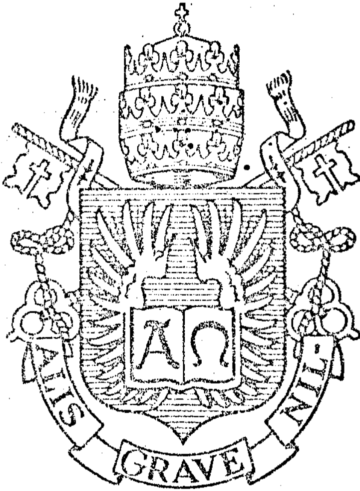

PUC

Series : Monografias em Ciência da Computação

No. 7 / 89

ON THE REQUIREMENT - SPECIFICATION - PROGRAM TRIANGLE : A THEORETICAL ANALYSIS

Armando M. Haeberer

Paulo A. S. Veloso

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 — CEP 22453

RIO DE JANEIRO — BRASIL

PUC-RJ - Departamento de Informática

Series : Monografias em Ciência da Computação

No. 7 / 89 July 1989

Series Editor : Paulo A. S. Veloso

ON THE REQUIREMENT - SPECIFICATION - PROGRAM TRIANGLE : A THEORETICAL ANALYSIS +

Armando M. Haeberer *

Paulo A. S. Veloso

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

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

In charge of publications:

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

ABSTRACT

The interconnections among requirements, specifications and programs are analyzed from a theoretical standpoint by means of Carnap's Two-Level Theory of the Language of Science and the Algebraic Theory of Problems.

The goal of software development is argued to be the construction of an engineering model of the application concept. The synthetic relation "being-an-engineering-model" is argued to be a disposition, which has interesting consequences. On the one hand, specification validation and program testing are inevitable, but the latter must be preceded by some correctness verification. On the other hand, the software process exhibits an inherent non-monotonocity, both globally and locally.

Within this formal framework we are able to state, and establish, in a precise way some facts that are generally believed on intuitive grounds only.

Keywords : Software development process, formal specifications, correctness, specification validation, program testing, non-monotonocity, software development process formalisms, epistemology.

RESUMO

As relações entre requisitos, especificações e programas são analisadas de um ponto de vista teòrico por meio da Teoria dos Dois Nìveis da Linguagem da Ciência de Carnap e da Teoria Algèbrica de Problemas.

Argumenta-se que o objetivo do processo de desenvolvimento de programas è a construção de um modelo "engenheril" do conceito de aplicação. Mostra-se que a relação sintètica de ser-úm-modelo-"engenheril" è uma disposição, o que tem conseqüências interessantes. Por um lado, validação de especificações e teste de programas são inevitàveis, mas este deve ser precedido por alguma verificação de corretude. Por outro lado, o processo de programção exibe uma não monotonia inerente, tanto a nìvel global quanto local.

Com este arcabouço formal conseguimos enunciar, e demonstrar, de maneira precisa alguns fatos que são geralmente aceitos apenas por razões intuitivas.

Palavras chave : Processo de desenvolvimento de programas, especificações formais, corretude, validação de especificações, teste de programas, não monotonia, formalismos para processo de desenvolvimento de programas, epistemologia.

RESUMEN

Se analizan las relaciones entre requerimientos, especificaciones y programas desde un punto de vista teòrico por medio de la Teoría de los Dos Niveles del Lenguaje de la Ciencia de Carnap y de la Teoría Algebraica de Problemas.

Se argumenta que el objetivo del proceso de desarrollo de software es la construcciòn de un modelo ingenieril del concepto de aplicaciòn. Se muestra que la relaciòn sintètica "ser-un-modelo-ingenieril" es una disposiciòn, lo cual tiene consecuencias interesantes. Por una parte, la validaciòn de especificaciones y el testeo de programas son inevitables, pero èste debe ser precedido por alguna verificaciòn de corrección. Por otra parte, el proceso de programaciòn exhibe una no monotonìa inherente, tanto a nivel global como local.

En este marco formal logramos enunciar, y demostrar, de manera precisa algunos hechos que son generalmente aceptados sòlo por razones intuitivas.

Palabras clave : Proceso de desarrollo de software, especificaciones formales, corrección, validaciòn de especificaciones, testeo de programas, no monotonìa, formalismos para proceso de desarrollo de software, epistemologìa.

Table of Contents

I. INTRODUCTION. 1
II. THE GOAL OF THE SOFTWARE PROCESS. 2
II.1 The application-concept and its verbalization 2
II.2. The concept of virtual machine. 3
II.3. The relation of being-an-engineering-model 3
II.3.1 A naive hypothetico-deductive experiment for being-an-engineering-model. 4
II.3.2. The inadequacy of operational definitions. 4
III. THE TWO -LEVEL THEORY OF THE LANGUAGE OF SCIENCE 5
III.1. Classification of the scientific statements. 5
III.2. The introduction of dispositions into the language of science. 5
III.2.1. Reductive sentences. 6
III.2.2. The decidability of observational terms 8
III.3. The Two-Level Theory. 9
III.3.1. Motivations. 9
III.3.2. Overview of the Two-Level Theory and of a new version of the hypothetico-deductive method. 9
III.3.3. The Observational Language 10
III.3.4. The theoretical Language. 10
III.3.5. Correspondence Rules 11
IV. THE SOFTWARE PROCESS GOAL REVISITED 11
IV.1. The observational indecidibility of the naive version of being-an-engineering-model. 11
IV.2. Halting of virtual machine and program termination. 11
IV.3. Observational and theoretical aspects of being-an-engineering-model 12
IV.4. A new hypothetico-deductive experiment for being-an-engineering-model 13
V. THE ALGEBRAIC THEORY OF PROBLEMS. 14
V.1 The concept of problem and solution. 14
V.2 Relaxation of problems. 15
V.3. Operations on problems. 15
V.3.1. Sum of problems. 15
V.3.2. Additive subproblems. 16
V.3.3. Complete additive subproblems. 16
V.3.4. Product of problems. 16
V.3.5. Direct product of problems.. 17
V.4. Operations on problem solutions 17
V.4.1. Sum of solutions 17
V.4.2. Product of problem solutions. 18
V.4.3. Direct product of problem solutions. 18
V.5. Relaxation and the operations on problems 18
V.6. Reduction and abstraction. 18
VI. OBSERVATIONAL AND THEORETICAL OBJETS IN THE SOFTWARE PROCESS. 19
VI.1. The observational level for the software process. 19

VI.2. The theoretical level for the software process. 20
VI.2.1. Specifications, problems, programs and solutions. 20
VI.2.2. Target machines and wide-sense solutions 21
VI.3. Application concept, specifications, programs and virtual machines. 23
VII. PROBLEMS IN THE FACTORIZATION OF BEING-AN-ENGINEERING-MODEL. 25
VII.1. The synthetic character inherent to the software process. 25
VII.2. The inevitability of validation. 26
VII.3. The inevitability of vertical verification. 27
VII.4. The inherent non-monotonicity of software development. 27
VII.4.1. Global non-monotonicity. 27
VII.4.2. Local non-monotonicity 27
VIII. CONCLUSIONS 28

I. INTRODUCTION.

In this paper we will analyze how requirements, informal and formal specifications and programs relate among themselves, emphasizing the non-monotonicity of software development.

Although extensively treated in the literature, there is a certain degree of confusion about the character of these interconnections, as well as about the concepts, activities and obligations derived from them. This confusion derives from the informal and superficial treatment usually given to these concepts. In addition, some questions have been answered ambiguously or only in part. For instance, what are the relationships between verification and validation, how do they together ensure correctness?

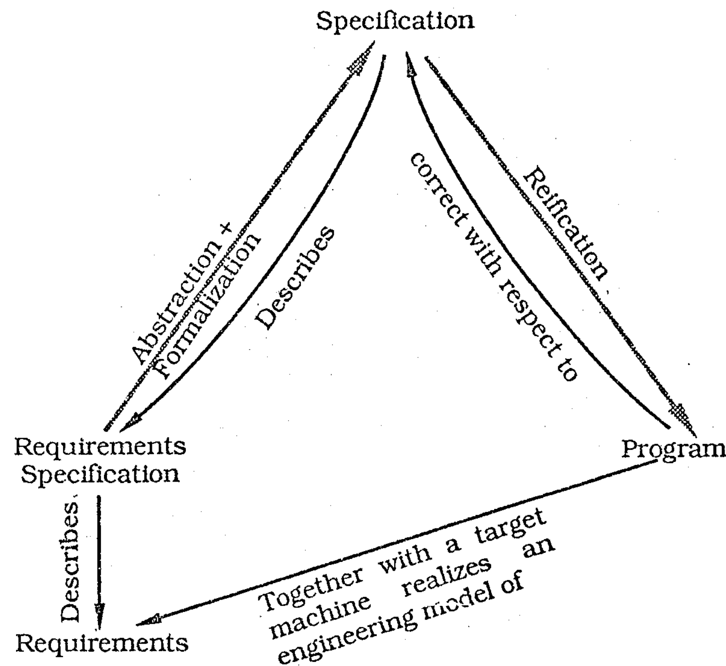

Figure I.1.

Software development, at a high level of abstraction, may be regarded as involving four main objects. They are: the requirements, reflecting the *real problem* extension; the requirement specification, as the informal description of the requirements; the formal specification[1], as its formal description; and the program giving rise to a virtual machine, as the final product of the process.

To elude the above mentioned informality and superficiality of analysis, one should embed it into a formal framework. This framework should have two main goals. First, its formalism should allow the treatment of software processes and their components at any level of abstraction. Second, its formal structure should shed light on concepts like validation, verification, informal description, formal specification, abstraction, etc. We will use as formalism the Algebraic Theory of Problems [Vel84;Hae87;Hae89;Var89], and as formal structure Rudolf Carnap's Two-Level Theory of the Language of Science [Car56;Ste70;Hem65].

Some terms related to software processes, software engineering, etc., are not very well defined. In order to achieve precision, we will resort to a terminology like that of [Tur87;Mai84;Leh84], instead of the "classical" software engineering one [Boe76;Ton79;Agr86]. So, we will use application concept for requirements,

---

[1] A common misconception is that of considering formal specification only as formulas in some mathematical language. But, an abstract program is a formal, though somewhat odd, specification. Furthermore, a rapid prototype is a formal specification, probably incomplete or wrong in its early stages, but obviously formal.

verbalization for requirement specification, virtual machine for implemented program, etc., as explained in section II.

Within this formal framework we will, then, analyze relations such as "describes", "realizes an engineering model", "is correct", etc., as well as the validation and proof obligations involved in factorizing the diagram in figure I.1.

The structure of this paper is as follows. In the next section we analyze the goals of software development in terms of the relation being-an-engineering-model. Section III outlines the Two-level Theory of the Language of Science in our context, which we use to reexamine the software process goal in section IV. Then, section V sketches the main concepts and results of the algebraic theory of problems needed here. Sections VI and VII contain the main results of our analysis. In section VI we establish the formal basis of the factorization of the software process, leaving the analysis of its problems for section VII.

II. THE GOAL OF THE SOFTWARE PROCESS.

The goal of the software process is the construction of a software artifact. This construction starts from a description of some *real problem*. The software artifact, together with a given target machine , is to behave as an engineering model of the *real problem*.

Although this statement is quite informal and vague-which we will try to fix in the sequel- it suffices to show the point we are trying to emphasize here, namely the enormous difficulty in achieving the goal. This difficulty stems from the fact that the description of the *real problem* is usually vague, informal, ambiguous, and lacking in details, whereas the software artifact is a complex syntactic object belonging to a formal language.

Thus, the software process ranges over a wide spectrum of activities, such as formalization, abstraction, interpretation, construction of solutions, development of algorithms, validation, verification, and so forth.

In addition, the lack of precision in the initial description of the *real problem* prevents this process from being linear. Backtracking occurs frequently due to, for instance, the use of the formality and precision inherent to specification and programming languages as heuristic tools for the understanding and clarification of the initial description.

II.1 The application-concept and its verbalization.

Two terms related to the *real problem*, application concept and verbalization, should be made more precise .

II.1.1. Convention. By *application concept* we mean the extensional knowledge about the *real problem* that serves as starting point of the software process. We shall generally use A to refer to an application concept.

II.1.2. Convention. By a *verbalization* of the application concept A we mean a meta-linguistic description $\mathcal{V}_A$ of this application concept.

Application concept is the knowledge about the *real problem*, as detailed and clear as one understands it at a given moment. Thus, an application concept may be vague, not detailed, etc. But it may also be as precise and detailed as one wishes; such would be the case if one knew exactly the extension of the *real problem*. So, an application concept may be initially wrong and may be corrected by backtracking in view of the heuristic power of the development process of a software artifact.

On the other hand, a verbalization of an application concept amounts to its description in the meta-language. So, a verbalization will be incomplete if the application concept is incomplete, but no matter how detailed and precise is the latter, ambiguities and fuzziness of the meta-language carry over to verbalizations expressed in it.

Since the application concept is an extensional object, it cannot be an informal one. The application concept is an observable object, which can be ill-determined because of lack of knowledge about its extension. Its verbalization, from which the software process starts, is the informal and ambiguous object that is often mistaken for the application concept itself[1].

II.1.3. Convention. We will say that a *data* δ *belongs to the domain of* A, which we will be denoted by $\mathcal{B}\delta A$, iff δ designates an acceptable input for A.

II.1.4. Convention. We will say that the ordered pair $\langle\delta, \rho\rangle$ is an *instance of* A, denoted by $I\delta\rho A$, iff δ belongs to the domain of A and ρ designates an acceptable output for A corresponding to input δ.

We will accept as input and output any pair of *observable events* related by the application, in the sense of belonging to the extension of the *real problem.*

II.2. The concept of virtual machine.

A software artifact is a program or a set of programs written in some programming language to be interpreted by a target machine. So a software artifact is a formal syntactic object that, upon interpretation by a target machine, realizes a device that accepts data δ and produces results ρ.

II.2.1. Convention. By *target machine* we mean a device, made out of hardware and software, that is capable of interpreting a program p. We shall generally use H to refer to a target machine.

II.2.2. Convention. By *virtual machine* we mean the device constructed by interpreting a software artifact p on a target machine H. We shall generally use m_{pH} to refer to such a virtual machine.

II.2.3. Convention. By the *result of virtual machine* m_{pH} *for data* δ *at instant* t, denoted by $m_{pH}\,t(\delta)$, we mean the output [2] ρ produced by m_{pH} at instant t after being fed data δ, provided that it halts.

II.3. The relation of being-an-engineering-model.

We will now try to state in precise terms the meaning of the relation of being-an-engineering-model. This relation, denoted by $m_{pH} <\!| A$, connects a virtual machine m_{pH} to an application A. Clearly, the observation of property $m_{pH} <\!| A$ on a virtual machine m_{pH} presupposes a systematic activity. This activity can be described as follows:

---

[1] The terms application concept and verbalization are not well established and vary from one author to another. The point here is, no matter what names we use, there are two distinct objects. The one is an extensional object, which can be incomplete or ill-detailed at the very beginning of the development process, but becomes completely determined at process completion. The other is the description of the extensional one in the meta-language and may remain ambiguous regardless of its degree of detail.

[2] Here we mean the machine output, rather than the application output mentioned in convention II.1.4. In both cases we regard "output", "input", "halts", etc., as primitive terms whose meaning is assumed to be understood.

i) a data δ from the domain of A is selected,

ii) data δ is introduced into machine m_{pH} at some instant t_o,

iii) if m_{pH} does not halt after (ii) then it is not the case that $m_{pH} <\mid A$,

iv) if m_{pH} does halts after (ii) and $\neg(\mathcal{I}\delta m_{pH} t(\delta)\ A)$ then $\neg(m_{pH} <\mid A)$,

v) if m_{pH} halts after (ii) and $\mathcal{I}\delta m_{pH} t(\delta)\ A$ then we may assume $m_{pH} <\mid A$.

We can attempt to give a formal definition of the relation $m_{pH} <\mid A$. Let us first establish some abbreviations , namely

$\mathcal{A}m_{pH}\delta t_o$: machine m_{pH} is applied to data δ at instant t_o ,

$\mathcal{H}m_{pH}\delta t$: machine m_{pH} halts at instant t for data δ.

We can now give a first precise definition of being -an-engineering-model.

II.3.1. Definition .

$$m_{pH} <\mid A \leftrightarrow (\forall\delta)\ (\forall t_o)\ [\ \mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_o \rightarrow (\exists t)\ (t > t_o \wedge \mathcal{H}m_{pH}\delta t) \wedge \mathcal{I}\delta m_{pH} t(\delta)\ A\ )]$$

II.3.1 A naive hypothetico-deductive experiment for being-an-engineering-model. Why have we state "we may assume $m_{pH} <\mid A$ " instead of directly "$m_{pH} <\mid A$" in item (v) of the preceding paragraph? The answer resides in the asymmetry of factual hypotheses.

To understand this asymmetry we must realize the factual character of definition II.3.1. It stems from the mere presence of a universal quantification over the set of data δ belonging to the application domain. Except in some very special cases, this set of data is not exhaustible, and any attempt to define it by comprehension will be marred by the informality and ambiguity of the meta-language.

So, one can never actually fulfill the condition "for all data δ" in the definiens of that definition. One generally induces this condition from a certain set of data. Hence, definition II.3.1 is in fact a hypothesis in the sense of empirical science. The hypothetico-deductive method [Hem65] then suggests an experiment[1] for it, which we can describe naively as follows:

Main hypothesis:
H: $m_{pH} <\mid A$.
Auxiliary hypotheses:
H_1: $\mathcal{B}\delta_i A$ for i=1, . . . , n.
H_2: $\mathcal{A}m_{pH}\delta_i t_o^i$ for i=1, . . . , n.
The observational consequence is:
Oc: $(\exists t)\ (t > t_o^i \wedge \mathcal{H}m_{pH}\delta_i t) \wedge \mathcal{I}\delta_i m_{pH} t(\delta_i)\ A\ )$ for i=1, . . . , n.

Therefore $H \wedge H_1 \wedge H_2 \rightarrow Oc$. If the experiment rejects *Oc*, then we get by *modus tollens* $(H \wedge H_1 \wedge H_2 \rightarrow Oc) \wedge \neg\ Oc \vdash \neg(H \wedge H_1 \wedge H_2)$. Hence $\neg\ H \vee \neg(H_1 \wedge H_2)$. As we assume H_1 and H_2 true by construction, by applying *modus ponendo tollens*, we refute H. If, on the other hand, the experiment does not falsify *Oc*, then we have

[1] The existential quantifier in the observational consequence will be discussed later.

$((H \wedge H_1 \wedge H_2) \rightarrow Oc) \wedge Oc$. From this we are not entitled to deduce H, for we would be resorting to a well-known fallacy. In this case, we can accept $(\mathcal{B}\delta_i A \wedge \mathcal{A}m_{pH}\delta_i t_o^i \wedge (\exists t)(t > t_o^i \wedge \mathcal{H}m_{pH}\delta_i t) \wedge I\delta_{i\ pH}\ t(\delta_i)\ A))$ for i=1, . . . , n, only as a good inductive support for $m_{pH} \lhd A$[1].

II.3.2. The inadequacy of operational definitions. Definition II.3.1 belongs to a well-known class, that of operational definitions. This name arises from the fact that the definiens of such definitions involves an "operation", in our case the application of machine m_{pH} to the data δ_i, $1 \leq i \leq n$.

This kind of definition appears to embody our intuition. The properties we are dealing with involve systematic observation of reactions to operations. Thus, it seems reasonable to use such operations and reactions in a definiens.

Properties whose observation involves a systematic activity are called *dispositions*. Examples of dispositions are "soluble", "breakable", "magnetic", etc. Operational definitions were proposed as the way to introduce such dispositions into the language of science.

Unfortunately, as Carnap showed in a now classical argument [Car36], operational definitions are too wide and so fail to accomplish their goal.

Let us point out this problem in our definition II.3.1. Our analysis was based on the fact that test conditions $\mathcal{B}\delta A$ and $\mathcal{A}m_{pH}\delta t_o$ were true. But what if these conditions did not hold? Then, the antecedent of the definiens of our definition would be false and, hence, the implication true. Consider a machine m_{pH} that has never been tested with respect to application concept A. We then have $(\forall\delta)(\forall t_o)(\mathcal{B}\delta A \rightarrow \neg\mathcal{A}m_{pH}\delta t_o)$, hence $(\forall\delta)(\forall t_o)\neg(\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_o)$. Therefore $(\forall\delta)(\forall t_o)(\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_o \rightarrow (\exists t)(t > t_o \wedge \mathcal{H}m_{pH}\delta t) \wedge I\delta m_{pH} t(\delta)\ A))$ and immediately $m_{pH} \lhd A$. In other words, from II.3.1 we can conclude that a virtual machine that has never been tested with respect to an application concept A turns out to be an engineering model of A!

This is a troublesome feature of operational definitions. Such features led Carnap [Car36; Car56] to the proposal of the Two-Level Theory of the Language of Science [Hem65,71].

The theoretical basis of this paper consists of Carnap's theory and the Algebraic Theory of Problems [Vel84;Hae87,89;Var89]. We will outline them in the sequel.

III. THE TWO-LEVEL THEORY OF THE LANGUAGE OF SCIENCE.

III.1. Classification of the scientific statements.

Throughout this paper we shall be dealing with analytically determinate and synthetic statements. So, we will give an idea of the meaning of such terms by briefly presenting the classification of scientific statements.

We shall gloss over some polemical issues. These include the demarcation criteria for empirical meaning and empiricists' thesis of the analytico-synthetic dichotomy. We shall, however, assume that every meaningful scientific statement can be classify as either analytically determinate or synthetic [Ste70].

The class of analytically determinate statements includes those statements whose truth can be determined by a mere analysis of meaning. In this class we find the purely logico-formal truths and the logical falsities, i.e., statements whose truth or falsity is completely determined by the meaning of the logical symbols (connectives, quantifiers, etc.). To these we add the analytical truths (consequences of statements where the meaning relations among descriptive expressions are fixed) and their negations (the analytical falsities).

---

[1] Here , for simplicity, we gloss over the condition "for all t_o".

The synthetic statements are those that are not analytically determinate; their truth can be analyzed only experimentally. Thus the synthetic statements can be identified with the empirically determinate ones.

Summing up, synthetic statements are those testable only by experiments, whereas the analytically determinate statements are those that can be formally proved.

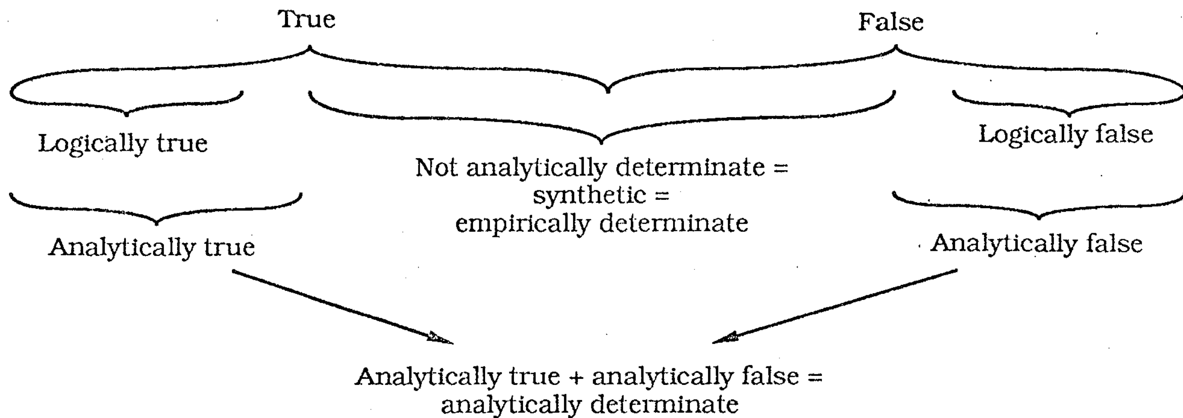

Figure III.1.1

III.2. The introduction of dispositions into the language of science.

Directly observable properties, such as "red", "liquid", etc. appear in the language of science as primitive predicates. For any systematic construction of science, one should have few primitive predicates. On the other hand many properties are dispositions[1] [Ste70]. Thus, one cannot take all dispositional predicates as primitive, if one wishes a simple system.

As mentioned, the use of operational definitions to introduce dispositional predicates into the language of science presents problems. These difficulties appear to be easily overcome, since they arose from the use of material implication to interpret the conditional "if ...then...". Therefore we can naively suppose that the solution lies simply in changing this interpretation.

In other words this proposal to save operationalism amounts to restricting the truth function of the conditional. There were some attempts of this kind, such as:

i) replacement of material implication $\rightarrow$ by causal implication $\underset{c}{\rightarrow}$. So, "$p \underset{c}{\rightarrow} q$" shall mean not only that q holds if p does, but that q holds with causal necessity if p holds.

ii) replacement of material implication $\rightarrow$ by subjunctive conditional $\Rightarrow$. The definiens $(\forall\delta)\,(\forall t_0)\,(\,\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_0 \Rightarrow (\exists t)\,(t > t_0 \wedge \mathcal{H}m_{pH}\delta t) \wedge \mathcal{I}\delta m_{pH}t(\delta)A\,))$ of definition II.3.1 would be understood roughly as follows: if at time t_0 one have introduced data δ of the domain of A into machine m_{pH}, then the result $m_{pH}t(\delta)$ obtained at time $t>t_0$ would be an acceptable result for δ in this application.

Unfortunately these attempts so far lack precision in the basic concepts: $\underset{c}{\rightarrow}$ and $\Rightarrow$.

III.2.1. Reductive sentences.

A better approach to the introduction of dispositions was Carnap's proposal of replacing operational definitions by reductive sentences. Following his proposal one constructs a generalized conditional sentence having as antecedent the predicates corresponding to the action and related experimental conditions, while its consequent relates the dispositional predicate to the reaction predicates. Consider, for

[1] It seems that all the proprieties of virtual machines are dispositions; consider complexity, efficiency, etc.

instance, the disposition introduced in definition II.3.1. The corresponding reductive sentence would be:

III.2.1.1. $(\forall\delta)\,(\forall t_o)\,(\,\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_o \rightarrow$

$$(\, m_{pH} \lhd A \leftrightarrow ((\exists t)\,(t > t_o \wedge \mathcal{H}m_{pH}\delta t \wedge \mathcal{I}\delta m_{pH}\, t(\delta)A)))$$

Such reductive sentences involve predicates introducing properties known as permanent dispositions [Car36]. They present an intrinsic difficulty arising from the fact that the dispositional predicate occurs within the scope of quantifiers over variables other than its free variables. In our case, dispositional predicate $m_{pH} \lhd A$ does not involve variables δ and t_o. This yields a logical contradiction if for some values of δ and t_o formula $((\exists t)\,(t > t_o \wedge \mathcal{H}m_{pH}\delta t) \wedge \mathcal{I}\delta m_{pH}\, t(\delta)A))$ holds while failing for other such values. We will overcome this difficulty by replacing the bilateral reductive sentence introducing $m_{pH} \lhd A$ by the pair :

III.2.1.2. $(\forall\delta)\,(\forall t_o)\,(\,\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_o \rightarrow$

$$(\, m_{pH} \lhd_{\delta t_o} A \leftrightarrow ((\exists t)\,(t > t_o \wedge \mathcal{H}m_{pH}\delta t \wedge \mathcal{I}\delta m_{pH}\, t(\delta)A)))$$

a reductive sentence for the "local instantaneous disposition" $m_{pH} \lhd_{\delta t_o} A$, and

III.2.1.3. $m_{pH} \lhd A \leftrightarrow (\forall\delta)\,(\forall t_o)\,(\,\mathcal{B}\delta A \rightarrow (\, m_{pH} \lhd_{\delta t_o} A))$

which defines $m_{pH} \lhd A$ in terms of $m_{pH} \lhd_{\delta t_o} A$.

The value of this modification can be assessed from distinct viewpoints.

i) From the viewpoint of the theory of definitions, III.2.1.2 amounts to what is known as a conditional definition: the definiendum is related to the definiens under the condition $\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_o$.

ii) From the viewpoint of philosophy of science, there is a great difference between III.2.1.2 and an operational definition. Under the light of the principle of partial interpretation of the theoretical terms (which will be discussed later on), reductive sentences such as III.2.1.2 are already cases of a partial axiomatic characterization of a dispositional term, namely $m_{pH} \lhd_{\delta t_o} A$.

The inadequacy of operational definitions, in the sense of being too wide, disappears upon their replacement by reductive sentences. If condition $\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_o$ does not hold, the previous puzzling conclusion is now replaced by an indetermination about the presence of the disposition.

On the other hand, a definition must satisfy the so called "eliminability principle": the definiendum can be replaced by the definiens in every context. Reductive sentences do not satisfy this principle.

Thus two differences between definitions and reductive sentences are:

a) reductive sentences do not satisfy the eliminability principle,

b) the meaning of the concepts introduced by reductive sentences is determined only partially.

The first of these differences can be crucial if one requires every new concept to be totally reducible to primitive ones. But this is one of the requirements abandoned by Carnap in introducing two levels in the language of science [Car56]. The second one requires the definition of some procedures to restrict the above mentioned margin of indetermination.

Two procedures have been proposed to carry out that restriction [Ste70].

i) The first procedure applies to the case of a single reductive sentence and uses inductive arguments. Assume one has only the reductive sentence III.2.1.2:

$$(\forall\delta)\ (\forall t_o)\ (\ \mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_o \rightarrow (\ m_{pH} \lhd_{\delta t_o} A \leftrightarrow ((\exists t)\ (t > t_o \wedge \mathcal{H}m_{pH}\delta t) \wedge \mathcal{I}\delta m_{pH}\ t(\delta)A)))$$

Let δ_i, for i=1, ..., n, be n test data in the domain of A, i.e., for which conditions $\mathcal{B}\delta_1 A, \ldots, \mathcal{B}\delta_n A$ hold. Assume they have been put to test, i.e., $\mathcal{A}m_{pH}\delta_1 t_o^1, \ldots, \mathcal{A}m_{pH}\delta_n t_o^n$ hold, and all the reactions have been positive. Then, the n formulas $(\exists t)\ (t > t_o^1 \wedge \mathcal{H}m_{pH}\delta_1 t) \wedge \mathcal{I}\delta_1 m_{pH} t(\delta_1)A), \ldots, ((\exists t)\ (t > t_o^n \wedge \mathcal{H}m_{pH}\delta_n t) \wedge \mathcal{I}\delta_n\ _{pH} t(\delta_n)A)$ may be regarded as a "valid inductive support" for $m_{pH} \lhd A$, which states that over the entire domain of A virtual machine m_{pH} is an engineering model of A (see formula III.2.1.3).

In fact, the indetermination of the dispositional concept is not actually restricted by this procedure. Rather, by including intuitive inductive considerations, one assumes that it holds in general. From now on, we assume that the disposition holds at any time if it holds at a given time t_o.

ii) The second procedure restricts the indetermination by introducing additional reductive sentences for the same dispositional predicate. Consider a dispositional predicate $\mathcal{D}x$ with experimental condition $\mathcal{B}_1 x$ and reaction $\mathcal{S}_1 x$. A reductive sentence introducing this disposition is

III.2.1.4. $(\forall x)\ (\ \mathcal{B}_1 x \rightarrow (\ \mathcal{D}x \leftrightarrow \mathcal{S}_1 x\ ))$

Consider other experimental conditions $\mathcal{B}_2 x$, $\mathcal{B}_3 x$, ... and corresponding reactions $\mathcal{S}_2 x$, $\mathcal{S}_3 x$, ... Then, one can add

III.2.1.5. $(\forall x)\ (\ \mathcal{B}_2 x \rightarrow (\ \mathcal{D}x \leftrightarrow \mathcal{S}_2 x\ ))$

III.2.1.6. $(\forall x)\ (\ \mathcal{B}_3 x \rightarrow (\ \mathcal{D}x \leftrightarrow \mathcal{S}_3 x\ ))$

....

To ensure that each experimental condition $\mathcal{B}_i x$ is realizable, Carnap requires $(\forall x)\ (\neg\mathcal{B}_i x)$ (para i= 1, 2, ...) to be non-deductible from accepted scientific laws.

Before applying this procedure, indetermination appeared when experimental condition $\mathcal{B}_1 x$ did not hold. Even after applying it, there will in general remain some margin of indetermination. It could only be avoided if we had $(\forall x)\ (\mathcal{B}_1 x \vee \mathcal{B}_2 x \vee \mathcal{B}_3 x \vee \ldots \vee \mathcal{B}_n x)$ as an accepted scientific law.

In any case we now have a third difference between reductive sentences and definitions.

c) The introduction of a dispositional term by means of more than one reductive sentence gives rise to the establishment of an empirical hypothesis.

As an example, consider an object a satisfying experimental conditions $\mathcal{B}_1 a$ and $\mathcal{B}_2 a$, as well as $\mathcal{S}_1 a$ but not $\mathcal{S}_2 a$. In view of III.2.1.4 $\mathcal{D}a$ should hold, but by III.2.1.5 $\neg\mathcal{D}a$ should hold as well. Since this is a contradiction, III.2.1.4 and III.2.1.5 entail $\neg(\forall x)(\mathcal{B}_1 x \wedge \mathcal{B}_2 x \wedge \mathcal{S}_1 x \wedge \neg\mathcal{S}_2 x)$ and hence $(\forall x)\ (\ \mathcal{B}_1 x \wedge \mathcal{B}_2 x \rightarrow (\ \mathcal{S}_1 x \leftrightarrow \mathcal{S}_2 x\ ))$.

These two statements are synthetic rather than analytically determinate. They express that whenever $\mathcal{B}_1 x$ and $\mathcal{B}_2 x$ hold, then $\mathcal{S}_1 x$ holds iff $\mathcal{S}_2 x$ does.

Nevertheless further empirical research may reveal that this statement is not correct[1].

Note that in a strict logical sense the introduction of a disposition by means of several reductive sentences gives rise to a non-conservative extension [Sho67].

Despite the, supposedly odd, features a, b and c, of reductive sentences , there seems to be no reason to abandon this method of introducing dispositional predicates into the language of science. But two new unsurmountable difficulties arose at this point.

The first difficulty is the matchlessness of the reductive sentence method with actual behavior of aworking scientist facing a negative result of an experiment. Indeed, consider the reductive sentence $(\forall x)\ (\mathcal{B}_1 x \rightarrow (\ \mathcal{D}x \leftrightarrow \mathcal{S}_1 x\ ))$. Assume that the corresponding experiment on object a has a negative outcome: $\mathcal{B}_1 a \wedge \neg\mathcal{S}_1 a$ holds. One then has $\neg\mathcal{D}a$, in other words this observational outcome is to be considered as conclusive proof that object a does not present disposition $\mathcal{D}$.

Nevertheless, it often happens that a researcher does not reject a disposition just because of a single negative experimental outcome. For instance, the researcher may suspect that certain disturbing conditions render the experiment unreliable. From this standpoint, an experiment permits an exception clause, which reductive sentences do not.

This difficulty and that discussed in the next section led Carnap to propose the Two-Level Theory of the Language of Science, which we will outline in section III.3.

III.2.2. The decidability of observational terms.

To clarify the context of the following discussion, it would be helpful to realize that before Carnap's Two-Level Theory, the whole language of science was considered to be a global empiricist language, denoted L_E. This language included all the terms of science, both observable and theoretical ones.

One of the two languages proposed by Carnap is the observational language L_O, which we will discuss in the next section. Let us now turn our attention to the confirmability and refutability in principle of the terms of L_O. We can classify terms definable by means of an observational vocabulary according to their definitions. Observationally decidable terms are introduced by definitions providing explicit criteria for confirming or refuting a property; the remaining terms are observationally undecidable.

From the standpoint of observational decidability, the class of terms of L_O can be divided into:

- Observationally decidable terms, introduced by means of:
  - primitive predicates of L_O, expressing directly observable properties.
  - predicates whose definiens must not contain quantifiers.
- Observationally undecidable terms introduced by means of predicates whose definiens may contain quantifiers with potentially infinite domain:
  - predicates whose definiens contain only universal quantifiers
    non-confirmable, but refutable, in principle;
  - predicates whose definiens contain only existential quantifiers
    non-refutable, but confirmable, in principle;
  - predicates whose definiens contain both kinds of quantifiers
    neither confirmable nor refutable in principle.

III.3. The Two-Level Theory.

III.3.1. Motivations.

Carnap assumed that observationally undecidable terms where not definable in L_O. He then decided that such terms should be introduced into a theoretical language,

[1] This reasoning about bilateral reductive sentences can be extended to n sufficient conditions and r necessary conditions [Ste70].

denoted L_T. Here, as Stegmüller [Ste70] points out, Carnap made a mistake, for he identified L_O-definable with L_O-decidable. But Hempel [Hem65], in a classic example, showed the existence of concepts that are L_O-definable without being L_O-decidable.

This, partly mistaken, conclusion and the one of the last paragraph of section III.2.1, led Carnap to abandon both the idea of a global empiricist language L_E and, hence, the need to introduce dispositional predicates into the language only by means of reductive sentences.

III.3.2. Overview of the Two-Level Theory and of a new version of the hypothetico-deductive method.

Carnap proposed to split the language of science into two languages. One part is the basic empiricist language, understandable by itself, which he called observational language L_O. The other part, called the theoretical language L_T, is the language for formulating theory T. The latter language is not understandable by itself and does not have a complete empirical interpretation. A partial empirical interpretation is obtained by means of a set C of correspondence rules connecting some extralogical expressions of L_T to expressions of the observational language. Some dispositions may be regarded as closer to theoretical concepts than to observational ones; as such, they should be introduced into the theoretical language. Furthermore, by means of the connection between L_O and L_T one can take into account the exception clause needed for a "reasonable" treatment of negative experimental outcomes.

Let M be a dispositional concept, introduced as a theoretical term [Car56;Ste70]. Consider the following hypothetico-deductive scheme.

Main hypothesis:
H_M: a theoretical hypothesis about the presence of M.
Auxiliary hypotheses:
H_K: other theoretical hypotheses,
H_L: some descriptive observational statements.
The use of the underlying theory T.
The correspondence rules C.
The observational consequence is *Oc*.

We consider C as either a set of correspondence rules or their conjunction, according to the context. As in the case of the naive version presented in section II.3.1, we will have the meta-theoretical statement:

III.3.2.1. $H_M \wedge H_K \wedge H_L \wedge T \wedge C \vdash Oc$

Assume now that the expected observational consequence *Oc* does not happen, i.e., $\neg Oc$ holds. From III.3.2.1 one can obtain:

III.3.2.2. $\neg Oc \wedge H_K \wedge H_L \wedge T \wedge C \vdash \neg H_M$

Nevertheless, in contrast to the case of reductive sentences (see end of section III.2.1), even though $\neg Oc$, one may keep the theory T and the correspondence rules C, without having to accept $\neg H_M$. One may instead assume that some theoretical hypotheses of H_K or some descriptive observational statements of H_L are false.

III.3.3. The Observational Language.

L_O is an extensional, completely interpreted language, whose alphabet V_O is the observational vocabulary. Then all the predicates of V_O designate observable properties of events or things. Carnap uses L_O as what he calls the restricted observational language. In such a language we have only directly observable properties. On the contrary, we will use an extended observational language $L_O{}^*$, which allows the introduction of L_O-definable dispositional predicates (see [Car56] section IX and [Ste70]).

Since a theoretical language will have all the freedom needed, some requirements were imposed on an observational language L_O^*:

i) observability for primitive descriptive terms,

ii) reducibility by conditional definitions (e. g., by reduction sentences, as proposed in section III.2.1)[1],

iii) of nominalism: the values of the variables must be concrete, observable entities.

iv) of finitism: the rules of L_O^* do not state or imply that the basic domain is infinite, i.e. L_O^* has at least one finite model.

v) of constructivism: every value of each variable of L_O^* is designated by an expression in L_O^*.

vi) of extensionality: L_O^* contains only truth-functional connectives, no modalities (necessity, possibility, etc.)

III.3.4. The theoretical Language.

The primitive symbols of L_T are divided into logical and descriptive (or extralogical) symbols. Hence the theoretical vocabulary V_T will be the class of all theoretical symbols In general, it is not possible to give explicit definitions for such symbols on the basis of L_O^*.

A theoretical language L_T contains:

i) the usual truth-functional connectives (e.g., negation and conjunction),

ii) other symbols, such as signs of logical modalities (e.g., logical necessity and strict implication),

iii) the domain $\mathcal{U}$ of entities admitted as values of variables in L_T must fulfil the following conventions:

1) $\mathcal{U}$ *includes* a denumerable subdomain of entities,

2) any ordered n-tuple of entities in $\mathcal{U}$(for any finite n) belongs to $\mathcal{U}$,

3) any class of entities in $\mathcal{U}$ *also* belongs to $\mathcal{U}$,

A Theory T in L_T consists of the conjunction of a finite number of postulates formulated in L_T.

As we will see in the next section, some correspondence rules are given, which connect some terms of V_T to those of V_O, in order to provide the partial interpretation of L_T in terms of L_O^*. So, theory T is not an uninterpreted calculus, rather T comes together with a set C of correspondence rules. A scientist will use the partially interpreted theory $T \wedge C$ to guide his espectations by deriving predictions about the behavior of observable objects.

In the previous section we imposed certain restrictive requirements on L_O^*, such as nominalism, finitism. For L_T we will only claim to have an indirect and partial interpretation given by the correspondence rules. Therefore, we are free to choose the logical structure of L_T to suit our needs.

Thus, there is no reason against the acceptance of requirements (i) to (iii) (especially iii.1, iii.2 and iii.3) on L_T, although their acceptance violates requirements (i) to (v) imposed on L_O^* in section III.3.3. Before the introduction of the set C, L_T with T and the corresponding inference rules is an uninterpreted calculus; as such, it does not make sense to apply these requirements to it. After the introduction of C, the only difference is that its rules allow the derivation of certain sentences of L_O^* from certain sentences of L_T or vice versa. It should be noted that we always use the path through L_T to obtain a sentence of L_O^* from certain sentences of L_O^*. Then, both the premises and

[1] Carnap [Car56] proposes here the requirement of explicit definability, since, as we have said, he uses as observational language L_O instead of L_O^*.

the conclusions belong to L_O*, which fulfills the restrictive requirements. Hence, there can be no objection against the use of C and L_T, as far as the meaningfulness of the results of the derivation procedure is concerned.

III.3.5. Correspondence Rules.

As we have said, theory T by itself is uninterpreted. Some symbols of V_T are connected to certain symbols of V_O by means of rules of the set C. This provides a partial interpretation for some symbols of V_T, which is extended to symbols connected to these by means of the postulates of T. Note that this extension is still only a partial interpretation, in general.

Correspondence rules connect sentences of L_O with certain sentences of L_T by means of derivations in either direction. Since we assume that the logical structure of the language $L_T \cup L_O$ is sufficiently rich, we may assume that the rules of C are formulated as axioms. In addition, C can be regarded as the conjunction of such rules. In the following sections we will use the abbreviation C(...) to express the result of applying the rules of C to an object ... belonging either to V_T or to V_O.

IV. THE SOFTWARE PROCESS GOAL REVISITED.

IV.1. The observational indecidibility of the naive version of being-an-engineering-model.

Let us examine the introduction of the disposition being-an-engineering-model within the framework of the Two-Level Theory. Recall reduction sentence III.2.1.2:

$$(\forall\delta)\ (\forall t_O)\ (\ \mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_O \rightarrow (\ m_{pH} \underset{\delta t_O}{\trianglelefteq} A \leftrightarrow ((\exists t)\ (t > t_O \wedge \mathcal{H}m_{pH}\delta t) \wedge \mathcal{I}\delta m_{pH}\ t(\delta)A)))$$

The test conditions and the reactions are expressed in terms of the primitive predicates $\mathcal{B}\delta A$, $\mathcal{A}m_{pH}\delta t_O$, $\mathcal{H}m_{pH}\delta t$ and $\mathcal{I}\delta m_{pH}t(\delta)A)$ of the observational language L_O*. Unfortunately, predicate $((\exists t)\ (t > t_O \wedge \mathcal{H}m_{pH}\delta t) \wedge \mathcal{I}\delta m_{pH}\ t(\delta)A))$ is not L_O-decidable, if the domain of the variable t is infinite.

The L_O-undecidability of this predicate can be easily seen as follows. If machine m_{pH} halts at time t yielding a result $m_{pH}t(\delta)$, one is able to examine whether $\mathcal{I}\delta m_{pH}\ t(\delta)A)$ holds. If, on the other hand, machine m_{pH} has not yet halted at time t, one is unable to know whether or not it would do so at time t+1; and this holds whenever m_{pH} has not halted.

How can we overcome this undecidability? Following Carnap's method, we should introduce predicate $\mathcal{H}m_{pH}\delta t$ in the theoretical language. But, $\mathcal{H}m_{pH}\delta t$ is an example of a predicate inducing the above mentioned confusion between L_O-definability and L_O-decidability. For, $\mathcal{H}m_{pH}\delta t$ is L_O-definable but not L_O-decidable. So, what we should do is to introduce $\mathcal{H}m_{pH}\delta t$ in the observational language and solve the L_O-undecidability by introducing of a theoretical concept related to $\mathcal{H}m_{pH}\delta t$ by means of correspondence rules.

IV.2. Halting of virtual machine and program termination.

Let us take a closer look at the connection between virtual machine m_{pH} and program p inducing it (figure IV.2.1). As any language, L_T has syntax and semantics. A program p is a syntactical object, whose interpretation, in the semantics of L_T, is a function σ. In other words, L_T has a semantic interpretation function v such that $\sigma = v\,[\![p]\!]$. Now, program p, interpreted by "target machine" H, realizes virtual machine m_{pH}, which is clearly an object of L_O. In this context, one may regard the instructions of target machine H, as well as the fetching and decoding mechanism, as constituting a subset H of the set C of correspondence rules. Thus, upon interpreting p, H realizes virtual machine m_{pH}, which computes function $\sigma = v\,[\![p]\!]$.

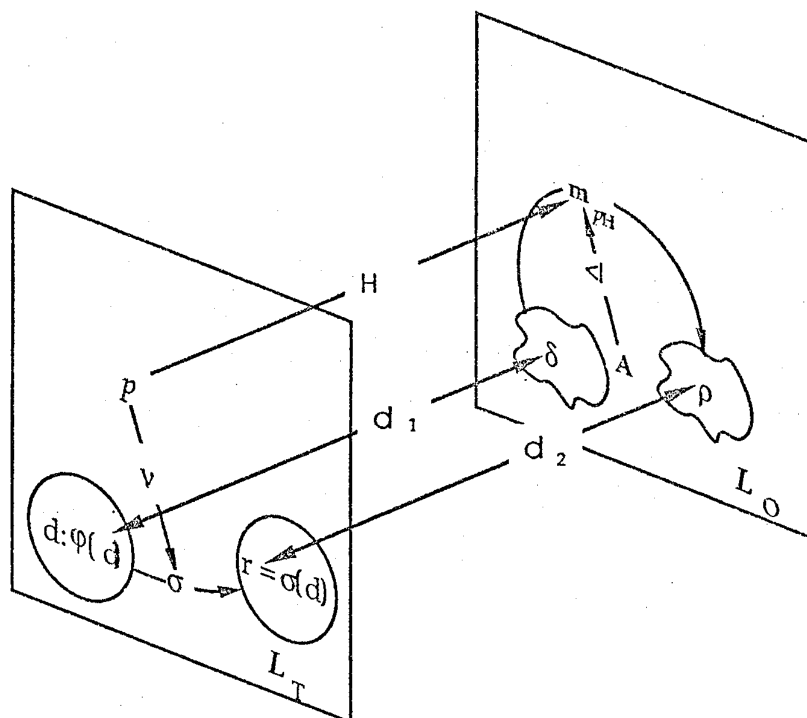

Figure IV.2.1

In the theoretical language we shall say that program p terminates over φ iff the restriction σ' of $\nu[\![p]\!]$ to the set Φ defined by φ is total. More formally, we define the predicate $\mathcal{T}p\varphi$ to mean "program p terminates over φ" as follows:

IV.2.1. Definition. $\mathcal{T}p\varphi \leftrightarrow (\Phi \subseteq \mathcal{D}om(\nu[\![p]\!]))$

where $\mathcal{D}om$ stands for "domain".

The crucial difference between $\mathcal{T}p\varphi$ and $\mathcal{H}\mathrm{m}_{pH}\delta t$ resides in the fact the former is a predicate introduced into the theoretical language, hence amenable to formal proof, whereas the latter is an observational predicate that is not L_O-decidable. Thus, one can consider the syntactical object p and try to prove its termination over φ.

IV.3. Observational and theoretical aspects of being-an-engineering-model.

There are two kinds of objects. On the one hand, we have data δ and results ρ, respectively, of the application domain in the observational language. On the other hand, we have data d and results r in the theoretical language. Their distinction should be kept in mind. The only connection between them is provided by a pair d_1, d_2 of correspondence rules in C involving injective translation functions :

$$f : \{ \delta : \mathcal{B}\delta A\} \rightarrow \mathcal{U} \text{ and } g : \{ \rho : (\exists\delta)\ (\mathcal{B}\delta A \wedge \mathcal{I}\delta\rho A)\} \rightarrow \mathcal{U}$$

where set $\mathcal{U}$ was introduced in section III.3.4. These rules are:

(d_1) $\quad (\forall\delta)\ (\mathcal{B}\delta A \leftrightarrow (\exists d)\ (d = f(\delta) \wedge \delta = f^{-1}(d))$

(d_2) $\quad (\forall\rho)\ [(\exists\delta)\ \mathcal{I}\delta\rho A \leftrightarrow (\exists r)\ (r = g(\rho) \wedge \rho = g^{-1}(r)]$

We will also have correspondence rules of the form:

($C_{\mathcal{B}\varphi}$) $\quad (\forall\delta)\ (\mathcal{B}\delta A \rightarrow \varphi(f(\delta))$

($C_{\mu H}$) $\quad (\forall\delta)\ (\mathcal{B}\delta A \rightarrow g(m_{pH}t(\delta)) = \nu[\![p]\!](f(\delta))$

Call $\mathcal{H}$ the conjunction $d_1 \wedge d_2 \wedge C_{\mathcal{B}\varphi} \wedge C_{\mu H} \wedge H$. Then, predicates $\mathcal{T}p\varphi$, of L_T, and $\mathcal{H}\mathrm{m}_{pH}\delta t$, of $L_O{}^*$, are connected by the derived rule:

(p1) $$\mathcal{H} \vdash \mathcal{T}p\varphi \to (\forall\delta)(\forall t_0)(\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_0 \to (\exists t)(t > t_0 \wedge \mathcal{H}m_{pH}\delta t))$$

We can Skolemize the consequent of p_1 by introducing a function symbol ζ to mean "the instant the machine halts", in the following sense:

(p2) $$(\exists t)(t > t_0 \wedge \mathcal{H}m_{pH}\delta t) \to \zeta(\delta, t_0) > t_0 \wedge \mathcal{H}m_{pH}\delta\zeta(\delta, t_0)$$

Hence, in order to solve our L_0-undecidability problem, we can replace III.2.1.2 and III.2.1.3 by the following stronger version:

IV.3.1. $$\mathcal{H} \vdash (\forall\delta)(\forall t_0)\,[\,\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_0 \wedge \mathcal{T}p\varphi \to$$
$$(\,\zeta(\delta, t_0) > t_0 \wedge \mathcal{H}m_{pH}\delta\zeta(\delta, t_0) \wedge (\mathcal{I}\delta m_{pH}\,\zeta(\delta, t_0)(\delta)A) \to m_{pH} \lhd A))$$

Notice that in IV.3.1 we have eliminated the problem of non-refutability in principle by means of a formal proof of $\mathcal{T}p\varphi$, which ensures that $\mathcal{H}m_{pH}\delta t$ will hold for some finite $t = \zeta(\delta, t_0)$.

Now, if we take regularity for granted, we do not have to worry about the particular time t_0 of the experiment. We can thus eliminate the quantifier $(\forall t_0)$ and simplify IV.3.1 further to :

IV.3.2. $$\mathcal{H} \vdash (\forall\delta)\,(\,\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_0 \wedge \mathcal{T}p\varphi \to$$
$$(\mathcal{I}\delta m_{pH}\,\zeta(\delta, t_0)(\delta)A) \to m_{pH} \lhd A))$$

As mentioned before, Carnap requires the non-deducibility of $(\forall\delta)\,\neg(\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_0 \wedge \mathcal{T}p\varphi)$ from accepted scientific laws. But, what are the "accepted scientific laws" in this case?. Clearly Carnap's requirement concerns the realizability of experimental conditions. Indeed, if $(\forall x)(\neg B_1 x)$ were deducible from accepted scientific laws, then the experimental condition for a reductive sentence like $(\forall x)(B_1 x \to (Dx \leftrightarrow S_1 x))$ would never hold. In our case, $(\forall\delta)\,\neg(\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta t_0 \wedge \mathcal{T}p\varphi)$ should not be deducible from theory T, in which we prove facts about programs (for example $\mathcal{T}p\varphi$), correspondence rules C, application concept A, and program p itself.

IV.4. A new hypothetico-deductive experiment for being-an-engineering-model.

Now we are able to reformulate the naive experiment for being-an-engineering-model, presented in section II.3.1. By applying the scheme introduced in section III.3.2 to reduction sentence IV.3.2, we obtain:

Main hypothesis:
H_M: $m_{pH} \lhd A$.

Auxiliary hypotheses:
H_K: i. $\mathcal{T}p\varphi \leftrightarrow (\Phi \subseteq \mathcal{D}om(v[\![p]\!]))$ (definition IV.2.1).

H_L: i. $\mathcal{B}\delta_i A$, for i=1, . . . , n,
ii. $\mathcal{A}m_{pH}\delta_i t_0$, for i=1, . . . , n.

The use of the underlaying theory T:
Formal proof of $\Phi \subseteq \mathcal{D}om(v[\![p]\!])$

The correspondence rules $\mathcal{H} \wedge p_2$.

Observational consequence:
Oc: $\mathcal{I}\delta_i m_{pH}\,\zeta(\delta, t_0)(\delta_i)A)$, for i=1, . . . , n.

Hence, by instantiating meta-theoretical statement III.3.2.1:

H_M: $m_{pH} \lhd A$

$$\wedge$$

$$H_K: \mathcal{T}p\varphi \leftrightarrow (\Phi \subseteq \mathcal{D}om(v[p]))$$

$$\wedge$$

$$H_L: \bigwedge_{i=1}^{n} \mathcal{B}\delta_i A \wedge \bigwedge_{i=1}^{n} \mathcal{A}m_{pH}\delta_i t_o$$

$$\wedge$$

$$T: \Phi \subseteq \mathcal{D}om(v[p])$$

$$\wedge$$

$$\mathcal{H} \wedge p_2$$

$$\vdash$$

$$Oc: \bigwedge_{i=1}^{n} I\delta_i m_{pH}\, \zeta(\delta, t_o)(\delta_i)A)$$

As we have discussed in section III.3.2, if it is not the case that $I\delta_i m_{pH}\, \zeta(\delta, t_o)(\delta_i)A)$ for some i, we are not forced to reject the main hypothesis $m_{pH} \lhd A$. We can instead doubt the truth of other statement, such as $\mathcal{B}\delta_i A$, i.e. the fact that this δ_i belongs to the domain of A.

On the contrary, if it is the case that $I\delta_i m_{pH}\, \zeta(\delta, t_o)(\delta_i)A)$, for every i from 1 to n, we cannot conclude the main hypothesis $m_{pH} \lhd A$; we can only consider $\bigwedge_{i=1}^{n} I\delta_i m_{pH}\, \zeta(\delta, t_o)(\delta_i)A)$ as a good inductive support for $m_{pH} \lhd A$.

V. THE ALGEBRAIC THEORY OF PROBLEMS.

The concept of problem and a General Theory of Problems were developed from the ideas of G. Polya [Pol57] by P. A. S. Veloso [Vel84]. The goal of this development is a formal tool for reasoning about problem solving and modeling various strategies, techniques, methods, etc.

On the basis of this General Theory of Problems an Algebraic Theory of Problems [Hae87] was developed, aiming at a tool for the formal treatment of the software development process at various levels. These levels range from the purely epistemological one of process explication [Hae89], through those of prescribing different process obligations [Vel89] and modeling programming methods [Zar88], to a calculus for program derivation [Elu88, 89; Hae89; Vaz89].

In the sequel we will outline a version of the Algebraic Theory of Problems (called Theory of Irestricted Problems, TIP), constructed as an extension of the Zermelo-Fraenkel set theory. We will concentrate on the aspects relevant to our present purposes. For more details, see [Var89; Vaz89; Elu89].

V.1 The concept of problem and solution.

A *problem* over the universe $\mathcal{U}$ is a 3-tuple $P = \langle D, R, q \rangle$ where D and R are subsets of $\mathcal{U}$ and $q \subseteq D \times R$. This mathematical structure attempts to capture the essence of the three classical questions suggested by Polya in approaching a problem: *what are the data? what are the results?* and *what is the problem condition?* Hence, in the 3-tuple $P = \langle D, R, q \rangle$, D stands for the *data domain*, R for the *result domain* and q for the problem *condition*.

We will denote by **P** the set of all problems P with data and results domains included in $\mathcal{U}$.

We say that problem $P = \langle D, R, q \rangle$ is *viable*, denoted by $\mathcal{V}ib P$, iff for every data in the data domain there exists a result in the result domain related with it by the condition q. Formally:

V.1.1. Definition. $\mathcal{V}ib P \leftrightarrow (\forall d)\ (d \in D \rightarrow (\exists r)\ (r \in R \wedge q(d, r)))$

Note that $\mathcal{V}ib\mathrm{P} \leftrightarrow \mathcal{D}om(\mathrm{q}) = \mathrm{D}$[1].

This representation of a problem captures the idea of choice associated with obtaining an acceptable result for each given data in a stated, but still unsolved, problem. Then, a solution should be an object that eliminates this choice and, therefore, solving a problem should mean constructing such an object. Now, in this context what kind of object could a solution for a problem be? An answer suggests itself: a Skolem function for its condition. To be condemned to the eternal activity of choosing results for each given data is completely different from to construct a function to do such a choice once and for all. Then:

V.1.2. Definition. $\sigma \leftarrow \mathrm{P} \leftrightarrow \sigma \in \mathrm{R}^{\mathrm{D}} \wedge (\forall d)\,(d \in \mathrm{D} \rightarrow q(d, \sigma(d)))$

The set Ω_P of all the solutions of P will be called the *solution space* of P[2]. We say that problem P is *solvable*, denoted by $\mathcal{S}ol\mathrm{P}$, iff P has some solution, i.e. $\Omega_P \neq \emptyset$. Therefore, the Axiom of Choice yields the following:

V.1.3. Theorem. $\mathcal{V}ib\mathrm{P} \leftrightarrow \mathcal{S}ol\mathrm{P}$

We said that two problems P and Q are equal iff are equal as 3-tuples.

We will introduce now the notion of disjointeness, which we will denote by $\mathrm{P} \,\square\, \mathrm{Q}$.

V.1.4. Definition. $\mathrm{P} \,\square\, \mathrm{Q} \leftrightarrow \mathrm{D}_P \cap \mathrm{D}_Q = \emptyset$

V.2 Relaxation of problems.

Transfer of solutions from one problem to another one is an important relationship between them, which we call relaxation, denoted by $\lrcorner$.

V.2.1. Definition. $\mathrm{P} \lrcorner \mathrm{Q} \leftrightarrow \mathrm{D}_Q \subseteq \mathrm{D}_P \wedge q_P|_{\mathrm{D}_Q} \subseteq q_Q$

The following theorem holds:

V.2.2. Theorem. $\mathrm{P} \lrcorner \mathrm{Q} \rightarrow (\sigma \leftarrow \mathrm{P} \rightarrow \sigma \leftarrow \mathrm{Q})$

Clearly, relaxation is a preorder on $\mathbf{P}$.

V.3. Operations on problems.

The introduction of the operations on problems has been motivated by the aim of modeling problem-solving strategies, such as reduction, decomposition, specialization, etc. These operations together with the set of problems over the universe $\mathcal{U}$ constitute the algebra of problems. We will introduce here only the main operations, namely sum, product and direct product.

---

[1] In a fully extensional and reductionistic way, one might view a problem as merely its condition q. Thus, the theory of problems would amount to the theory of binary relations. Why, then, did Polya suggest three questions instead of only the third one? In other words: why have we introduced the concept of problem as a 3-tuple? To understand this, one must realize that a problem is an intensional object rather than just an extensional one. Assume one is given a problem $\mathrm{P} = \langle \mathrm{D}, \mathrm{R}, q \rangle$; usually, having an expression for q do not means does not imply that one also knows by comprehension its domain $\mathcal{D}om(\mathrm{q})$ and its range $\mathcal{R}an(\mathrm{q})$. So, in stating a problem, we must state its data and result domains, which are part of one's knowledge about P. Anyway, establishing viability is at least as difficult as determining $\mathcal{D}om(\mathrm{q})$. But defining something must not imply deciding it, i.e., with D and R one can discuss viability, even though one may have a hard time in establishing it. Furthermore, the order in which Polya stated his three questions is not irrelevant. In the context of the Two-level Theory, we will use problems as theoretical counterparts of application concepts. Then, viability is the theoretical counterpart of an important observational concept, which is unfortunately L_0-undecidable.

[2] We use subindex P to suggest that P is the problem we are referring to. So, D_P is the data domain of problem P, R_P is its result domain, etc.

V.3.1. Sum of problems. The motivation for the definition of sum, +, is to model the decomposition of a problem induced by a partition of its data domain.

V.3.1.1. Definition. $P + Q = \langle D_P \cup D_Q, R_P \cup R_Q, q_P \cup q_Q \rangle$

Hence, we can state:

V.3.1.2. Theorem.

i. associativity: $(P + Q) + R = P + (Q + R)$

ii. commutativity: $P + Q = Q + P$

iii. idempotence: $P + P = P$

iv. neuter: $P + 0 = P$ where $0 = \langle \varnothing, \varnothing, \varnothing \rangle$

V.3.1.3. Theorem. $\mathcal{V}i\mathcal{b}P \wedge \mathcal{V}i\mathcal{b}Q \rightarrow \mathcal{V}i\mathcal{b}(P + Q)$

We can now extend the concept of sum to that of summation over a set P of problems.

V.3.1.4. Definition. $\sum_{x \in P} x = \langle \bigcup_{x \in P} D_x, \bigcup_{x \in P} R_x, \bigcup_{x \in P} q_x \rangle$

V.3.2. Additive subproblems. We will now define the relation, $\Subset$, of being an additive subproblem.

V.3.2.1. Definition. $P \Subset Q \leftrightarrow (\exists R)(R \in P \wedge P + R = Q)$

V.3.2.2. Theorem. The following statements are equivalent:

i. $P \Subset Q$

ii. $P + Q = Q$

iii. $D_P \subseteq D_Q \wedge R_P \subseteq R_Q \wedge q_P \subseteq q_Q$

V.3.2.3. Theorem. $\langle P, \Subset \rangle$ is a complete partial order.

V.3.2.4. Definition. $P^+ = \{ Q : Q \in P \wedge Q \Subset P \}$

Then, P^+ is the set of all the additive subproblems of P.

V.3.2.3. Theorem. i. $\langle P^+, \Subset \rangle$ is an upper semilattice,

ii. P is the **lub** of this semilattice,

iii. $\sum_{x \in P^+} x = P$

V.3.3. Complete additive subproblems. An important property is transfer of solutions. In general, an arbitrary subproblem of P does not have this property. It would be interesting to study those subproblems of P that do have this property; these are the so called complete additive subproblems of P.

V.3.3.1. Definition. $P \Subset_C Q \leftrightarrow P \Subset Q \wedge P \lrcorner Q$

where the predicate $P \Subset_C Q$ means "P is a complete additive subproblem of Q"

V.3.3.2. Theorem. $P \Subset_C Q \leftrightarrow R_P \subseteq R_Q \wedge q_P \subseteq q_Q \wedge D_P = D_Q$

V.3.4. Product of problems. The motivation for the definition of the product, $\bullet$, is to model the problem decomposition induced by the interpolation into its condition of an intermediate data domain. Here we will employ the usual notation $\mathcal{R}/\mathcal{S}$ for the relative product of two relations $\mathcal{R}$ and $\mathcal{S}$, and $\mathcal{R}^{1}$ for the inverse of $\mathcal{R}$.

V.3.4.1. Definition. $P \bullet Q = \langle D_P, R_Q, q_P/q_Q \rangle$

Hence, we can state:

V.3.4.2. Theorem.

i. associativity: $(P \bullet Q) \bullet R = P \bullet (Q \bullet R)$

ii. left distributivity over sum:

$$P \circ (P + Q) = P \circ Q + P \circ R$$

iii. right distributivity over sum:

$$(P + Q) \circ P = Q \circ P + R \circ P)$$

V.3.4.3. Definition. $\mathit{Coup}P\ Q \leftrightarrow R_P \subseteq D_Q$[1]

V.3.4.4. Theorem. $\mathcal{V}ib P \wedge \mathcal{V}ib Q \wedge \mathit{Coup}P\ Q \rightarrow \mathcal{V}ib(P \circ Q)$

V.3.4.5. Definition. $1^P = \langle R_P, R_P, \Im^P \rangle$

where $\Im^P$ is the identity on R_P.

V.3.4.6. Definition. $1_P = \langle D_P, D_P, \Im_P \rangle$

where $\Im_P$ is the identity on D_P.

V.3.4.7. Definition. $P^{\circ 0} = 1_P$

$$P^{\circ n+1} = P^{\circ n} \circ P$$

V.3.4.8. Definition. $P^{*} = \sum_{x \in \{P^{\circ n} : n \in N\}} x$

V.3.5. Direct product of problems. The motivation for the definition of the operation direct product, ×, is to model the problem decomposition induced by the inner structure of each data of its data domain.

V.3.5.1. Definition. $P \times Q = \langle D_P \times D_Q, R_P \times R_Q, q_{P \times Q} \rangle$

where $q_{P \times Q} = \{ \langle\langle d, r\rangle, \langle d', r'\rangle\rangle : \langle d, r\rangle \in q_P \wedge \langle d', r'\rangle \in q_Q \}$.

Hence, we can state:

V.3.5.2. Theorem.

i. left distributivity over sum:

$$P \times (Q + R) = P \times Q + P \times R$$

ii. right distributivity over sum:

$$(Q + R) \times P = Q \times P + R \times P$$

iii. $P \times 0 = 0 \times P = 0$ where $0 = \langle \emptyset, \emptyset, \emptyset \rangle$

iv. $P \circ Q \times R \circ S = (P \times R) \circ (Q \times S)$

v. $(P \times Q) + (R \times S) \not\subseteq (P + R) \times (Q + S)$

V.3.5.3. Definition. $P^{\times 1} = P$

$$P^{\times n+1} = P^{\times n} \times P$$

V.3.5.6. Definition. $P^{\dagger} = \sum_{x \in \{P^{\times n} : n \in \{N - \{0\}\}} x$

V.4. Operations on problem solutions.

V.4.1. Sum of solutions.

V.4.1.1. Definition. Let $\sigma: A \rightarrow B$ and $\sigma': A' \rightarrow B'$, be a pair of functions and $K \subseteq A \cap A'$. We define $\sigma \oplus_K \sigma': A \cup A' \rightarrow B \cup B'$ by:

$$\sigma \oplus_K \sigma'(d) = \begin{cases} \sigma(d) & \text{if } d \in (A - A') \cup K \\ \sigma'(d) & \text{if } d \in A' - K \end{cases}$$

V.4.1.2. Theorem. $\sigma \leftarrow P \wedge \sigma' \leftarrow Q \wedge K \subseteq D_P \cap D_Q \rightarrow (\sigma \oplus_K \sigma' \leftarrow P + Q)$

---

[1] A less restrictive version of this condition can be formulated [Vaz89].

Note that in defining the sum of σ and σ' we may choose any K, since all are in principle equally suitable. Hence, if we denote by ~K the set $((\mathcal{Dom}(\sigma) \cap \mathcal{Dom}(\sigma')) - K)$ we can state

V.4.1.3. Theorem.

i. commutativity: $\sigma \oplus_K \sigma' = \sigma' \oplus_{\sim K} \sigma$

ii. idempotence: $\sigma \oplus_K \sigma = \sigma$

iii. neuter: $\sigma \oplus_K \varnothing = \sigma$ where $\varnothing$ is the empty function

V.4.1.4. Theorem.

$$\sigma \leftarrow P + Q \rightarrow (\exists\tau)(\exists\rho)(\exists K)\,(\tau \leftarrow P \wedge \rho \leftarrow Q \wedge K \subseteq D_P \cap D_Q \wedge \sigma = \tau \oplus_K \rho)$$

V.4.1.5. Corollary.

$$\Omega_{P+Q} = \Omega_P \oplus \Omega_Q, \text{ where } \Omega_P \oplus \Omega_Q = \{\sigma \oplus_K \sigma' : \sigma \in \Omega_P \wedge \sigma' \in \Omega_Q \wedge K \subseteq D_P \cap D_Q\}$$

If $A \cap A' = \varnothing$ then K is irrelevant. Thus:

V.4.1.6. Definition. $\mathcal{Dom}(\sigma) \cap \mathcal{Dom}(\sigma') = \varnothing \rightarrow \sigma \oplus \sigma' = \sigma \oplus_\varnothing \sigma'$

V.4.1.7. Theorem. $P \,\square\, Q \rightarrow (\sigma \leftarrow P \wedge \sigma' \leftarrow Q \rightarrow \sigma \oplus \sigma' \leftarrow P + Q)$

V.4.2. Product of problem solutions.

We will introduce now the function symbol $\circledast$ for the product of functions:

V.4.2.1. Definition. $\sigma \circledast \sigma' = \sigma' \circ \sigma$

where $\circ$ is the usual function composition, i. e. $\sigma \circledast \sigma'(d) = \sigma'(\sigma(d))$.

V.4.2.2. Theorem. $\mathit{Coup}P\ Q \rightarrow (\sigma \leftarrow P \wedge \sigma' \leftarrow Q \rightarrow \sigma \circledast \sigma' \leftarrow P \bullet Q)$

V.4.2.3. Corollary. $\mathit{Coup}P\ Q \rightarrow (\Omega_P \circledast \Omega_Q \subseteq \Omega_{P \bullet Q})$

where $\Omega_P \circledast \Omega_Q = \{\sigma \circledast \sigma' : \sigma \in \Omega_P \wedge \sigma' \in \Omega_Q\}$

V.4.3. Direct product of problem solutions.

V.4.3.1. Definition . $\sigma \otimes \sigma' : A \times A' \rightarrow B \times B';\ \sigma \otimes \sigma'(d, d') = \langle \sigma(d), \sigma'(d') \rangle$

V.4.3.2. Theorem. $\sigma \leftarrow P \wedge \sigma' \leftarrow Q \rightarrow \sigma \otimes \sigma' \leftarrow P \times Q$

V.4.3.3. Theorem. $\sigma \leftarrow P \times Q \rightarrow (\exists\tau)(\exists\rho)\,(\tau \leftarrow P \wedge \rho \leftarrow Q \wedge \sigma = \tau \otimes \rho)$

V.4.3.4. Corollary. $\Omega_{P \times Q} = \Omega_P \otimes \Omega_Q$

where $\Omega_P \otimes \Omega_Q = \{\sigma \otimes \sigma' : \sigma \in \Omega_P \wedge \sigma' \in \Omega_Q\}$

V.5. Relaxation and the operations on problems.

As we will see in section VI.2, relaxation is an outstanding relation between problems, so we will analyze closely its properties with respect to the operations defined in the previous sections.

V.5.1. Theorem. $P \,\square\, Q \rightarrow (P \lrcorner Q \wedge R \lrcorner S \rightarrow (P + R) \lrcorner (Q + S))$

V.5.2. Theorem. $\mathit{Coup}Q\ S \wedge \mathit{Coup}P\ R \rightarrow (P \lrcorner Q \wedge R \lrcorner S \rightarrow (P \bullet R) \lrcorner (Q \bullet S))$

V.5.3. Theorem. $P \lrcorner R \wedge Q \lrcorner S \rightarrow (P \times R) \lrcorner (Q \times S)))$

V.6. Reduction and abstraction.

The notion of reduction is well known in problem solving. G. Polya [Pol57] suggest the following two questions as an strategy for the solution of a problem, viz. "do you know a related problem?", "do you know a problem with the same or a similar unknown?" This strategy is often used in various realms: geometry, graphs, automata, program derivation, etc. This notion has been studied in the contexts of the General Theory of Problems [Vel84] and of the Algebraic Theory of Problems [Hae87; Vaz89]. In this latter context we now introduce the predicate $P \leftarrow R_{RS}\ Q$ to mean "problem P is a reduction of problem Q via problems R and S".

V.6.1. Definition. $P \leftarrow R_{RS}\ Q \leftrightarrow \mathit{Coup}R\ P \wedge \mathit{Coup}P\ S \wedge R \bullet P \bullet S \lrcorner Q$

Hence, we can state the following:

V.6.2. Theorem. $P_{\leftarrow}R_{RS}\, Q \rightarrow (\sigma \leftarrow P \wedge \tau \leftarrow R \wedge \rho \leftarrow S \rightarrow (\tau \circledast \sigma \circledast \rho) \leftarrow Q))$

Versions of this strategy are widely used in program derivation under names like generalization [Par88]; for its use in program derivation by means of a problem-theoretic derivation calculus, see [Vaz89] .

We will now introduce the fundamental concept of abstraction. Towards this goal, we will define the predicate $P_{\leftarrow}P_R\, Q$ to mean "problem P is a *relevant part* of problem Q with respect to problem R " .

V.6.3. Definition. $P_{\leftarrow}P_R\, Q \leftrightarrow (R \times P = Q \vee P \times R = Q)$

This concept of relevant part of a problem reflects the fact that one usually wants to forget those "attributes" of the data objects that one considers irrelevant in some intensional sense, say, because such a parts constitute trivial problems. One should note here that this concept is an oversimplification, in the sense that we are talking about irrelevant parts of data objects up to the "fine structure" of the relations among them. We simplify those relations by replacing them by a cartesian product including them. Notice that the relation ${}_{\leftarrow}P$ may be considered as an abstraction relation, since the decomposition $R \times P$ (or its symmetric $P \times R$) can be performed if the elements of the data domain of Q are pairs $\langle x, y \rangle$; then, abstraction accounts for the fact that x (or y in the case $P \times R$) is irrelevant (in some intensional sense) for the solution of Q.

Now, how can we overcome the difficulty arising when the objects of the data domain of Q are not pairs? Simply by reduction. So, we can introduce the concept of abstraction as follows.

V.6.4. Definition. $P_{\leftarrow}\mathcal{A}_{RTS}\, Q \leftrightarrow (\exists P_1)\ (P_1 \in P \wedge P_{1\leftarrow}R_{TS}\, Q \wedge P_{\leftarrow}P_R\, Q))$

where the predicate $P_{\leftarrow}\mathcal{A}_{RTS}\, Q$ stands for " P is an abstraction of Q with respect to R via T and S".

If we analyze definition V.6.1 we will observe that the operation $R \circ P \circ S$ can be replaced by the simpler one $P \circ S$ in the case that $R = 1_P$ or by $R \circ P$ in the case that $S = 1^P$ or by P if it is the case $R = 1_P \wedge S = 1^P$. A similar reasoning applies to predicate $P_{\leftarrow}\mathcal{A}_{RTS}\, Q$.

VI. OBSERVATIONAL AND THEORETICAL OBJECTS IN THE SOFTWARE PROCESS.

VI.1. The observational level for the software process.

The observational level for the software process will comprise application concepts A -with data δ and results ρ belonging to their domains- and virtual machines m_{pH}. In addition, we also have in this level a denumerable ordered set called *time*. So, statements such as $m_{pH} \lhd A$, $\mathcal{B}\delta A$, $\mathcal{A}m_{pH}\delta t_0$, $t > t_0$ and $\mathcal{H}m_{pH}\delta t$ belong to $L_O{}^*$.

Since a virtual machine is the result of interpreting a program by a target machine, and because of some facts to be clarified in the next section, there are denumerably many virtual machines related to an application concept A by means of the relation $\lhd$ (see figure VI.1.1).

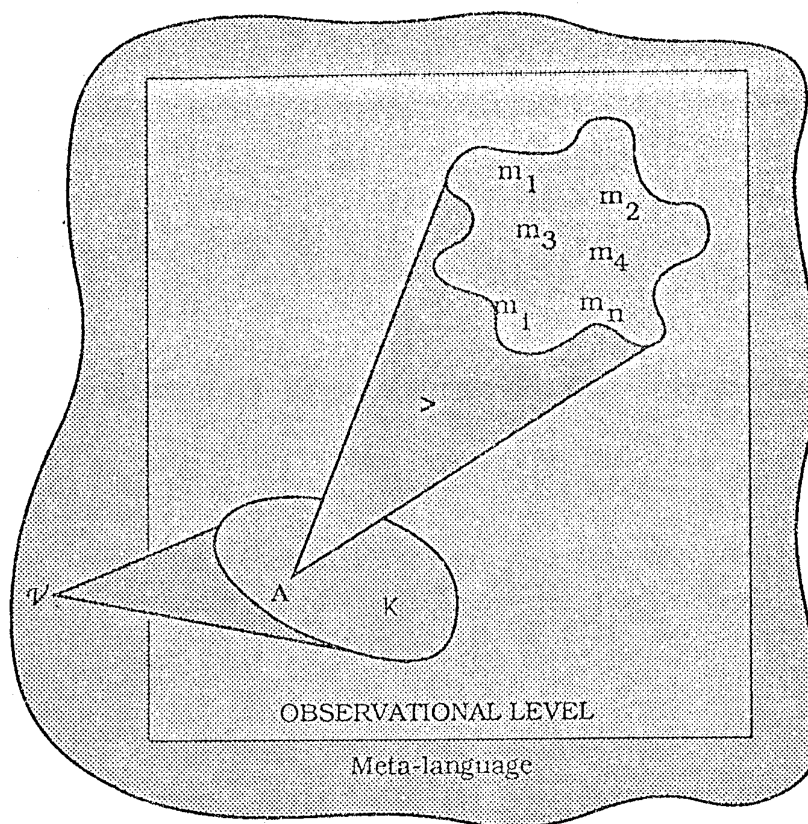

Figure VI.1.1.

Notice that, in contrast to the theoretical level, objects of L_0 have no formal denotations. Their meaning is given directly by observation, in view of requirements (i) of observabilty and (ii) of reducibility imposed on L_0 in section III.3.3.

At this point one should recall that, as stated in convention II.1.1, an application concept A stands here for the extension of a *real problem* and not necessarily for its description by comprehension, i. e. by means of a property. In some rare cases the application concept has a rather simple extension or can be described by comprehension in the observational language. Otherwise, the starting point of the software development process is an application verbalization $\mathcal{V}_A$, which as stated in convention II.1.2, is a sentence of the meta-language.

So, if, as it often happens, the meta-language is ambiguous, symbol $\mathcal{V}_A$ is not quite correct. For, an ambiguous verbalization $\mathcal{V}$ describes not exactly one but a class of application concepts $K_{\mathcal{V}}$.

VI.2. The theoretical level for the software process.

The theoretical level will include, in addition to the various objects allowed by conditions (i) to (iv) of section III.3.4, problems , functions, formal specifications and programs. Then, the Algebraic Theory of Problems, **ZF**, the Fixpoint Theory of programs, etc., will all be part of the theoretical level.

We must distinguish on this level between syntactic objects and their semantical denotations. Thus, a formal specification *Spc* is a syntactic object whose denotation, by means of a semantic interpretation function μ, is a problem, denoted $\mu[\![Spc]\!]$. The

usual idea that a deterministic program p denotes a recursive function σ is captured by another semantic interpretation function ν; so we write $\sigma=\nu[\![p]\!]$[1].

We introduce here the following general notation, which will be used in various places.

VI.2.1. Notation. Consider a subset S of V_T, a distinguished set Xof symbols belonging to V_T but not to S, and a set O of operation symbols over V_T. By the *closure* C(S, X, O) we mean the set of terms built from $S \cup X$ by means of the operation symbols of O.

VI.2.1. Specifications, problems, programs and solutions.

First of all, we will define the language in which we will write specifications and programs, i.e., syntactic descriptions of problems and functions.

VI.2.1.1. Definition. By *global programming language* we mean the language

$$\mathcal{L}_G = C(\{\mathcal{F}: \mu[\![\mathcal{F}]\!] \in \mathbb{P}, \varnothing, \{\alpha, \pi, \xi, *_\pi, *_\xi\})$$

where the symbols $\alpha, \pi, \xi, *_\pi$ and $*_\xi$, are interpreted through μ as the operations on problems $+, \circ, \times, *$ and $\dagger$, respectively.

So, we must bear in mind that theorems and definitions in the subsequent discussion refer to this global programming language $\mathcal{L}_G$.

At this point we can define the notion of correctness of programs with respect to formal specifications. Thus, we can introduce $p \sqsubseteq Spc$ to stand for "program p is partially correct with respect to the specification Spc".

VI.2.1.2. Definition. $p \sqsubseteq Spc \leftrightarrow \mu[\![p]\!] \lrcorner\ \mu[\![Spc]\!]$

We shall denote total correctness by means of the predicate $\sqsubset$.

VI.2.1.3. Definition. $p \sqsubset Spc \leftrightarrow \mathit{Vib}\,\mu[\![p]\!] \wedge p \sqsubseteq Spc$

In the above definition termination is guaranteed by the viability of $\mu[\![p]\!]$. As we will see in the next section, this definition of correctness includes the case of non-deterministic programs.

Some well known results about these concepts are expressed in our formalism as follows.

VI.2.1.4. Theorem. $\mathcal{T}p\varphi \wedge p \sqsubset Spc \rightarrow (D_{\mu[\![Spc]\!]} \subseteq \mu[\![\varphi]\!])$

VI.2.1.5. Theorem. $\mathcal{T}p\varphi \wedge (D_{\mu[\![Spc]\!]} \subseteq \mu[\![\varphi]\!]) \wedge p \sqsubseteq Spc \rightarrow p \sqsubset Spc$

Let us now introduce the concepts of functional problem and deterministic program.

VI.2.1.6. Definition. P *is functional* $\leftrightarrow \mathit{Vib}\mathrm{P} \wedge (\forall \mathrm{Q})(\mathit{Vib}\mathrm{Q} \wedge \mathrm{Q} \not\sqsubseteq_C \mathrm{P} \rightarrow \mathrm{Q} = \mathrm{P})$

VI.2.1.7. Definition. p *is deterministic* $\leftrightarrow \mu[\![p]\!]$ *is functional*

Now we can state the following theorem whose proof can be derived from theorem V.2.2:

VI.2.1.8. Theorem. p *is deterministic* $\wedge\ p \sqsubset Spc \leftrightarrow (\nu[\![p]\!] \leftarrow \mu[\![Spc]\!])$

---

[1] Note that if Spc and p are written in the same language (which is a condition always satisfiable, as we will see shortly) and p is deterministic, then problem $\mu[\![p]\!]$ will have as its condition the functional relation $\nu[\![p]\!]$.

Note that both semantic interpretation functions μ and ν are non-injective. Also, there are infinitely many programs correct with respect to a given specification. This situation (in the case of deterministic programs) is depicted in figure VI.2.1.1. Note that in this figure $P_{i,j} \sqsubseteq Spc_k$ holds for every i, j and k.

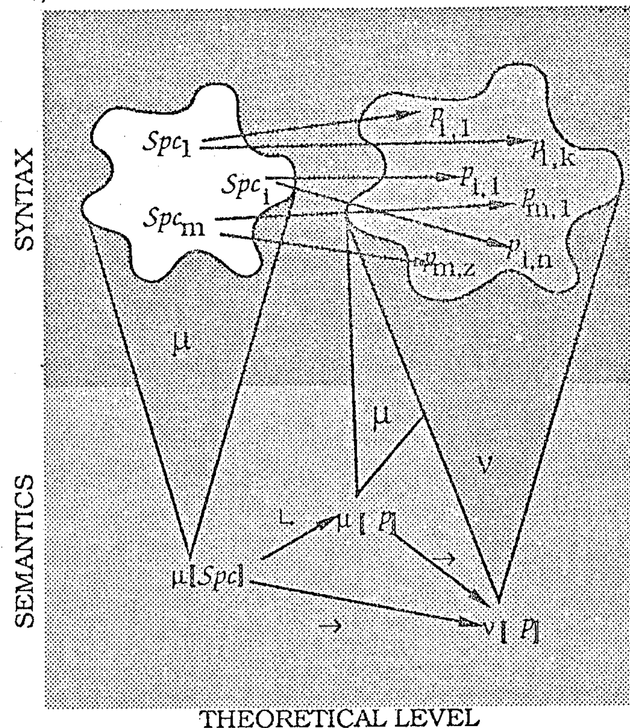

Figure VI.2.1.1

VI.2.2. Target machines and wide-sense solutions.

One may have noticed that the Theory of Problems presented in section V is mainly extensional. In particular, the definition of solution as a Skolem function is also an extensional, and consequently reductionistic, concept. Let us now analyze the activity of "solving" the problem denoted by a specification in the context of software development

If we consider specifications as syntactic objects, interpreted by function μ, and programs as syntactic objects, interpreted by function[1] ν, the distinction between them is clear. But this is an oversimplification for two reasons. First, it fails to contemplate nondeterministic or relational programs. Second, one considers a *real problem* as solved when one has a program for it. In other words, a program that, upon interpretation by the target machine, constitutes a virtual machine that is an engineering model of the application concept

Therefore, the difference between specifications and programs seems to be intensional, and relative to the target machine,i.e., referred to the subset H of the correspondence rules. At this point we can introduce the concept of an easy problem.

VI.2.2.1. Definition. A constant symbol f, denoting a problem $\mu[\![f]\!]$, is said to be *easy* with respect to target machine H, denoted by $\phi_H(f)$, iff f appears in some rule of the set H.

Target machine H is able to choose for every data δ, whose interpretation d=f(δ) (by means of a correspondence rule such as (d_1)) belongs to $D_{\mu[\![f]\!]}$, a result ρ, whose

[1] In other words, if we accept that a specification describes a relation while a program describes a recursive function.

interpretation $r=g(\rho)$ (by means of a correspondence rule such as (d_2)) belongs to $R_{\mu[\![f]\!]}$, so that $\langle d, r\rangle \in q_{\mu[\![f]\!]}$.

We can now introduce the concept of target programming language.

VI.2.2.2. Definition. By the *target programing language* of target machine H we mean the following sublanguage of $\mathcal{L}_G$:

$$\mathcal{L}_H = C\,(\{f: \mu[\![f]\!] \in \mathbb{P} \wedge \phi_H(f)\}, \varnothing, \{\alpha, \pi, \xi, *_\pi, *_\xi\})$$

where the operation symbols $\alpha, \pi, \xi, *_\pi$ and $*_\xi$ also appear in rules of H.

The idea behind this definition is: the target programing language is the sublanguage of $\mathcal{L}_G$ directly supported by target machine H, in the sense that its symbols can be interpreted by the correspondence rules belonging to H[1]. H is able to replace each one of the function symbols of the set $\{\alpha, \pi, \xi, *_\pi, *_\xi\}$ that appears in a formula of $\mathcal{L}_H$ by an operation that reflects the corresponding function symbol on solutions.

We will now introduce a concept of solution in an intensional sense.

VI.2.2.3. Definition. Given formulas $\mathcal{F}$ and $\mathcal{F}'$ of $\mathcal{L}_G$, we say that $\mathcal{F}'$ is a *wide-sense solution of* $\mathcal{F}$ with respect to target machine H, denoted $\mathcal{F}' \underset{H}{\Leftarrow} \mathcal{F}$, iff $\mathcal{F}'$ is totally correct with respect to $\mathcal{F}$ and belongs to $\mathcal{L}_H$. More formally:

$$\mathcal{F}' \underset{H}{\Leftarrow} \mathcal{F} \leftrightarrow \mathcal{F}' \sqsubseteq \mathcal{F} \wedge \mathcal{F}' \in \mathcal{L}_H$$

Hence, the virtual machine $m_{\mathcal{F}H}$ resulting of the interpretation of formula $\mathcal{F}'$ by means of the set H of rules will be able to choose for every data δ, whose interpretation $d=f(\delta)$ belongs to $D_{\mu[\![\mathcal{F}']\!]}$, a result ρ, whose interpretation $r=g(\rho)$ belongs to $R_{\mu[\![\mathcal{F}']\!]}$, so that $\langle d, r\rangle \in q_{\mu[\![\mathcal{F}']\!]}$.

VI.2.2.4. Theorem. $\mathcal{F}' \underset{H}{\Leftarrow} \mathcal{F} \rightarrow (\forall\sigma)(\sigma \leftarrow \mu[\![\mathcal{F}']\!] \rightarrow \sigma \leftarrow \mu[\![\mathcal{F}]\!])$

This result relates the intensional concept of wide-sense solution with respect to a target machine to the extensional one of solution. As we will see in the next sections, it allows the use of the extensional version of the Algebraic Theory of Problems in reasoning about the software process.

We will now introduce a generalization of the concept of abstraction proposed in definition V.6.4. This generalization resides in replacing problems R, S and T, used in section V.6, by a term. This term, belonging to language $\mathcal{L} = C\,(\{f: \mu[\![f]\!] \in \mathbb{P}, \{x\}, \{\alpha, \pi, \xi\}\}) \subseteq \mathcal{L}_G$, has a free variable x to represent the result of the abstraction.

VI.2.2.5. Definition. We say that formula $\mathcal{F}$ is a *general abstraction*, or simply an *abstraction*, of problem P via term $\mathcal{T}(x)$, denoted $\mathcal{F} \leftarrow A_{\mathcal{T}(x)}$ P, iff term $\mathcal{T}(x)$ belongs to $\mathcal{L}$ and its denotation is a relaxation of P. More formally:

$$\mathcal{F} \leftarrow A_{\mathcal{T}(x)}\, P \leftrightarrow (\mathcal{T}(x) \in \mathcal{L} \wedge \mu[\![\mathcal{T}(\mathcal{F})]\!] \sqsupseteq P)$$

With this definition, viable problems may have abstractions whose denotations are non-viable problems. If we want to eliminate this possibility of *vacuous* abstractions, we must require viability of $\mu[\![\mathcal{T}(\mathcal{F})]\!]$.

---

[1] This is a precise version of a machine "supporting" extra-logical axioms [Leh84].

This concept of abstraction reflects the idea of the method of decomposition by abstraction, much as exploited in [Elu89;Vaz89]. Note that this concept connects semantical and syntactic levels of $\mathcal{L}_G$.

In the sequel we will assume that term $\mathcal{T}(\mathcal{F})$ of $\mathcal{L}$ belongs to $\mathcal{L}_H$. This assumption amounts to the intension that what is abstracted away should be easy. We must bear in mind that easiness is a relative concept with respect to either target machine H or the application domain. The former is a well defined concept. For the latter we will not introduce a formal definition, restricting ourselves to some comments in next section.

In both cases of easiness, one is not concerned with solving the "environmental" part of term $\mathcal{T}(\mathcal{F})$ but only with $\mathcal{F}$ itself. For the sake of simplicity, we will restrict the following discussion to the case in which problems of this environmental part are easy with respect to the target machine.

We can now state a simple but important transitivity result.

VI.2.2.6. Theorem. $\mathcal{F}' \sqsubseteq \mathcal{F}'' \wedge \mathcal{F}'' \leftarrow A_{\mathcal{T}(x)}\, \mu[\![\mathcal{F}]\!] \rightarrow \mathcal{F}' \leftarrow A_{\mathcal{T}(x)}\, \mu[\![\mathcal{F}]\!]$

Then, we introduce a substitutivity result relating abstraction and wide-sense solution. This is an important result for the analysis of the connection between application concept and specification, carried out in the next section.

VI.2.2.7. Theorem. $\mathcal{F}' \underset{H}{\Leftarrow} \mathcal{F}'' \wedge \mathcal{F}'' \leftarrow A_{\mathcal{T}(x)}\, \mu[\![\mathcal{F}]\!] \rightarrow \mathcal{T}(\mathcal{F}') \underset{H}{\Leftarrow} \mathcal{F}$

VI.3. Application concept, specifications, programs and virtual machines.

In order to analyze the connections among application concept, specifications, programs, and virtual machines, we need a way to talk formally about the application concept. Although problems are theoretical objects, in figure IV.2.1 we have associated, in a purposely vague manner, the application concept with the idea of problem, in that we associate data and result domains to it. In our framework, application concepts are objects of $L_O{}^*$ whereas problems belong to L_T. In such a context, any attempt to treat applications concepts as problems must be based on an explicit connection.

VI.3.1. Convention. We say that an object o of $L_O{}^*$ is *coextensive* with an object w of L_T, denoted by $o \equiv w$, iff their extensions are *congruent* up to the correspondence rules C, i.e., $w = C(o)$.

VI.3.2. Axiom. Every application concept A in $L_O{}^*$ is coextensive with a problem C(A) in L_T.

This axiom can be interpreted [1] as stating that our extended observational language $L_O{}^*$ will contain only applications concepts that can be regarded as having a data domain, a result domain and a condition, i.e., our application concepts can be apprehended by means of Polya's three questions.

Therefore, we will have new correspondence rules:

(d_3) $\quad (\forall d)(d \in D_{C(A)} \leftrightarrow \mathcal{B} f^{-1}(d) A)$

(d_4) $\quad (\forall r)(\exists \delta)(\mathcal{B} \delta A \wedge I\, \delta\, g^{-1}(r)\, A \rightarrow r \in R_{C(A)})$

(d_5) $\quad (\forall d)(\forall r)(\langle d, r\rangle \in q_{C(A)} \leftrightarrow \mathcal{B} f^{-1}(d) A \wedge I\, f^{-1}(d)\, g^{-1}(r)\, A)$

Notice that we are not claiming knowledge of C(A), but only its existence and uniqueness.

The process of writing a specification $\mathcal{S}pc$ begins with a verbalization $\mathcal{V}_A$, which describes not a single application concept A but a class $K_{\mathcal{V}}$, as we have discussed in section VI.1. Then, the weakest requirement one can impose on such a specification

---

[1] As Ludwig Wittgenstein states in his Tractatus Logico-Philosophicum: "was sich überhaupt sagen läßt, läßt sich klar sagen, und wovon man nicht reden kann, darüber muß man schweigen".

Spc, in order to ensure that one is solving the correct problem, is that it should be an abstraction of C(A).

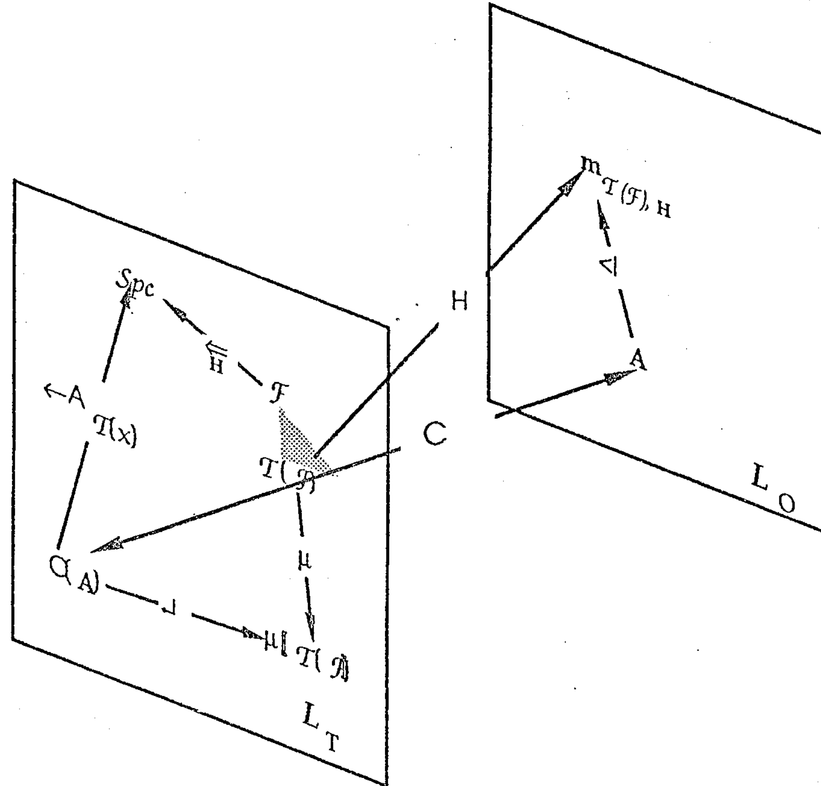

Figure VI.3.1

Figure VI.3.1 shows the connections among an application concept A, a specification *Spc* for it, and a program *p* satisfying *Spc*.

The concept of abstraction can be applied to the theoretical counterpart C(A) of an application concept A with respect to the theoretical counterpart of a decomposition term on the observational level. Then, the intuitive idea of being easy with respect to the application domain reflects the fact that a solution for the abstracted away part of an application can be supplied by the application domain itself, instead of the target machine. For instance, in developing large systems, one deals both with software artifacts, to become virtual machines, and with human activities providing subsidies for such machines[1]. In this sense we will use an abstraction of the application concept A, rather than A itself, as the observational object in the software process.

Now, we can state the *fundamental factorization theorem* for the relation of being-an-engineering-model.

VI.3.3. Theorem. $Spc \leftarrow\!\!A_{\mathcal{T}(x)}\, C(A) \wedge \mathcal{F} \Leftarrow_{H} Spc \rightarrow m_{\mathcal{T}(\mathcal{G}),\,H} \lhd A$

This factorization theorem states formally the general belief of the working software engineer: one can be sure that virtual machine $m_{\mathcal{T}(\mathcal{G}),\,H}$ will be an engineering model for application concept A if

i. one constructs a problem specification *Spc* that is an abstraction (up to the correspondence rules) of the application concept A, and

[1] Another interesting example is that of embedded software in which the software artifact (or more properly the virtual machine) belongs to a machinery environment with which is interrelated.

ii. one derives from this specification $\mathcal{Spc}$ a program $\mathcal{F}$, in the target programing language $\mathcal{L}_H$, that is totally correct with respect to $\mathcal{Spc}$,

From the theoretical viewpoint, this theorem provides conditions for the commuting of the diagram in figure VI.3.1, which relates the theoretical and observational levels involved in the software development process.

VII. PROBLEMS IN THE FACTORIZATION OF BEING-AN-ENGINEERING-MODEL.

VII.1. The synthetic character inherent to the software process.

As we have said, the assertion $m_{\mathcal{I}(\mathcal{F}), H} <\!| A$ is synthetic. This synthetic character arises from the fact that A is an extensional object. The only description one has for it is the set of pairs $\langle\delta, \rho\rangle$ such that $\mathcal{B}\delta A$ and $\mathcal{I}\delta\rho A$. In other words, we have no device in L_O^* for determining the extension of A. In fact, similar considerations apply to $m_{\mathcal{I}(\mathcal{F}), H}$ if we restrict ourselves to the observational language itself. The difference between both objects resides in the fact that $m_{\mathcal{I}(\mathcal{F}), H}$ is an observational object constructed from a term $\mathcal{F}$ of the formal language $\mathcal{L}_H$ and the set H of rules "interpreting " these terms $\mathcal{F}$ of $\mathcal{L}_H$. On the contrary, A is an observational object expressed directly in the observational language. Note that, although we postulate correspondence rules relating A and C(A), one's knowledge of both objects is only partial. So, we "construct" a formula $\mathcal{Spc}$ of $\mathcal{L}_G$ from an incomplete extensional knowledge of A and , except in trivial cases, one cannot know its complete extension because of its size.

Let us rewrite theorem VI.3.3 as:

$$\mathcal{Spc} \leftarrow A_{\mathcal{I}(x)}\, \mu[\![\, \mathcal{F}']\!] \wedge \mathcal{F} \underset{H}{\Leftarrow} \mathcal{Spc} \rightarrow m_{\mathcal{I}(\mathcal{F}), H} <\!| C(\mu[\![\, \mathcal{F}']\!])$$

i.e., we impose theoretical definitions on observational objects. This rewritten version is analytically determinate, since $m_{\mathcal{I}(\mathcal{F}), H}$ and $C(\mu[\![\, \mathcal{F}]\!])$ are "constructed" from formal objects defined by comprehension. But, one must bear in mind that the actual starting point of the process that yields $m_{\mathcal{I}(\mathcal{F}), H}$ via $\mathcal{Spc}$ and $\mathcal{F}$ is A itself. In addition, one requires $m_{\mathcal{I}(\mathcal{F}), H} <\!| A$; hence one has to accept that theorem VI.3.3 relates two synthetic formulas.

This theorem factorizes the synthetic character of $m_{\mathcal{I}(\mathcal{F}), H} <\!| A$ into a synthetic part, $\mathcal{Spc} \leftarrow A_{\mathcal{I}(x)}\, C(A)$, and an analytically determinate one, $\mathcal{F} \underset{H}{\Leftarrow} \mathcal{Spc}$. So it seems that we can validate $\mathcal{Spc} \leftarrow A_{\mathcal{I}(x)}\, C(A)$ by an hypothetico-deductive experiment and then prove $\mathcal{F} \underset{H}{\Leftarrow} \mathcal{Spc}$, instead of validating directly the disposition $m_{\mathcal{I}(\mathcal{F}), H} <\!| A$ as discussed in section IV.4.

Let us develop an experiment to test the hypothesis $\mathcal{Spc} \leftarrow A_{\mathcal{I}(x)}\, C(A)$:

Main hypothesis:
H_M: $\mathcal{Spc} \leftarrow A_{\mathcal{I}(x)}\, C(A)$.

Auxiliary hypotheses:
H_K: i. $\mathcal{Vib}\, C(A)$
ii. $\mathcal{Vib}\, \mu[\![\mathcal{Spc}]\!]$
iii. easiness of the abstraction term $\mathcal{I}(x)$.
H_L: i. $E_A \subseteq \{\delta: \delta \in V_O \wedge \mathcal{B}\delta A\}$

The use of the underlying theory T (FTP):
We derive that testing H_M is equivalent to testing
H_M^{i}. $D_{C(A)} \subseteq D_{\mu[\![\mathcal{I}(\mathcal{Spc})]\!]}$
H_M^{ii}. $q_{\mu[\![\mathcal{I}(\mathcal{Spc})]\!]}|_{D_{C(A)}} \subseteq q_{C(A)}$
We calculate
$\wp_D = \{d: d=f(\delta) \wedge \delta \in E_A\}$

$$\wp_R \subseteq \mathcal{R}(q_{\mu[\mathcal{T}(Spd)]}),$$
$$q_P \subseteq q_{\mu[\mathcal{T}(Spd)]}|_{\wp_D}$$

so that,
with $P^{\circ} = \langle \wp_D, \wp_R, q_P \rangle$ we have $\mathcal{V}ib P^{\circ}$
We prove (formally) that

$$\wp_D \subseteq D_{\mu[\mathcal{T}(Spd)]}$$

The correspondence rules C.

$$
\begin{aligned}
C=\{&(d_1) && (\forall\delta)\,(\mathcal{B}\delta A \leftrightarrow (\exists d)\,(d= f(\delta) \wedge \delta= f^{-1}(d)),\\
&(d_2) && (\forall\rho)(\exists\delta)\,(\mathcal{I}\delta\rho A \leftrightarrow (\exists r)\,(r= g(\rho) \wedge \rho= g^{-1}(r)),\\
&(d_3) && (\forall d)(d\in D_{C(A)} \leftrightarrow \mathcal{B}f^{-1}(d)A)\\
&(d_4) && (\forall r)(\exists\delta)(\mathcal{B}\delta A \wedge \mathcal{I}\,\delta\, g^{-1}(r)\, A \rightarrow r\in R_{C(A)}\,)\}\\
&(d_5) && (\forall d)(\forall r)(\langle d, r\rangle \in q_{C(A)} \leftrightarrow\\
& && \qquad \mathcal{B}f^{-1}(d)A \wedge \mathcal{I}\, f^{-1}(d)\; g^{-1}(r)\; A)
\end{aligned}
$$

Observational consequence:
Oc: $(\forall\delta)\;(\forall r)\;(\delta\in E_A \wedge r\in \wp_R \wedge \langle f(\delta), r\rangle\in q_P \rightarrow \mathcal{I}\,\delta\; g^{-1}(r)\; A.$

Now, if we fail to prove $\wp_D \subseteq D_{\mu[\mathcal{T}(Spd)]}$ then we must reject H_M^{i} and if the experiment falsifies the observational consequence then we must reject H_M^{ii}. Clearly in either case we must reject H_M.

On the contrary, what is the situation when we prove the above mentioned inclusion and the experiment does not falsify the observational consequence? Obviously, we will not reject H_M but should we accept it? As we have seen, this is not the case. We only accept that $Spc \leftarrow A_{\mathcal{T}(x)}\, P^{\circ}$, where P° is the viable problem calculated above.

Then, following the "recipe" of the hypothetico-deductive method we should choose other sets $E_A^1, E_A^2, \ldots, E_A^k$, construct problems $P^1, P^2, \ldots, P^k$, and validate them by using the same experiment scheme. Assume that these experiments do not succeed in falsifying H_M. Then, as working scientists, we will accept H_M, until some new evidence happens to falsify it.

VII.2. The inevitability of validation.

Consider k + 1 non-rejecting experiments involving problems $P^{\circ}, P^1, P^2, \ldots, P^k$ and let $P=\{P^i : 0\leq i\leq k\}$. Assume, for the sake of simplicity, that the problems in P are pairwise disjoint (see definition V.1.4). The only assertion we can make is:

$$Spc \leftarrow A_{\mathcal{T}(x)} \sum_{y\in P} y$$

After deriving a program $\mathcal{F}$ such that $\mathcal{F} \Leftarrow_H Spc$, the previous theorems guarantee only that:

$$m_{\mathcal{T}(\mathcal{F}),\,H} \lhd C\big(\sum_{y\in P} y\big)$$

Let us assume now that, after such a program $\mathcal{F}$, we perform an experiment with a set of data E_A^{k+1} and it rejects H_M. Then, we will have a problem P^{k+1} for which $D_{P^{k+1}} \subseteq D_{\mu[\mathcal{T}(Spd)]} \vee q_P \subseteq q_{C(A)}$ does not hold.

Let us analyze each case.

$\neg(D_{P^{k+1}} \subseteq D_{\mu[\mathcal{T}(Spd)]})$. In this case one has data δ belonging to the domain of the application concept (i.e., such that $\mathcal{B}\delta A$ holds) that are not "covered" by

Spc. Then, for each such δ, if $C_{\mathcal{B}\kappa\varphi}$ holds, virtual machine $m_{\mathcal{T}(\mathcal{F}),H}$ will halt at some time t but the result $m_{\mathcal{T}(\mathcal{F}),H}t(\delta)$ will not satisfy $\mathcal{I}\delta m_{\mathcal{T}(\mathcal{F}),H}t(\delta)A$.

$(D_P{}^{k+1} \subseteq D_{\mu[\mathcal{T}(Spc)]}) \wedge \neg(q_P \subseteq q_{C(A)} \mid\ )$. In this case we have two possibilities.

i. If $\mathcal{V}i\mathcal{b}\,\mu[\![Spc]\!]$ then the specification denotes a problem whose data domain "covers" the data domain of C(A) but it is not an abstraction of C(A). So, $m_{\mathcal{T}(\mathcal{F}),H}$ will be an engineering model of an application concept other than A, but intersecting A for every δ satisfying $\mathcal{B}\delta A$. This is a case in which this experiment is unlikely to point out the failure of *Spc* being an abstraction of C(A).

ii. If $\neg\mathcal{V}i\mathcal{b}\,\mu[\![Spc]\!]$, i.e. one has an incomplete specification, then any one of the above cases may happen.

Hence, after being sure of $\mathcal{F} \underset{H}{\Leftarrow} Spc$, no matter how much one has validated *Spc*, $m_{\mathcal{T}(\mathcal{F}),H}$ should be repeatedly validated with respect to A, as is the case with any construction in empirical science. In particular, the case of $C_{\mathcal{B}\kappa\varphi}$ failing to hold is one case in which $m_{\mathcal{T}(\mathcal{F}),H}$ may fail to halt even though $\mathcal{F} \underset{H}{\Leftarrow} Spc$ is guaranteed.

VII.3. The inevitability of vertical verification.

Proving correctness eliminates the possibility of dealing with a program that does not satisfy its specification. Theorem VI.3.3 guarantees that if $\mathcal{F} \underset{H}{\Leftarrow} Spc$, then the only possible sources of negative experimental outcomes are either the fact that $\neg(Spc \leftarrow A_{\mathcal{T}(x)}\ C(A))$, or the failure of some auxiliary hypothesis or of some rule of C.

Nevertheless, a naive software craftsman may consider that, since validation of disposition $m_{\mathcal{T}(\mathcal{F}),H} \lhd A$ is inevitable, there is no use at all for vertical verification. Nothing could be farther from the truth. As we have shown in section IV, $(\exists t)\ (t > t_0 \wedge \mathcal{H}m_{\mathcal{T}(\mathcal{F}),H}\delta t \wedge \mathcal{I}m_{\mathcal{T}(\mathcal{F}),H}t(\delta)A)$ is an $L_0{}^*$-undecidable property, unless one can restrict the domain of the existential quantifier by guaranteeing termination.

Hence, one must verify termination of $\mathcal{T}(\mathcal{F})$ to be "convinced " that virtual machine $m_{\mathcal{T}(\mathcal{F}),H}$ will halt. In fact, one should prove theoretical counterparts, i.e., theoretical predicates related to observational ones by means of rules of C, for every disposition of $L_O{}^*$ that is non-confirmable in principle.

Therefore, validation and verification are deeply imbricated and their use in a given order is not only a heuristic strategy but a matter of formal necessity.

VII.4. The inherent non-monotonicity of software development.

VII.4.1. Global non-monotonicity.

Let us state once more the fundamental factorization theorem VI.3.3:

$$Spc \leftarrow A_{\mathcal{T}(x)}\ C(A) \wedge \mathcal{F} \underset{H}{\Leftarrow} Spc \rightarrow m_{\mathcal{T}(\mathcal{F}),H} \lhd A$$

and analyze it under the light of the situation proposed in section VII.2.

Before the appearance of problem P^{k+1}, both premises of this theorem were true. But, after acquiring the new knowledge about A represented by P^{k+1}, the first conjunct, $Spc \leftarrow A_{\mathcal{T}(x)}\ C(A)$, becomes evidently false. This is a flagrant case of non-monotonicity. If one accepts the inherently synthetic character of $Spc \leftarrow A_{\mathcal{T}(x)}\ C(A)$, then one should accept that this non-monotonicity is inherent to the software development process itself. We call this phenomenon *global non-monotonicity*, since it involves the entire software process.

VII.4.2. Local non-monotonicity.

Let us now analyze, under the circumstances of the preceding paragraphs, the decomposition of the reification leg of the software process (see figure I.1) into steps

[Tur87]. To accomplish such a decomposition, one bridges the gap between $\mathcal{Spc}$ and $\mathcal{F}$ by introducing intermediate formulas $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_n$ of $\mathcal{L}_G$ but still not restricted to $\mathcal{L}_H$. In doing this, one creates a sequence of intermediate steps: from $\mathcal{Spc}$ to $\mathcal{F}_1$, from $\mathcal{F}_1$ to $\mathcal{F}_2$, ... , from $\mathcal{F}_i$ to $\mathcal{F}_{i+1}$, ... , and from $\mathcal{F}_n$ to $\mathcal{F}$. So, we can write VI.3.3 in the form:

$$Spc \leftarrow A_{\mathcal{T}(x)}\, C(A) \wedge \mathcal{F}_1 \sqsubset Spc \wedge \mathcal{F}_2 \sqsubset \mathcal{F}_1 \wedge \ldots \wedge \mathcal{F}_{i+1} \sqsubset \mathcal{F}_i \wedge \ldots \wedge \mathcal{F} \underset{H}{\Leftarrow} \mathcal{F}_n \rightarrow m_{\mathcal{T}(\mathcal{J}),\,H} \lhd A$$

Then, in view of VI.2.2.6, we can decompose the preceding statement into:

$$Spc \leftarrow A_{\mathcal{T}(x)}\, C(A) \wedge \mathcal{F}_1 \sqsubset Spc \rightarrow \mathcal{F}_1 \leftarrow A_{\mathcal{T}(x)}\, C(A)$$

$$\mathcal{F}_1 \leftarrow A_{\mathcal{T}(x)}\, C(A) \wedge \mathcal{F}_2 \sqsubset \mathcal{F}_1 \rightarrow \mathcal{F}_2 \leftarrow A_{\mathcal{T}(x)}\, C(A)$$

....

$$\mathcal{F}_i \leftarrow A_{\mathcal{T}(x)}\, C(A) \wedge \mathcal{F}_{i+1} \sqsubset \mathcal{F}_i \rightarrow \mathcal{F}_{i+1} \leftarrow A_{\mathcal{T}(x)}\, C(A)$$

....

$$\mathcal{F}_n \leftarrow A_{\mathcal{T}(x)}\, C(A) \wedge \mathcal{F} \underset{H}{\Leftarrow} \mathcal{F}_n \rightarrow m_{\mathcal{T}(\mathcal{J}),\,H} \lhd A$$

Note that the argument in the previous section applies to each one of these statements. So, no matter how "microscopic" are the development steps, non-monotonicity will spread to all of them. *Local non-monotonicity* is the name we give to the fact that non-monotonicity permeates into every development step.

VIII. CONCLUSIONS.

We have analyze the software development process by using Carnap's Two-level Theory of the Language of Science and the Algebraic Theory of Problems. This analysis has shown that many of Carnap's ideas concerning empirical science shed light on software development. In particular, the need of a theoretical level, distinct from the observational one, has been heavily felt for several reasons.

On the observational level one has application concepts and machines; on the theoretical level one has formal specifications and programs. This separation, in addition to its heuristic value, is a matter of necessity. This necessity arises from the fact that most interesting properties of machines turn out to be dispositions that are non-confirmable or non-refutable in principle. A prime example of non-refutability in principle is the case of machine halting. One has to deal with it on the theoretical level, by proving termination of the corresponding program. Before doing this, one cannot validate programs by means of experiments, since total correctness is neither confirmable nor refutable in principle.

The use of this formal framework has had two main advantages. On the one hand, we have been able to state and establish, in a precise way, some facts that are generally believed on intuitive grounds only. On the other hand, this framework has been helpful in paving the way to some important new facts. Thus, we have proved the synthetic character of correctness with respect to an application concept, as well as the inevitability of validation and verification, and the deep imbrication of both techniques for the analysis of program correctness with respect to the application concept. We have also formally proved that the software development process is inherently non-monotonic at any level of decomposition, in that it presents both global and local non-monotonicity. Therefore, any formalism that purports to describe software development must accommodate non-monotonicity and the decision between rival formalisms should be based on other features.

REFERENCES.

[Agr86] Agresti, W., W. (ed):
New Paradigms for Software Development. IEEE Computer Society Press, 1986.

[Boe76] Boehm, B., W.:
Software engineering, IEEE Trans. Comput., Dec. 1976, pp. 1226-1241.

[Car36] Carnap, R.,:
Testability and Meaning. Philosophy of Science, t. 3 (1936), and t. 4 (1937).

[Car56] Carnap, R.:
The Methodological Character of Theoretical Concepts, in Feigl, H., Scriven, M.,:Minnesota Studies in the Philosophy of Science, Minneapolis, 1956.

[Elu88] Elustondo, P., M.:
Informe Final de Pasantía. Proyecto ETHOS. ESLAI, 1988. Informe Final de Trabajo de Grado. ESLAI, 1988.

[Elu89] Elustondo, P., M., Veloso, P., A., S., Haeberer, A., M., Vazquez, L., A.
Program Development in the Algebraic Theory of Problems. Pont. UniversidadeCatólica; Res. Rept. ; Rio de Janeiro, 1989, forthcoming.

[Hae 87] Haeberer, A., M., Baum G., Veloso P., A., S.:
On an algebraic theory of problems and software development. Pont. Universidade Católica; Res. Rept. MCC 2/87; Rio de Janeiro, 1987.

[Hem65] Hempel C.:
Aspects of Scientific Explanation and Others Essays in the Philosophy of Science. Free Press, New York, 1965.

[Hem71] Hempel C.:
The Meaning of Theoretical Terms: A Critique of the Standard Empiricist Construal. In Suppes, P., Henkin, L., Joja A., Moisil, G., C.; Logic, Methodology and Philisophy of Science, IV. Proceedings of the 1971 International Congress. Bucarest, 1971.

[Leh84] Lehman, M., M., Stenning V. and Turski W., M.:
Another Look at Software Desing Methodology. ICST DoC Res. Rep., 83/13, London SW7 2BZ, Jun 1983.

[Mai84] Maibaum, T., S., E. and Turski, W., M.:
On What Exactly is Going On When Software is Developed Step by Step. Proc. 7th ICSE, Orlando, FA. March 1984. Publ. IEEE Comp. Soc., Silver Spring, MA. IEEE cat. no. 84ch2011-5, pp 525-533.

[Pat88] Partsch, H., :
The Majority Problem. Res Rept. KU Nijmegen. 1988

[Pol57] Polya, G.:
How to Solve it: a new Aspect of the Mathematical Method. Princeton Univ. Press, Princeton, 1957.

[Sho67] Shoenfield, J., R.,:
Mathematical Logic. Reading, Mass., Addison-Wesley, 1967.

[Ste70] Stegmüller W.,

Probleme und Resultate der Wissenschaftstheorie und Analytischen Philosophie, Band II: Theorie und Erfahrung. Springer Verlag, Heilderberg, 1970.

[Ton79] Tonies, C., C.:
Project Management Fundamentals. In Jensen, R., W., Tonies, C., C., Software Engineering, Prentice Hall, Eglewood Cliffs, N. J., 1979.

[Tur87] Turski, W., M., Maibaum, T., S., E.:
The Specification of Computer Programs. Addison-Wesley, Wokingham, 1987.

[Vel84] Veloso, P., A., S.:
Outline of a mathematical theory of general problems. Philosophia Naturalis; 21(2-4), 1984, pp. 354-365.

[Var89] Vargas, D., C., Haeberer, A., M.
Formal Theories of Problems. Pont. Universidade Católica; Res. Rept. MCC6/89; Rio de Janeiro, 1989.

[Vaz89] Vazquez, L., A., Elustondo, P., M.
Towards Construction with the Algebraic Theory of Problems: The Majority Problem. Pont. Universidade Católica; Res. Rept. ; Rio de Janeiro, 1989, forthcoming.

[Vel89] Veloso, P., A., S., Haeberer, A., M.
Software Development: A Problem-theoretic Analysis and Model. 22nd Hawaii International Conference on System Science, Honolulu, January 1989.

[Zar88] Zara, A.
El Método de Jackson como una instanciación de Divide-and-Conquer inducida por los datos. Informe de trabajo final. Universidad del Centro de la Provincia de Buenos Aires (Argentina), Facultad de Ciencias Exactas, 1988.