n$, we have $\mu[s_0(s_1 , \ldots , s_n)] = \mu[s]$. It is illustrated in fig. 4.4.

*

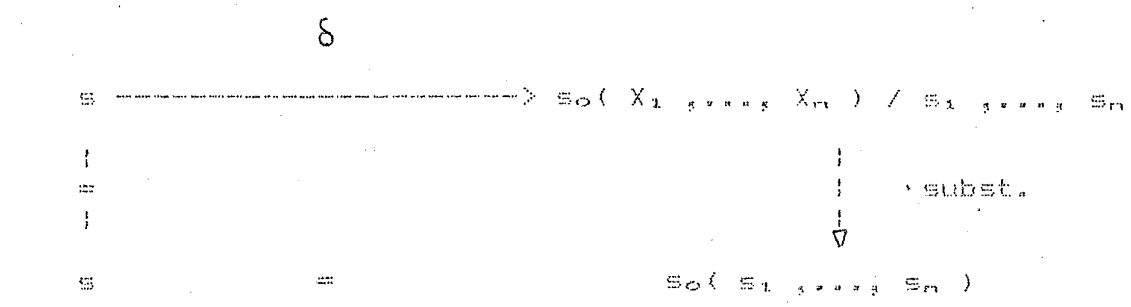


Fig. 4.4 : Decomposition transformation δ

Now, let us turn to the algorithmic language A . We say that program p is a recombination transformation of the parameterized

program $p_o(x_1 , \ldots , x_n)$ and the n programs $p_1 , \ldots , p_n$ iff p is the result $p_o(p_1 , \ldots , p_n)$ of substituting $p_1 , \ldots , p_n$ for, respectively, $x_1 , \ldots , x_n$ in $p_o(x_1 , \ldots , x_n)$. Denoting this ϱ transformation by $p \leftarrow\!\!- \varrho - p_o(x_1 , \ldots , x_n) / p_1 , \ldots , p_n$, we have $\nu[p_o(p_1 , \ldots , p_n)] = \nu[p]$. It is illustrated in fig. 4.5..

*

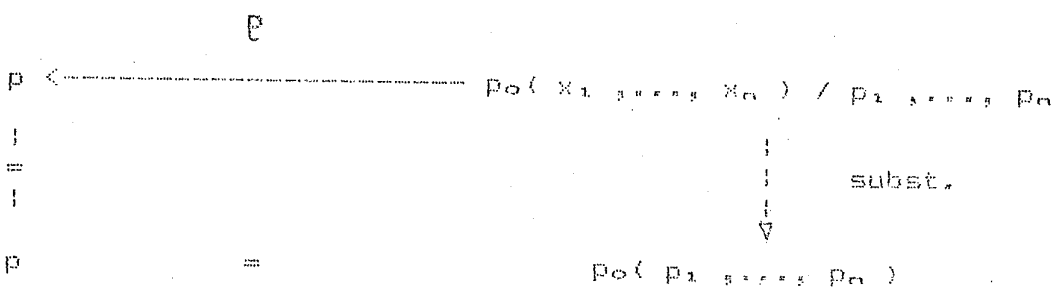


Fig. 4.5 : Recombination transformation ϱ .

Divide-and-conquer involves the idea of reducing a problem to others by decomposition [VV]. This is embodied in the correctness theorem for a decomposition together with a corresponding recombination : if $s - \delta \rightarrow s_o(X_1 , \ldots , X_n) / s_1 , \ldots , s_n$, then we have $s - \sigma \rightarrow p$, whenever $s_j - \sigma \rightarrow p_j$, for every $j = 1, \ldots , n$, $s_o(X_1 , \ldots , X_n) - \sigma \rightarrow p_o(x_1 , \ldots , x_n)$, and $p \leftarrow\!\!- \varrho - p_o(x_1 , \ldots , x_n) / p_1 , \ldots , p_n$.

The correctness theorem for a pair decomposition-recombination is illustrated in fig. 4.6. Notice the similarity with program development by means of abstract data types [VMS], especially the independence that exists between the " abstract program " $p_o(x_1 , \ldots , x_n)$ and the " implementation module " consisting of the "concrete" programs $p_1 , \ldots ,$ and p_n .

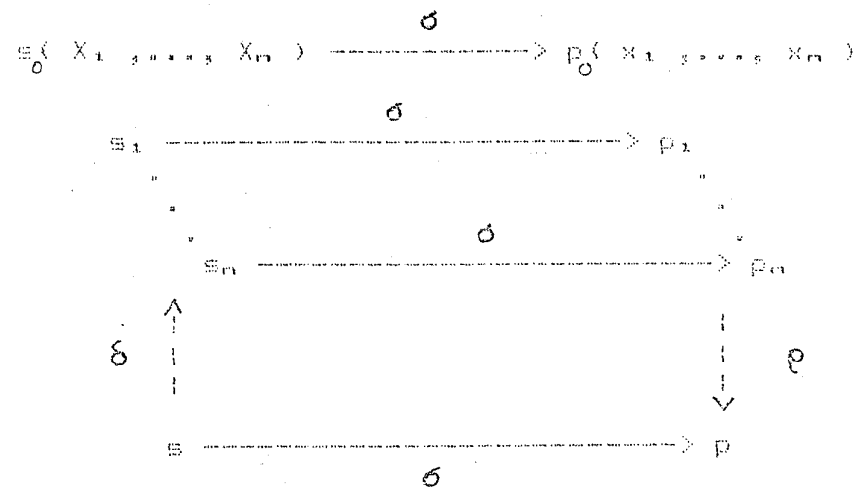


Fig. 4.6 : Decomposition theorem for corresponding δ and ρ transformations.

Now, let us return once more to the question of the amount of creativity involved in the various transformations. One sees that the above requirements for decomposition and recombination, even though permitting some room for design decisions, do impose severe limitations on both the languages and the transformations themselves.

These limitations can be explained by our problem-theoretic (two-sorted) formal system. In sort S we have variables ranging over problems and operation symbols like $+$ and $,$ and a closed S-term $s_0(s_1, \ldots, s_n)$ is a (structured) specification for a (decomposed) problem. Similarly, in sort P we have variables ranging over programs and operation symbols like $;$ and $,$ and a closed P-term $p_0(p_1, \ldots, p_n)$ is a (structured) presentation of a (recombined) program. The transformation of solution

construction is represented by the binary predicate symbol ó from sort S to sort P . Given an S-term $T(X_1 , \ldots , X_n)$ we can obtain its corresponding P-term $t(x_1 , \ldots , x_n)$ by replacing each problem operation symbol by the corresponding program operation symbol and distinct S-variables by distinct P-variables.

The above idea of corresponding operation symbols on specifications and on programs is similar to the one exploited in Jackson's method [Ja]. Jackson describes data by definition trees involving data structuring constructs and produces program schemas structured by the usual control structuring constructs [Ho]. The correspondence between these two sets of constructs stems from their being interpretations of the (deterministic) regular expressions [Hu]. We can now provide an alternative, more precise, view of this correspondence in our case.

We can envisage a (one-sorted) formal system L on a higher level. Among its operation symbols we have $+$, $\cdot$, and $/$, together with constant symbols like 0 and 1 ; its axioms include formulas like $v + v = v$, $v + 0 = v$, $v \cdot 1 = v$, $u / v = v$, and $(u + v) / w = w + v$, together with associativity of $+$ and $\cdot$ and commutativity of $+$. This formal system L has two natural interpretations in the logical sense [Sh]. The S-interpretation maps a term t of L to a problem specification t^S . This is done by interpreting $+$ and $\cdot$ as $\dagger$ and $;$, respectively. The interpretation of $/$ is the operation γ such that $s \; \gamma \; s'$ replaces the first specification variable in s by s' , so that the specification $s_0 \; \gamma \; (s_1 , \ldots , s_n)$ is the S-interpretation of the L-term $(\ldots (t_0 / t_1) \ldots / t_n)$ provided that $(t_j)^S = s_j$.

for $j = 0, \ldots, n$. The P-interpretation maps a term t of L to a program t^P in a similar manner. Thus, corresponding specification and program terms are the interpretations of a common term t of L and we require that $t^S \sigma t^P$ as shown in fig. 4.7. Abstraction may be viewed as giving some freedom beyond this correspondence, the price paid being the need to solve the "abstract" problem specified by the parameterized specification.

Our formal systems for problem specifications and programs solving them are illustrated in fig. 4.7. This idea can be used as a basis for a calculus of problems and solutions, similar in motivation to Sintzoff's design calculus [Si].

*

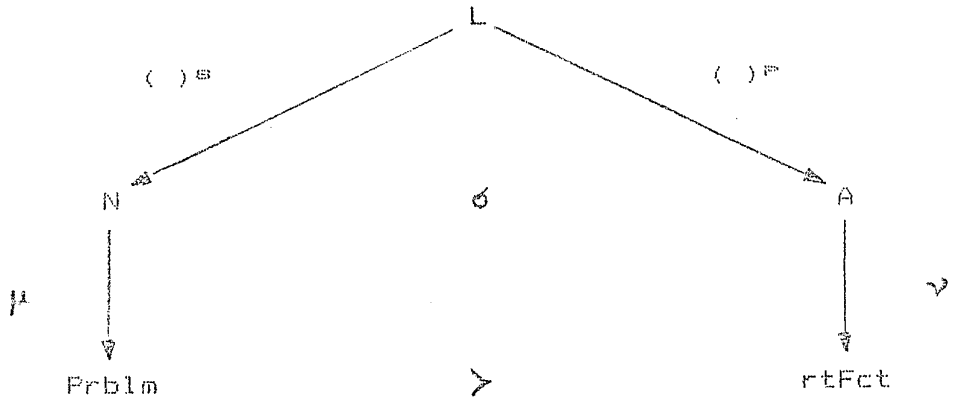


Fig. 4.7 : Formal systems for problems and solutions.

4.4 The Process of Program Construction

We are now in a position to give a more detailed description of the inverted U meta-model outlined in item 2.2. The process of program construction starts from a formal specification s of the problem Q to be solved. It then climbs up the left leg via

specification refinements and decompositions $s \ldots -\!\!- \eta \longrightarrow s' \ldots -\!\!- \delta \longrightarrow s_0 / s_1 , \ldots , s_n \ldots -\!\!- \eta \longrightarrow s'_0 / s'_1 , \ldots , s'_n$. Then the top curve is traversed by constructing solutions for the component problems $s'_j -\!\!- \sigma \longrightarrow p'_j$ for $j = 0 , \ldots , n$. Finally it goes down the right leg with a series of program recombinations and extensions $p'_0 / p'_1 , \ldots , p'_n -\!\!- \rho \longrightarrow p' \ldots -\!\!- \xi \longrightarrow \ldots p$. This is illustrated in fig. 4.8.

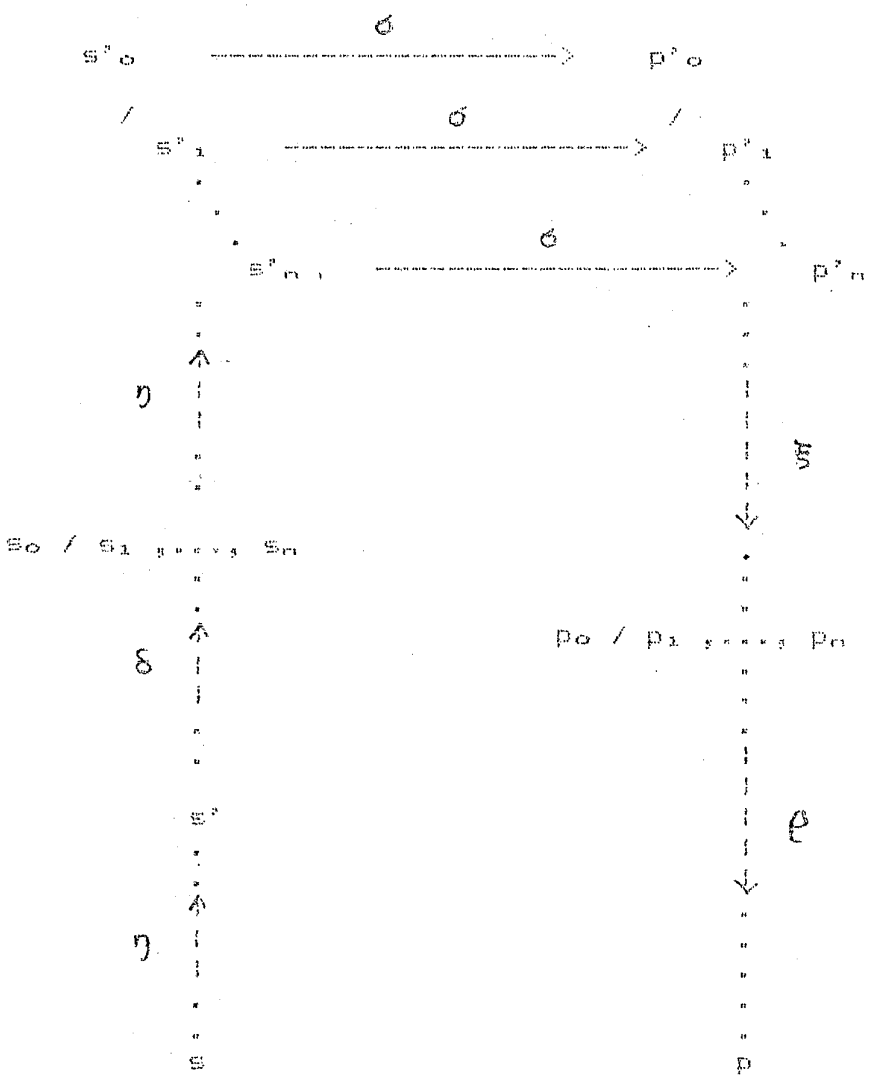


Fig. 4.8 : Inverted U meta-model for program construction.

Notice that along the left leg we have problem specifications whereas along the right one we have programs solving these problems, the transition along the top curve being effected by constructing solutions. Also, the decompositions δ and recombinations ρ correspond to each other and there is not necessarily any correspondence between the linguistic transformations on specifications and on programs, the η and ξ transformations. But, by including identical linguistic transformations, we can pair them off. So, let

$\frac{\eta}{\xi}$ denote a pair of refinement and extension and

$\frac{\delta}{\rho}$ denote a matching pair of decomposition and recombination.

Now, we can express the program construction process as

$\frac{s}{p} = \left[\frac{\eta}{\xi} + \frac{\delta}{\rho} \right]^* . \sigma$, where $\frac{s}{p}$ denotes the pair consisting of the original specification and the final program. The correctness criteria for these transformations guarantee that indeed $s \text{ -- } \sigma \text{ --> } p$.

We shall now revert to the simpler view of section 2. Let us call a series of specification refinements followed by a decomposition an α transformation, i. e. $\alpha = \eta^* . \delta$. Also, call a recombination followed by a series of program extensions a β transformation, i. e. $\beta = \rho . \xi^*$. We shall also simplify the notation by using s and p to denote tuples of specifications and programs, respectively. So, the process of program construction amounts to $s \; \alpha^* . \sigma . \beta^* \; p$.

We have two correctness lemmas, derived from the correctness criteria for the transformations :

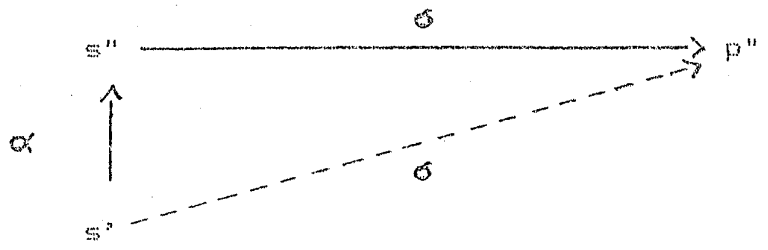
(α) if s' -- α --> s" and s" -- σ --> p" then s' -- σ --> p" ;

(β) if s" -- σ --> p" and p" -- β --> p' then s" -- σ --> p' .

The effect of these lemmmas is allowing the top σ-curve to slide down along the left and right legs of the inverted U , as illustrated in fig. 4.9.

*

α transformation :



β transformation :

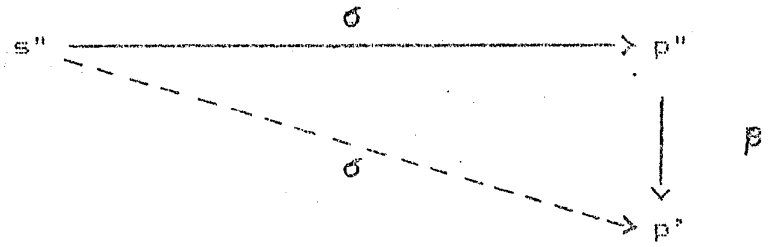


Fig. 4.9 : Correctness lemmas for α and β transformations.

By putting the above two lemmas together, we obtain the correctness theorem for the whole program construction process : if s α* . σ . β* p then s -- σ --> p , which is illustrated in fig. 4.10.

*

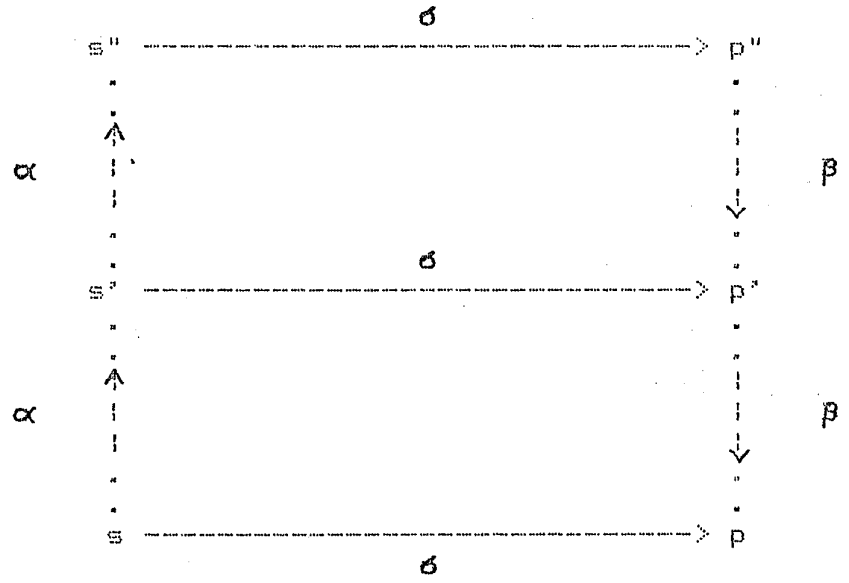


Fig. 4.10 : Correctness theorem for program construction process.

*

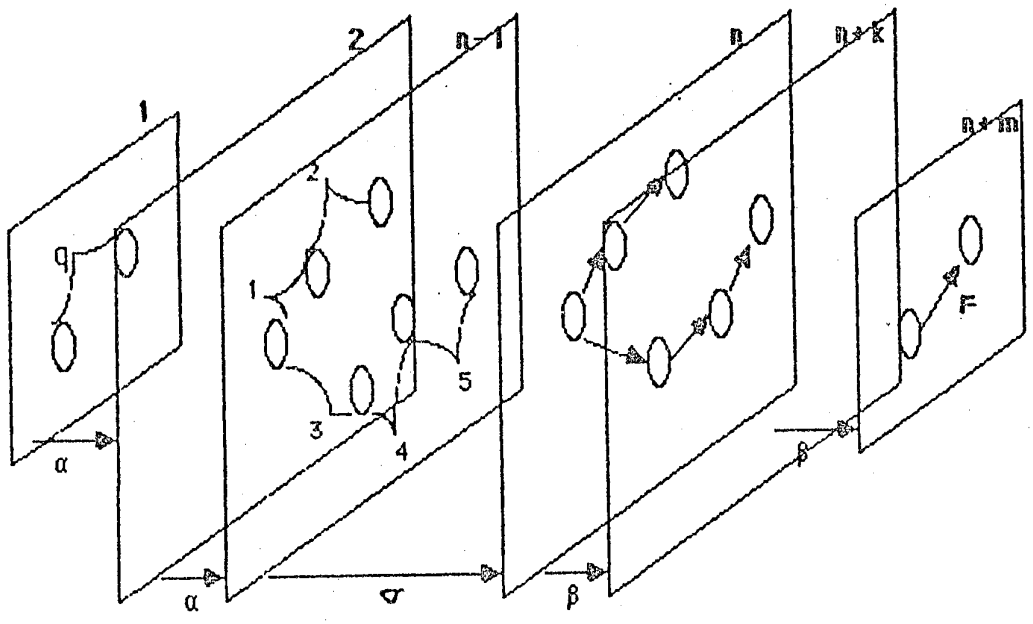


Fig. 4.11 : The process of program construction.

This view of the process of program construction is illustrated in fig. 4.11. In plane 1 we have the original problem specification. After a series of refinements, the problem is decomposed into (Q_1 ; Q_2) + (Q_3 ; Q_4 ; Q_5) as shown in plane (n-1) . By solving these component problems one obtains programs for them, shown in plane n . These programs are recombined, perhaps with extensions, into a program p , in plane (n+m) , describing a solution for the original problem.

*

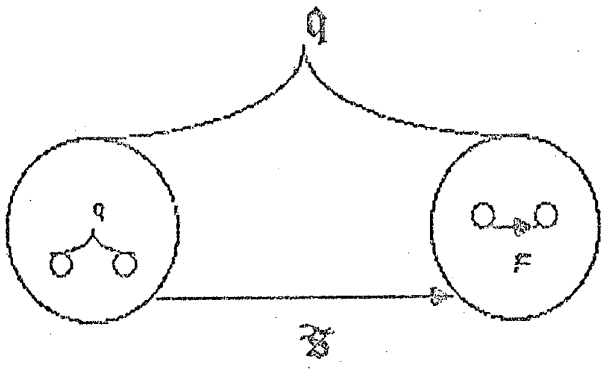


Fig. 4.12 : The (meta-)problem of program construction.

Note that we actually have two levels of problems, namely the application problem(s) to be solved and, on a higher level, the problem of program construction. In fig. 4.11 the former are represented in the vertical planes and the latter in the horizontal plane. The (meta-)problem of program construction is depicted in fig. 4.12. It has (application) problem specifications as data and (efficient) programs as results, its condition is that the program indeed solves the problem specified and its set of instances of interest consists of some specifications of problems, say, from a given application domain.

A solution for this (meta-)problem can be regarded as a *method for program construction* (for the given application domain).

A remark concerning our formulation of the process of program construction may be in order. For the sake of simplicity, we have been assuming that the passage from the left leg to the right one consists of constructing solutions for all the component problems. This is not strictly so : some component problems may very well be solved before others, as they may present different degrees of difficulty. Our meta-model can accommodate this aspect of "distributed" solution construction.

5. SOFTWARE DEVELOPMENT : ANALYSIS AND META-MODEL

The process of going from a problem to a program solving it is dealt with in this section, which is structured as follows. Items 5.1, 5.2 and 5.3 examine the requirement formalization phase, program testing and specification validation, respectively, emphasizing their similarity with the nomologico-deductive model of natural science. These ideas are then put together in item 5.4 to describe the whole software development process, including formalization, validation and testing.

5.1 The Requirement Formalization Phase

In view of the correctness theorem for program construction in the previous section, one can be sure that the program p arrived at does solve the problem described by the original specification s : $s -- \sigma --> p$. But, what one actually wanted was a program to solve the, perhaps informally stated, original problem Q , i. e. $\forall [p] \prec Q$. So, one still has a gap. This gap would be bridged if one could ensure that s is indeed a "correct" description of Q (see fig. 3.1).

The process of obtaining a formal specification for an informal problem may involve several intermediate steps. Once one has a precise description in a formal language, one may apply safe transformation rules; it is then that we view the process of program construction as starting. We shall argue that the process of going from an informal problem to a first formal specification bears some resemblance to the process of theory construction in natural sciences, and that specification validation is similar to

hypothesis testing.

Consider a problem Q and a specification s in the (formal) declarative language N . We say that s is a formalization for Q , denoted by Q -- Φ --> s , iff the problem specified by s is Q . So, the correctness criterion for this Φ transformation is simply μ[s] = Q , as illustrated in fig. 5.1. Since N is a formal language with precisely given semantics, we know that μ[s] is a mathematically precise object. The problem lies in Q , of which one may have only an informal description to start with.

*

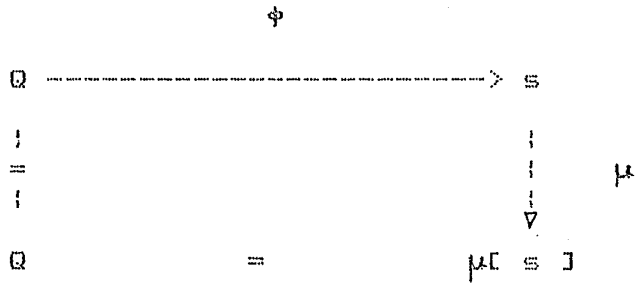


Fig. 5.1 : Formalization transformation Φ :
correctness criterion.

5.2 Program Testing

We shall now briefly describe the process of program testing in order to motivate our problem-theoretic view of specification validation and its comparison with hypothesis testing in natural science. Assume that one has a program p that alledgedly solves a problem Q . How does one goes about testing p ? One chooses some test data, runs the program on them and then checks whether the results obtained are as expected.

We may describe the above, somewhat simplified, view of program testing as follows (see fig. 5.2).

(H) Hypothesis under test : p solves Q (p is correct for Q) ;

(A) Auxiliary hypotheses : for $j = 1, \ldots, k$

$d_j \in I$ (d_j is a test data) ,

p with input d_j produces r_j (test runs) ;

(C) Observable consequence : for $j = 1, \ldots, k$

$\langle d_j , r_j \rangle \in q$ (r_j is an acceptable result for d_j) .

Now, we have H & A ---> C , by definition of correctness, and A , by construction. So, the experiment amounts to checking C . If C fails to hold, then we can conclude that so does H .

*

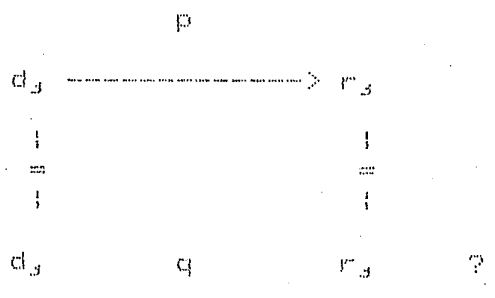


Fig. 5.2 : Program testing :
a problem-theoretic view.

Notice the similarity between the above set-up and the nomologico-deductive view of the natural sciences. If a hypothesis passes the test, this increases one's confidence in its truthfulness. If, on the other hand, it fails the test, then it is refuted and ought to be revised [Hel]. Tests are conclusive only in the case of refutation, which is another way of stating Dijkstra's well-known remark on program testing.

Now, a program does solve some problem, even if not the one intended. In other words, the input-output behavior of a program describes a problem that it solves. Thus, we can adapt the above view of program testing to specification validation, as explained in the next item.

5.3 Specification Validation

In order to formulate our problem-theoretic view of the process of specification validation, we shall need some simple definitions. Note that the selection of test data for a program amounts to specializing it to this set of inputs. Let us now consider its analogue for problems. First, for problems Q and Q' we say that Q' is a subproblem of Q, denoted by $Q' \subseteq Q$, iff we have $D' = D$, $R' = R$, $q' \subseteq q$, $I' \subseteq I$, and $q' = q \mid I'$. Now, consider specifications s and s' in languages N and N', respectively, perhaps the same. We say that s' is a particularization of s, denoted by $s \text{ -- } \pi \text{ --> } s'$, iff we have $\mu'[s'] \subseteq \mu[s]$, as illustrated in fig. 5.3.

*

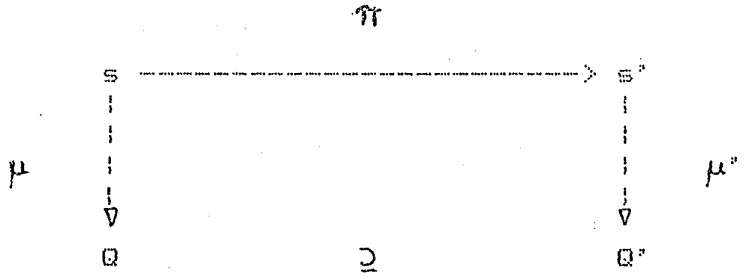


Fig. 5.3 : Particularization transformation π.

Now, consider a problem Q, a perhaps somewhat informal view

of the application concept, and a formal specification s allegedly describing it. We wish to test whether indeed this is so. For this purpose, we first add to s some details so as to describe a simple special case s' . We then read from s' the problem Q' = μ[s'] it describes. Since s' is simple enough by construction, we may trust that this verbalization is correctly done. Finally, we check whether Q' is indeed a subproblem of Q , say, by asking the customer who supplied Q in the first place.

The similarity between program testing and the above validation process is apparent. So, we formulate the latter, by means of our definitions, as follows.

(H) Hypothesis under test : μ[s] = Q (s specifies Q) ;

(A) Auxiliary hypotheses :

s --- ℘ ---> s' (s' is a particularization of s) ,

μ'[s'] = Q' (s' specifies Q') ;

(C) Observable consequence :

Q' ⊆ Q (Q' is a subproblem of Q) .

Now, we have H & A ----> C , by definition of particularization (see fig. 5.3), and A , by construction and the trustworthiness of the simple verbalization. So, the experiment amounts to checking C and refuting H if C fails to hold.

The similarity between specification validation and the hypothesis/theory testing in natural science [Hel] is apparent. In both cases, one actually tests simple observable consequences of the hypothesis by relying on trustworthy auxiliary hypotheses.

5.4 The Software Development Process

The software development process starts with a problem Q to be solved, the application concept, and tries to obtain a program p to solve it. First, problem Q is precisely described by a specification s , the formalization phase : Q -- ϕ --> s . Then, comes the process of program construction, which the simplified inverted U meta-model describes as s α^* . σ . β^* p . In view of the correctness results we have s -- σ --> p , whence we obtain $\nu[\ p\ ] \prec \mu[\ s\ ]$. We cannot be so sure about the connection between p and Q since the formalization phase is not as reliable and validation is not conclusive in the case of a positive outcome. So, the formalization phase is similar to the process of theory construction in natural science, as we have already mentioned and shall elaborate upon. On the other hand, the process of program construction deals with precisely defined formal objects, being more akin to a mathematical activity [BP].

```
*
  if  easy(p) ----> sigma(p)
  []  refinable(p) ----> constr( eta(p) )
  []  decomposable(p) ----> rho( constr( delta(p) ) )
  []  extendable(p) ----> xi( constr(p) )
  fi
```

Fig. 5.4 : Abstract procedure body for program construction.

The software development process can be described by the procedure invocation constr(phi(Q)) , where phi denotes the formalization process and constr(s : S) ----> P is the abstract

nondeterministic procedure whose body, with Dijkstra's guarded commands is shown in fig. 5.4.

Now, let us introduce tests into the picture. Let θ, respectively τ, denote the activity of validating a formal specification s, respectively testing program p, against an informal problem Q, followed by a backtracking with reformulation in case of refutation. The previous remarks indicate that these two activities, depicted in fig. 5.5, are conceptually quite similar.

*

Activity θ :

```
while ¬ satisf_validate( s , Q ) do
  reformulate( s )
end_while
```

Activity τ :

```
while ¬ satisf_test( p , Q ) do
  revise( p )
end_while
```

Fig. 5.5 : Activities θ and τ of specification validation, respectively program testing, with possible backtracking and reformulation.

Let us consider the role of validation in the software process. First, the process of program development with specification validation is described by the regular-like expression $\Phi \cdot \theta \cdot \alpha \quad \sigma \cdot \beta^*$, as illustrated in fig. 5.6. Since validation is not conclusive, we can increase our

confidence in the final result by validating the intermediate specifications, as well, which can be described by the expression $\phi \cdot \theta \cdot (\alpha \cdot \theta)^* \cdot \sigma \cdot \beta^*$ (see fig. 5.7).

*

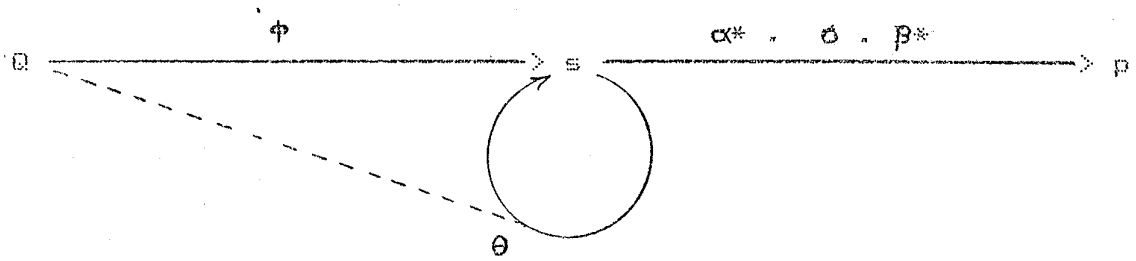


Fig. 5.6 : Process of software development with specification validation.

*

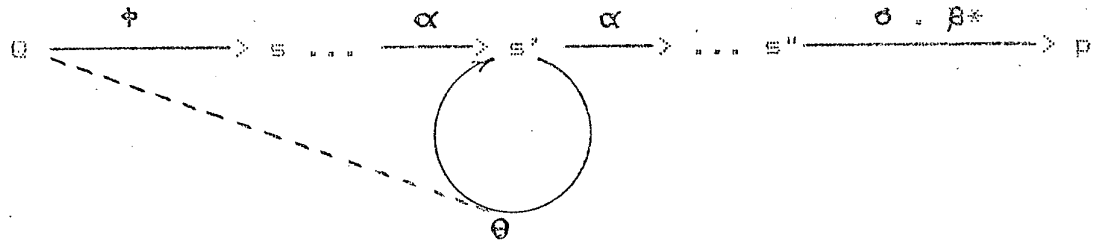


Fig. 5.7 : Process of software development with validation of intermediate specifications.

Now, let us consider the role of program testing. Similarly to the case with specification validations, we can test the various program versions in order to increase our confidence in the final result. These ideas lead to the complete version of the inverted U meta-model for the entire process of software development with intermediate specification validations and program tests, which is illustrated in fig. 5.8. The corresponding regular-like expression for the entire process is :

$\phi \cdot \theta : (\alpha \cdot \theta)^* \cdot \sigma \cdot \tau \cdot (\beta \cdot \tau)^* \cdot$

*

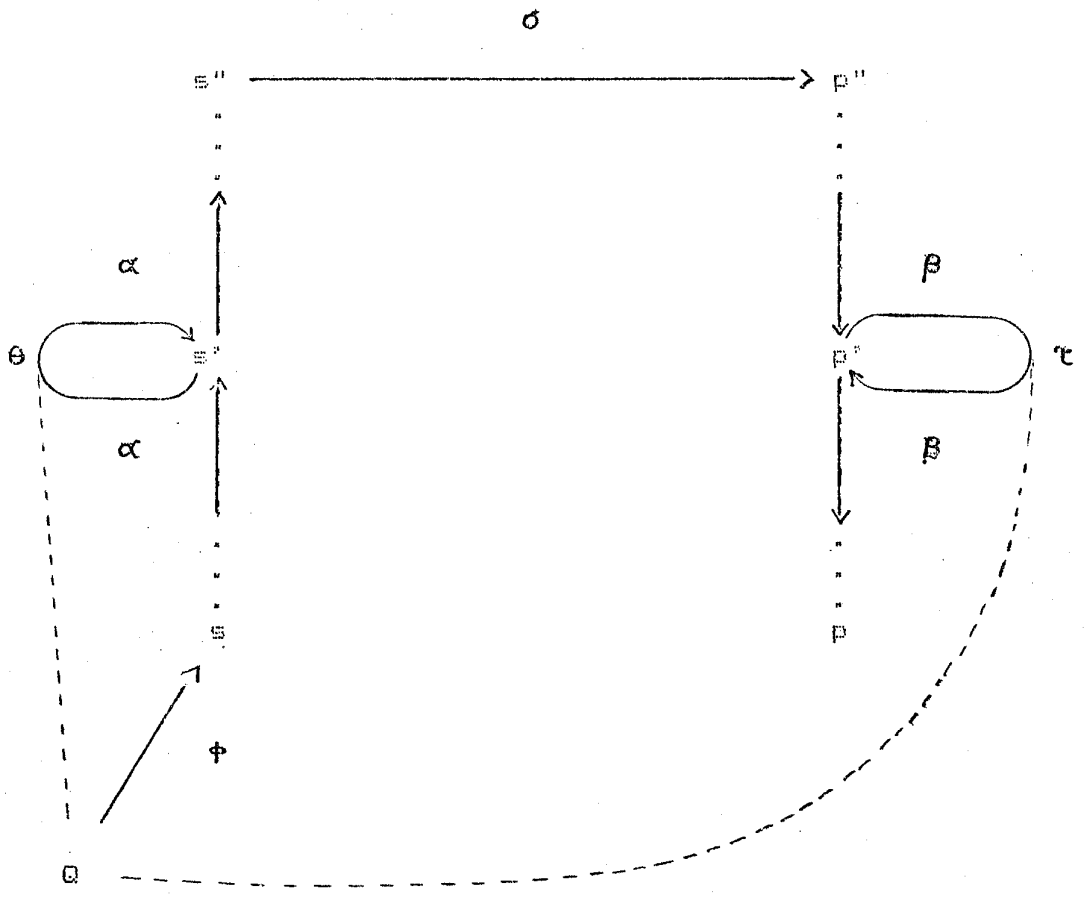


Fig. 5.8 : Software development process with validation and testing of intermediate specifications and programs.

6. CONCLUSION

A precise model of the software development process is needed for understanding it and for the design and implementation of environments to support this process. We have analyzed this process from a problem-theoretic viewpoint by using some logical concepts in order to clarify its semantics and to provide a formal meta-model for it.

The software development process goes from a perhaps informal description of a problem to an efficient program that solves it, via a precise specification. The first phase, of formalization, is akin to the process of theory construction in natural science; then starts the process of program construction, which renders the specification first executable and then efficient. The crucial and central step in program construction amounts to constructing solutions for (component) problems. In preparation for it one may use specification translations with refinements and decompositions. Similarly, the (component) programs are later translated by means of recombinations and extensions.

A problem is defined as a mathematical structure, its solutions being certain kinds of functions from data to results that satisfy the requirement. Some operations on problems, such as sum and product, allow a problem to be decomposed into component problems with the property that a solution for the original problem can be obtained from solutions for the component ones, which is one of the basic ideas of the divide-and-conquer strategy for problem solving. The crucial step in problem solving

is that of solution construction, which amounts to designing an algorithm to compute a function included in the requirement relation of the problem.

The linguistic translations involved in the process of program construction are specification refinement and program extension. They replace a specification or a program by another one, equivalent to it up to a small refinement or extension. Decomposition writes a specification in a parameterized form, whereas recombination builds up a program from component programs; thus they bear some resemblance to the idea of programming with abstract data types.

The process of program construction involves only precise objects, and the correctness criteria for the various steps provide a correctness theorem for the entire process. On the other hand, formalization passes from an informal object to a precise one. As such, it can only be validated by methods similar to the testing of programs and of hypotheses in the nomologico-deductive model of natural science.

Some conclusions are suggested by our analysis and meta-model. First, there is no canonical step for the entire process, since informal descriptions, precise specifications and executable algorithms are fundamentally distinct in nature. Second, decompositions and recombinations, as well as linguistic transformations in general, are constrained by the semantics of problems and solutions.

Program development can also be regarded as a (meta-)

problem. A solution for it amounts to a program development method. Such methods contain both formalized parts and heuristics for applying them. The formalized part guarantees the correctness of the result by means of its theorems, whereas the heuristic one provides a guide as to which theorem should be applied. Our analysis suggests that a widely applicable method is bound to have some heuristic part for selecting the appropriate decompositions. This appears to indicate severe obstacles to the ideas of complete formalization or automation of the whole software development process, even though our meta-model does provide a formalization for a large portion of it as a starting point for its partial, interactive automation.

We should remark that our meta-model has been instantiated to describe some program development methods. Among these, we mention the CIP method and Jackson's method [HVB], as well as Dijkstra's method of predicate transformers.

Another product of our analysis is the explicitation of important requirements to be satisfied by any formalism purporting to describe the software development process. Thus, our problem-theoretic framework, by clarifying the underlying semantics of the software development process, provides a descriptive meta-model for it, besides having prescriptive import.

REFERENCES

[BP] M. Broy ; P. Pepper - " Program development as a formal activity "; IEEE Trans. Software Engin. SE-7 (1), Jan. 1981, p. 14-22.

[BW] F. L. Bauer ; H. Wössner - Algorithmic Language and Program Development; Springer-Verlag, Berlin, 1982.

[En] H. B. Enderton - A Mathematical Introduction to Logic; Academic Press, New York, 1970.

[Go] J. A. Goguen - " Parameterized programming "; IEEE Trans. Software Engin. SE-10 (5), Sept. 1984, p. 528-543.

[HBV] A. M. Haeberer ; G. Baum ; P. A. S. Veloso - " On an algebraic theory of problems " (in Spanish); Proc. IV ETHOS Workshop, Petrópolis, Apr. 1987, p. 123-154.

[HVB] A. M. Haeberer ; P. A. S. Veloso ; G. Baum - Towards a software development process meta-model based on problem theory (in Spanish); Pont. Univ. Católica, Rio de Janeiro, Res. Rept. MCC 3/87, Oct. 1987.

[He] C. Hempel - Aspects of Scientific Explanation and Other Essays in the Philosophy of Science; Free Press, New York, 1965.

[Ho] C. A. R. Hoare - " Data structures "; in R. T. Yeh (ed.) Current Trends in Programming Methodoldogy, vol IV : Data Structuring; Prentice-Hall, Englewood Cliffs, 1978, p. 1-11.

[HS] E Horowitz ; S. Sahni - Fundamentals of Computer Algorithms; Computer Science Press, Potomac, 1978.

[Hu] J. W. 'Hughes - " A formalization and explication of the Michael Jackson method of program construction "; Software - Practice and Experience 9 (3), 1979, p. 191-202.

[Ja] M. A. Jackson - Principles of Program Design; Academic Press, London, 1980.

[LB] M. M. Lehman ; L. A. Belady (eds.) - Program Evolution : Processes of Software Change; Academic Press, London, 1985.

[LST] M. M. Lehman ; V. Stenning ; W. M. Turski - Another look at software design methodology; Imp. Coll. Sci. Techn., London, Res. Rept. DoC 83/13, Jun. 1983.

[LZ] B. Liskov ; S. Zilles - " Programming with abstract data types "; SIGPLAN Notices 9 (4), Apr. 1974, p. 50-59.

[Ma] Z. Manna - The Mathematical Theory of Computation; McGraw-Hill, New york, 1974.

[MT] T. S. E. Maibaum ; W. M. Turski - " On what exactly is going on when software is developed step by step "; Proc. 7th Intern. Conf. Software Engin., IEEE Computer Science Press, Silver Spring, 1984, p. 525-533.

[Po] G. Polya - How to Solve it : a new aspect of the mathematical method; Princeton Univ. Press, Princeton, 1957.

[Ro] H. Rogers,Jr. - The Theory of Recursive Functions and Effective Computability; McGraw-Hill, New York, 1962.

[Si] M. Sintzoff - Desiderata for a design calculus; Univ. Cath. Louvain, Unitè d'Informatique, memo UCL T3, Jun. 1985.

[Sh] J. R. Shoenfield - Mathematical Logic; Addison-Wesley, Reading, 1967.

[TH] T. Takahashi ; A. M. Haeberer - ETHOS Project : final report of the preliminary phase (in Spanish), Campinas, Jan. 1988.

[Tk] W. M. Turski - " The design of large programs "; in C. Floyd ; H. Kopetz (eds.) Software Engineering : Entwurf und Spezifikation, Teubner, Stuttgart, 1984, p. 129-160.

[TM] W. M. Turski ; T. S. E. Maibaum - The Specification of Computer Programs; Addison-Wesley, Wokingham, 1987.

[V80] P. A. S. Veloso - " Divide and conquer via data-types "; Proc. VII Latin American Computer Science Conf., Caracas, Jan. 1980, p. 530-539.

[V84] P. A. S. Veloso - " Outlines of a mathematical theory of general problems "; Philosophia Naturalis 21 (2-4),1984, p. 354-367.

[VMS] P. A. S. Veloso ; T. S. E. Maibaum ; M. R. Sadler - " Programme development and theory manipulation "; Proc. Intern. Workshop on Software Specification and Design, London, Aug. 1985, p. 155-162.

[VPMJ P. A. S. Veloso ; F. E. P. Pessoa ; T. S. E. Maibaum - " Theory of abstract data types for programming : a logical approach " (in Portuguese); Proc. IX Latin American Computer Science Conf.; Lima, Jan. 1982, p. 423-430.

[VV] P. A. S. Veloso ; S. R. M. Veloso - " Problem decomposition and reduction : applicability, soundness, completeness "; in R. Trappl ; J. Klir ; F. Pichler (eds.) Progress in Cybernetics and Systems Research, vol VIII, Hemisphere, Washington, 1981, p. 199-203.