WHY SOFTWARE DEVELOPMENT IS INHERENTLY NON-MONOTONIC: A FORMAL JUSTIFICATION.

ARMANDO M. HAEBERER

and

PAULO A. S.VELOSO

Depto. Informática, Pont.Univ. Católica do Rio de Janeiro,Rua Mq. de S. Vicente 225
Rio de Janeiro, RJ 22453, Brasil

Abstract

We analyze the software process within a formal framework based on Carnap's Two-Level Theory of the Language of Science and the Algebraic Theory of Problems. As a result, we establish in a precise way some facts generally believed on intuitive grounds only, as well as the inherent non-monotonicity of software development.

1. Introduction

Software development involves four main objects: the application concept (requirements), reflecting the real problem extension; the verbalization (requirement specification), as the informal description of the requirements; the formal specification, as its formal description; and the program giving rise to a virtual machine, as the final product of the process. The specification divides the path from verbalization to program into two "legs"[6], called abstraction and reification, respectively.

Our formal analysis requires a formal framework for dealing with *SDP* (short for software development process) and its components at any level of abstraction, and sheds light on concepts like validation, verification, formal specification, abstraction, etc. We will use as formalism the Algebraic Theory of Problems[11, 4] and as formal structure Carnap's Two-Level Theory of the Language of Science[1, 2, 8]. We will analyze relations such as "describes", "realizes an engineering model", "is correct", etc., as well as the validation and proof obligations involved in factorizing *SDP*.

2. The Software Process and its Synthetic Content

The goal of *SDP* is the construction of a software artifact (*SA*, for short), starting from a description of some *real problem*. The *SA*, together with a given target machine, is to behave as an engineering model of the real problem. Notice that the starting point is generally ill-defined, whereas the end product is a complex syntactic object belonging to a formal language. Thus, *SDP* ranges over a wide spectrum of activities: formalization, abstraction, interpretation, construction of solutions, development of algorithms, validation, verification, etc.

By *application concept* A we mean the extensional knowledge about the real problem that serves as reference point of *SDP*. By its *verbalization* we mean a meta-linguistic description $\mathcal{V}_A$ of A. Since A is an extensional object, it cannot be informal; it is an observable object, which can be ill-determined because of lack of knowledge about its

extension. *SDP* starts from $\mathcal{V}_A$, the informal and ambiguous object that is often mistaken for A itself.

We will say that an *input* δ *belongs to the domain of* A , denoted $\mathcal{B}\delta A$, iff δ designates an acceptable input for A, and that the ordered pair ⟨δ, ρ⟩ is an *instance of* A, denoted $I\delta\rho A$, iff $\mathcal{B}\delta A$ 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.

An *SA* is a program or a set of programs in some programming language that, upon interpretation by a target machine, realizes a device that accepts inputs δ and produces outputs ρ. By *target machine* we mean a device H, made out of hardware and software, that is capable of interpreting a program *p*. By *virtual machine* we mean the device $\mathbf{m}_{pH}$ constructed by interpreting a software artifact *p* on H. By the *result of* $\mathbf{m}_{pH}$ *for data* δ *at instant* t, denoted $m_{pH}\ t(\delta)$, we mean the output ρ produced by $\mathbf{m}_{pH}$ at instant t after being fed data δ, provided that it halts. We will use the notations $\mathcal{A}\mathrm{m}_{pH}\delta$ for "m_{pH} is applied to input δ" and $\mathcal{H}\mathrm{m}_{pH}\delta t$ for "m_{pH} on input δ halts at instant t".

We shall be dealing with analytically determinate and synthetic statements. We gloss over some polemical issues and assume that every meaningful scientific statement can be classified as either one or the other[8].

We now try to state in precise terms the meaning of "being-an-engineering-model", denoted by $\mathbf{m}_{pH} \angle A$. It is natural to try to use the operational definition:

$$\mathbf{m}_{pH} \angle A \leftrightarrow (\forall\delta)(\mathcal{B}\delta A \wedge \mathcal{A}\mathrm{m}_{pH}\delta \rightarrow (\exists t)(\mathcal{H}\mathrm{m}_{pH}\delta t \wedge I\delta m_{pH}\, t(\delta)A)) \qquad (1)$$

But, as Carnap showed[1], operational definitions fail to accomplish their goal, because a conditional with a false antecedent turns out to be true. Thus, Eq. (1) implies that a virtual machine that has never been tested with respect to A turns out to be an engineering model of A! So, it is replaced by the pair of sentences (where the first one is a *reductive sentence*)

$$(\forall\delta)(\mathcal{B}\delta A \wedge \mathcal{A}\mathrm{m}_{pH}\delta \rightarrow (\mathrm{m}_{pH} \angle_\delta A \leftrightarrow ((\exists t)(\mathcal{H}\mathrm{m}_{pH}\delta t \wedge I\delta m_{pH}\ t(\delta)A))) \qquad (2)$$

$$\mathbf{m}_{pH} \angle A \leftrightarrow (\forall\delta)(\mathcal{B}\delta A \rightarrow \mathbf{m}_{pH} \angle_\delta A) \qquad (3)$$

Notice the factual character of these formulations which stems from the universal quantification over the set of inputs δ belonging to the application domain. Except in rare cases, this set is not exhaustible, so one cannot fulfill the condition "for every input δ" . One generally induces it from a certain sample. Hence, these formulas are in fact synthetic statements in the sense of empirical science, which is not surprising since ∠ relates directly observable objects.

3. The Two Level Theory : Observational and Theoretical Objects

Some difficulties in introducing dispositions in the language of science led Carnap to abandon both the need to introduce them only by means of reductive sentences and the idea of a global empiricist language L_E in favor of the Two-Level Theory of the Language of Science[8].

In our context, the most serious of these difficulties stems from the confirmability and refutability in principle of the observable terms. Observationally decidable terms are introduced by definitions providing explicit criteria for confirming or refuting a property; the remaining terms are observationally undecidable[5]. Thus, the predicate $(\exists t)(\mathcal{H}\mathrm{m}_{pH}\delta t \wedge I\delta m_{pH}\ t(\delta)A)$ is not observationally decidable, due to the existential quantifier.

3. 1. *Observational and Theoretical Levels*

Carnap proposed to split the language of science into two: the basic empiricist language, L_O, understandable by itself, and L_T, the language for formulating a theory. The latter is not understandable by itself and does not have a complete empirical interpretation, a partial one being given by means of a set C of correspondence rules.

The *observational language* L_O is extensional, completely interpreted, and its predicates designate observable properties of events or things. Here, we use an extended L_O*, which allows the introduction of L_O-definable dispositional predicates. Some requirements are imposed[2, 8] on L_O*. The symbols of its vocabulary, V_O*, have no formal denotation, their meaning being reduced to observation.

The *theoretical language* L_T is designed to have all the freedom needed. Its primitive descriptive (or extralogical) symbols are not required to be explicitly defined on the basis of L_O*. A Theory T in L_T consists of a finite number of postulates in it, thus being an uninterpreted calculus.

Correspondence rules are formulas involving descriptive symbols both of L_T and of L_O*, thereby providing a partial interpretation of L_T in terms of L_O*. Now, theory T, together with C, becomes an interpreted calculus. The rules in C establish deductive connections between L_O* and L_T ; thus , in deriving a sentence of L_O* from sentences of L_O*, we can go via L_T . We will use C(...) to express the result of applying the rules of C to an object ... belonging either to L_T or to L_O*.

3.2. *Observational and Theoretical Objects in the Software Process*

The observational level for *SDP* comprises A and m_{pH}. Their inputs δ and outputs ρ belong to an *observable universe W*.We also have in this level a denumerable ordered set called *time*. So, statements such as $\mathcal{B}\delta A$ and $\mathcal{A}m_{pH}\delta$ belong to L_O*. Statements like $m_{pH} \angle A$ and $\mathcal{H}m_{pH}\delta t$ are also formulated in L_O* but must be dealt with in L_T.

The theoretical level is split into two layers, the syntactical and the semantical one. The syntactical layer has programs and specifications; the semantical layer comprises their denotations, problems, as will be explained shortly. Then, the Algebraic Theory of Problems, Set Theory, the Fixpoint Theory of programs, etc. will all be part of the theoretical level.

The correspondence rules relate observational inputs δ and outputs ρ to their theoretical counterparts d and r. Thus, they correlate denotations of specifications and observational objects (i.e. application concepts). H plays a special role in that it provides a set H of correspondence rules that connect a program p to m_{pH}, like an interpreter. So, m_{pH} gives the extensional behavior of p, which is a description by comprehension of m_{pH}. Likewise a specification tries to describe by comprehension an application concept.

4. A Problem-theoretic Approach to Software Development

A General Theory of Problems[11] was developed from the ideas of G. Polya[7] as a formal tool for reasoning about problem solving and modeling various strategies, techniques, methods, etc. On this basis, an Algebraic Theory of Problems[4] was developed, aiming at a tool for the formal treatment of *SDP* at various levels, ranging from the purely epistemic one of process explication[3], through those of prescribing different process obligations and modeling programming methods , to a calculus for program derivation[4]. In the sequel we will outline a version of the Algebraic Theory of Problems; for more details, see[4].

4.1 The Algebraic Theory of Problems

A *problem* over a *theoretical universe* $\mathcal{U}$ is a 3-tuple $P = \langle D_P, R_P, \mathcal{P} \rangle$ where D_P and D_P are subsets of $\mathcal{U}$ and $\mathcal{P} \subseteq D \times R$. This mathematical structure captures the three questions suggested by Polya in approaching a problem: *what are the data? what are the results?* and *what is the problem condition?* , in that D_P is the *data domain*, R_P the *result domain* and $\mathcal{P}$ the problem *condition* (also denoted by q_P). We say that problem $P = \langle D_P, R_P, \mathcal{P} \rangle$ is *viable*, denoted by $\mathcal{V}ib P$, iff $(\forall d)(d \in D_P \rightarrow (\exists r)(r \in R_P \wedge \mathcal{P}(d, r)))$. Notice that $\mathcal{V}ib P$ is equivalent to the equality of $\mathcal{D}om\, \mathcal{P}$ and D_P.

The 3-tuple representation 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.

Given problems P and Q, we define their *sum* $P + Q = \langle D_P \cup D_Q, R_P \cup R_Q, \mathcal{P} \cup \mathcal{Q} \rangle$, their *product* $P \bullet Q = \langle D_P , R_Q, \mathcal{P}/\mathcal{Q} \rangle$ (where / stands for the relative product of relations) and their *direct product* $P \times Q = \langle D_P \times D_Q, R_P \times R_Q, \{\langle\langle d, r\rangle, \langle d', r'\rangle\rangle : \langle d, r\rangle \in \mathcal{P} \wedge \langle d', r'\rangle \in \mathcal{Q}\}\rangle$. We will denote by $P^{\bullet n}$ the product of P by itself n times. By resorting to the generalized union and Cartesian product, we extend the binary operations sum and direct product to classes of problems, yielding *summation* (Σ) and *generalized direct product*. We will use $P^{\times n}$ to denote the direct product of n copies of P . Along these lines, $P^{\bullet *}$ and $P^{\times *}$ denote the closure of P under product and direct product, respectively.

Given P and Q, we define their *difference* $P - Q = \langle D_P - D_Q, R_P - R_Q, (\mathcal{P}-\mathcal{Q}) \cap ((D_P - D_Q) \times (R_P - R_Q))\rangle$ (where – stands for set difference) and the *inverse* $P^{-1} = \langle R_P, D_P, \mathcal{P}'' \rangle$, where $\mathcal{P}''$ is the *converse*[9] of $\mathcal{P}$.

Some important relations between problems were defined. Given problems P and Q, we say that P is a *relaxation* of Q (denoted $P \lrcorner Q$) iff $D_Q \subseteq D_P$ and $\mathcal{P}|_{D_Q} \subseteq \mathcal{Q}$, P is a *subproblem* of Q (denoted $P \subseteq Q$) iff there exists a problem R such that $P + R = Q$, and P is a *complete subproblem* of Q (denoted $P \subseteq_C Q$) iff $P \subseteq Q$ and $D_P = D_Q$. The relations $\lrcorner$, $\subseteq$ and $\subseteq_C$ are transitive. Notice that $\subseteq_C$ and = (equality between problems) are special cases of relaxation, deserving attention due to their better monotonicity properties. The decision of restricting the correctness relation to one of them is part of the particular software construction strategy being used.

Let be given a set $\mathfrak{V}$ of special problems, which will be called *easy problems*, and a set $\mathbb{C}$ of distinguished problems. By the *algebra of problems* over $\mathfrak{V}$ and $\mathbb{C}$ we mean the algebra $\mathcal{A} = \langle \mathbb{P}, \mathbb{C}, \mathfrak{V}, \{+, \bullet, \times, -, \Sigma, ^{-1}, ^{\times n}, ^{\times *}, ^{\bullet *}\}\rangle$, where $\mathbb{P}$ is the set of all the problems over $\mathcal{U}$.

4.2. Specifications, problems, programs and solutions

A general-purpose target machine interprets a finite, and usually not too large, set of constant symbols over problems ; their denotations form our set $\mathfrak{V}$.

Problems are on the semantical layer. We obtain the syntactical layer by introducing symbols for the constants and operations of the algebra $\mathcal{A}$. We will consider a language $\mathcal{L}$, called *global language*, with symbols for the constants and operations of $\mathcal{A}$ (we also use variables over problems in our program derivation calculus[4], but we do not need them here). Let us denote by T the set of terms of $\mathcal{L}$ and by $\mathcal{E}$ the corresponding algebra of terms generated from $\mathfrak{V} \cup \mathbb{C}$. There is a unique homomorphism

μ: $\mathcal{E} \rightarrow \mathcal{A}$, assigning a problem to each term. Such a term T∈T is a specification and the semantic function μ assigns to T the problem it describes.

A constant symbol f, denoting a problem $\mu[f]$, is said to be *easy* with respect to H iff f appears in some correspondence rule of a subset H of C, which includes the instructions of H, as well as the fetching and decoding mechanism[3]. We call a term *algorithmic* iff all its operation symbols correspond to algorithmic operations, +, •, ×, $\times n$, $\times^*$, $\bullet^*$ and *target* iff it is algorithmic and all its constant symbols are easy with respect to H. We will denote by T_H the set of target terms and by $\mathcal{L}_H$ the corresponding *target language.*

We can now consider notions of correctness between specifications, by lifting semantical notions to the syntactical layer. We refer to specifications, rather than to programs, to encompass the entire construction process, throughout specifications and (nondeterministic) programs. Given terms $\mathcal{F}$ and $\mathcal{G}$ of $\mathcal{L}$, we say that $\mathcal{F}$ is *partially correct* with respect to $\mathcal{G}$ (denoted $\mathcal{F} \leq \mathcal{G}$) iff $\mu[\mathcal{F}] \sqsupseteq \mu[\mathcal{G}]$; $\mathcal{F}$ *terminates* (denoted $\mathcal{TF}$) iff $\mathcal{V}ib\ \mu[\mathcal{F}]$, and $\mathcal{F}$ is *totally correct* with respect to $\mathcal{G}$ (denoted $\mathcal{F} < \mathcal{G}$) iff $\mathcal{TF}$ and $\mathcal{F} \leq \mathcal{G}$. Given terms p and $\mathcal{G}$ of $\mathcal{L}$, we say that p is a *solution* for $\mathcal{G}$, with respect to H, denoted $p \Leftarrow_H \mathcal{G}$, iff $p < \mathcal{G}$ and $p \in T_H$. Then, a *solution* for a problem P , with respect to H, is a is a target term p such that $\mu[p] \sqsupseteq P$ and $\mu[p]$ is viable; we denote this by $p \Leftarrow_H P$. These definitions of solution capture the idea of *construction* of an object for H.

An important tool for the analysis of the connection between A and *Spc* will be the following substitutivity result relating relaxation and solution.

Theorem. $$p \Leftarrow_H \mathcal{F} \wedge \mu[\mathcal{G}] \sqsupseteq \mu[\mathcal{F}] \rightarrow p \Leftarrow_H \mathcal{G} \tag{4}$$

5. Connecting the Theoretical and Observational Levels of the Software Process

In our framework we have data d and results r in the theoretical universe $\mathcal{U}$, whereas inputs δ and outputs ρ are in the observable universe $\mathcal{W}$. The distinction between $\mathcal{U}$ and $\mathcal{W}$ should be kept in mind. The only connection between them is provided by a function f : $\mathcal{W} \rightarrow \mathcal{U}$. In such a context, any attempt to treat A as a problem must be based on an explicit connection. So, we state:

Postulate. Every application concept A in L_O^* is *coextensive* with a problem C(A) in L_T (i.e. their extensions are *congruent* up to the correspondence rules C).

This desideratum can be interpreted as stating that our extended L_O^* will contain only application concepts that can be apprehended by means of Polya's three questions. (As Ludwig Wittgenstein states in his Tractatus Logico-Philosophicus: *was sich überhaupt sagen läßt, läßt sich klar sagen, und wovon man nicht reden kann, darüber muß man schweigen*).

Thus, A and C(A) are connected by the following natural correspondence rules:

$$d \in D_{C(A)} \leftrightarrow (\exists \delta)(d = f(\delta) \wedge \mathcal{B}\delta A) \tag{5}$$

$$(\exists \delta)(\mathcal{B}\delta A \wedge I\delta\rho A) \rightarrow f(\rho) \in R_{C(A)} \tag{6}$$

$$\langle d, r \rangle \in q_{C(A)} \leftrightarrow (\exists \delta)(\exists \rho)(d = f(\delta) \wedge r = f(\rho) \wedge \mathcal{B}\delta A \wedge I\delta\rho A) \tag{7}$$

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

Now, let us make explicit the connection between p and m_{pH}, which is provided by the subset H of C. If $\mu[f] \in \mathcal{V}$, H is able to choose for every data δ, such that $f(\delta) \in D_{\mu[f]}$, a

result ρ, so that ⟨f(δ), f(ρ)⟩ ∈ $q_{\mu[f]}$. Similarly for p. Thus, p and m_{pH} are connected by the following correspondence rules:

$$\mathcal{A}m_{pH}\delta \leftrightarrow f(\delta) \in D_{\mu[p]} \tag{8}$$

$$\mathcal{A}m_{pH}\delta \rightarrow ((\exists t)(\mathcal{H}m_{pH}\delta t) \leftrightarrow (\exists r)(r \in R_{\mu[p]} \wedge \langle d, r\rangle \in q_{\mu[p]})) \tag{9}$$

$$\mathcal{A}m_{pH}\delta \rightarrow ((\exists t)(\mathcal{H}m_{pH}\delta t \wedge \rho = m_{pH}t(\delta)) \leftrightarrow \langle f(\delta), f(\rho)\rangle \in q_{\mu[p]}) \tag{10}$$

Call Z the conjunction of the preceding 6 formulas. We now have:

$$Z \vdash p \underset{H}{\Leftarrow} C(A) \rightarrow m_{pH} \angle A \tag{11}$$

Recall that Eq. (2) involves two crucial aspects: halting of m_{pH} and appropriateness of the outputs. We examine halting first.

5.1. *The Need for Theoretical Reasoning*

The crucial difference between $\mathcal{T}p$ and $(\exists t)(\mathcal{H}m_{pH}\delta t)$ resides in that the former is a predicate of L_T, hence amenable to formal proof, whereas the latter is in L_O*, but is not L_O*-decidable. Thus, one can try to prove the termination of p.

Predicates $\mathcal{T}$, of L_T, and $\mathcal{H}$, of L_O*, are connected by the rule derived from Z:

$$\mathcal{T}p \rightarrow (\forall\delta)(\mathcal{A}m_{pH}\delta \rightarrow (\exists t)(\mathcal{H}m_{pH}\delta t)) \tag{12}$$

We can Skolemize $(\exists t)(\mathcal{H}m_{pH}\delta t)$ by introducing a function symbol ζ so that ζ(δ) means "the instant the machine halts after the introduction of the data δ", in the following sense:

$$\vdash (\exists t)(\mathcal{H}m_{pH}\delta t) \rightarrow \mathcal{H}m_{pH}\delta\zeta(\delta) \tag{13}$$

Hence, we can solve our L_O*-undecidability problem, by means of a formal proof of $\mathcal{T}p$, which ensures that $\mathcal{H}m_{pH}\delta t$ will hold for some finite t= ζ(δ), since

$$Z \wedge \mathcal{T}p \vdash (\forall\delta)(\mathcal{B}\delta A \wedge \mathcal{A}m_{pH}\delta \rightarrow \mathcal{H}m_{pH}\delta\zeta(\delta) \wedge (m_{pH} \angle A \rightarrow I\delta m_{pH}\zeta(\delta)(\delta)A)) \tag{14}$$

Now we are able to formulate an experiment[8] for H_M: $m_{pH} \angle A$. We add an auxiliary theoretical hypothesis H_K: $\mathcal{V}ib\mu[p]$ and a descriptive observational statement H_L: $\bigwedge_{i=1}^{n} \mathcal{B}\delta_i A \wedge \bigwedge_{i=1}^{n} \mathcal{A}m_{pH}\delta_i$. By proving $\mathcal{V}ib\mu[p]$ and using the previous correspondence rules, we define *Oc* as $\bigwedge_{i=1}^{n} I\delta_i m_{pH}\,\zeta(\delta_i)A$. If *Oc* fails, we can reject H_M, or doubt the truth of a statement like $\mathcal{B}\delta_i A$. On the contrary, if *Oc* holds, we can consider it only as a good inductive support for H_M.

5.2. *The Fundamental Factorization of Software Development*

The weakest requirement one can impose on *Spc* to ensure that one is solving the correct problem is $\mu[Spc] \lrcorner C(A)$. Now, from Eqs. (4) and (11) we derive the *fundamental factorization theorem* for ∠.

Theorem.

$$\mu[Spc] \lrcorner C(A) \wedge p \underset{H}{\Leftarrow} Spc \rightarrow m_{pH} \angle A \tag{15}$$

This theorem states formally the general belief of the working software engineer: one can be sure that $m_{pH} \angle A$ provided that (i) one constructs *Spc* whose denotation is a relaxation (up to the correspondence rules) of **A**, and (ii) one derives from *Spc* a program p, in L_H, that is totally correct with respect to *Spc*.

6. The Inherent Non-monotonicity of the Software Development Process

Notice that Eq. (15) relates two synthetic formulas. It factorizes the synthetic character of $\mathbf{m}_{pH} \angle \mathbf{A}$ into a synthetic part, $\mu[Spc] \lrcorner \mathbf{C(A)}$, and an analytically determinate one, $p \underset{H}{\Leftarrow} Spc$. So, we can validate $\mu[Spc] \lrcorner \mathbf{C(A)}$ by a hypothetico-deductive experiment and prove $p \underset{H}{\Leftarrow} Spc$, instead of directly validating $\mathbf{m}_{pH} \angle \mathbf{A}$.

Let us develop an experiment Exp to test the hypothesis $\mu[Spc] \lrcorner \mathbf{C(A)}$. We set H_M: $\mu[Spc] \lrcorner \mathbf{C(A)}$; H_K: $\mathcal{V}ib\mathbf{C(A)} \wedge \mathcal{V}ib\mu[Spc]$; H_L: $E_A \subseteq \{\delta: \delta \in V_O^* \wedge \mathcal{B}\delta\mathbf{A}\}$. Now, by using T (the Algebraic Theory of Problems) we derive that H_M is equivalent to H_{M1}: $D_{\mathbf{C(A)}} \subseteq D_{\mu[Spc]}$ and H_{M2}: $q_{\mu[Spc]}|_{D_{\mathbf{C(A)}}} \subseteq q_{\mathbf{C(A)}}$. Now, we calculate $\wp_D = \{d: d=f(\delta) \wedge \delta \in E_A\}$, $\wp_R \subseteq \mathcal{R}anq_{\mu[Spc]}$, and $\wp_q \subseteq q_{\mu[Spc]}|_{\wp_D}$, so that we have $\mathcal{V}ib\mathbf{P}^\circ$ for $\mathbf{P}^\circ = \langle \wp_D, \wp_R, \wp_q \rangle$. Then, we prove that $\wp_D \subseteq D_{\mu[Spc]}$. The correspondence rules will be Z, and the observational consequence Oc: $(\forall\delta)(\forall\rho)(\delta \in E_A \wedge f(\rho) \in \wp_R \wedge \langle f(\delta), f(\rho)\rangle \in \wp_q \rightarrow I\delta\rho\mathbf{A})$.

If we fail to prove $\wp_D \subseteq D_{\mu[Spc]}$ then we must reject H_{M1}, and if Exp falsifies Oc then we must reject H_{M2},rejecting H_M in either case. Otherwise, we can accept only that $\mu[Spc] \lrcorner \mathbf{P}^\circ$. Then, we should choose other sets $E_A^1, \ldots, E_A^k$, construct problems $\mathbf{P}^1, \ldots, \mathbf{P}^k$, and validate them by using Exp again. If these experiments do not falsify H_M, then we will accept H_M, until some new evidence happens to falsify it.

Consider a set P of k+1 non-rejecting experiments involving problems $\mathbf{P}^\circ, \mathbf{P}^1, \ldots, \mathbf{P}^k$, assumed, for simplicity, to have pairwise disjoint data domains. All we are entitled to assert is $\mu[Spc] \lrcorner \Sigma\mathrm{P}$ Now, assume that $p \underset{H}{\Leftarrow} Spc$. The previous results guarantee only that $\mathbf{m}_{pH} \angle \mathbf{C}(\Sigma\mathrm{P})$. Consider now an experiment with set of data E_A^{k+1} that falsifies H_M. Then, we will have $\mathbf{P}^{k+1}$ such that $D_{\mathbf{P}^{k+1}} \subseteq D_{\mu[Spc]} \vee q_{\mathbf{P}^{k+1}} \subseteq q_{\mu[Spc]}|_{\wp_D}$ does not hold[3]. Hence, after being sure of $p \underset{H}{\Leftarrow} Spc$, no matter how much one has validated Spc, $\mathbf{m}_{pH}$ should be repeatedly validated with respect to A, like any construction in empirical science. In particular, the case of $D_{\mathbf{C(A)}} \subseteq D_{\mu[p]}$ failing to hold is one in which $\mathbf{m}_{pH}$ may fail to halt even though $p \underset{H}{\Leftarrow} Spc$ is guaranteed. In addition, one must verify termination of p to be "convinced " that $\mathbf{m}_{pH}$ will halt. In fact, one should prove theoretical counterparts for every disposition non L_O^*-decidable. Therefore, validation and verification are deeply imbricated and their use in a given order is not only a heuristic strategy but a formal necessity.

Before the appearance of $\mathbf{P}^{k+1}$, both premises of Eq. (15) were true. But, after acquiring this new knowledge about A, $\mu[Spc] \lrcorner \mathbf{C(A)}$ becomes false. This is a flagrant case of non-monotonicity. In view of the inherent syntheticity of $\mu[Spc] \lrcorner \mathbf{C(A)}$, this *global non-monotonicity* is inherent to *SDP* itself.

We now analyze the decomposition of the reification leg of *SDP* into steps[10], which bridges the gap between Spc and p by introducing intermediate terms $\mathcal{F}_1, \ldots, \mathcal{F}_n$ of $\mathcal{L}$. So, we can write Eq. (15) in the form:

$$\mu[Spc] \lrcorner \mathbf{C(A)} \wedge \mathcal{F}_1 < Spc \wedge \ldots \wedge \mathcal{F}_n < \mathcal{F}_{n-1} \wedge p \underset{H}{\Leftarrow} \mathcal{F}_n \rightarrow \mathbf{m}_{pH} \angle \mathbf{A}. \qquad (16)$$

Then, we can decompose the preceding statement into:

$$\mu[Spc] \lrcorner C(A) \wedge \mathcal{F}_1 <Spc \rightarrow \mu[\mathcal{F}_1] \lrcorner C(A);\ldots.; \mu[\mathcal{F}_{n-1}] \lrcorner C(A) \wedge \mathcal{F}_n < \mathcal{F}_{n-1} \rightarrow \mu[\mathcal{F}_n] \lrcorner C(A);$$
$$\mu[\mathcal{F}_n] \lrcorner C(A) \wedge p \underset{H}{\Leftarrow} \mathcal{F}_n \rightarrow m_{pH} \angle A. \quad (17)$$

The previous argument now applies to each one of these statements. So, no matter how "microscopic" the development steps, this *local non-monotonicity* will spread to all of them.

7. Conclusions

We have analyzed the software development process by using Carnap's Two-level Theory of the Language of Science and the Algebraic Theory of Problems, which shows that many of Carnap's ideas on empirical science improve our understanding of software development.

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.

This formal framework is quite advantageous. It enables us to state and establish, in a precise way, some facts that are believed on intuitive grounds only. It also paves 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. We have also formally proved that the software development process is inherently non-monotonic at any level of decomposition.

Acknowledgements

Research reported herein has been partly sponsored by ETHOS Project (Argentine-Brazilian Program for Research and Advanced Studies in Computer Sciences) and the Brazilian agencies FINEP and FAPERJ.

References

1. R. Carnap, "Testability and Meaning", *Philosophy of Science* **3** (1936), **4** (1937).
2. R. Carnap, "The Methodological Character of Theoretical Concepts", *Minnesota Studies in the Philosophy of Science*, (Minneapolis 1956).
3. A. M. Haeberer and P. A. S. Veloso, "The Inevitability of Program Testing: A Theoretical Analysis",*Proc. of the IX Intern. Conf. Chilean Computer Sci. Soc.*, 208-240 (1989).
4. A. M. Haeberer, P. A. S. Veloso and P. Elustondo, "Towards a Relational Calculus for Software Construction", *Pont. Univ. Católica; Res. Rept. MCC 19/89*, (Rio de Janeiro,1989).
5. C. Hempel, "Aspects of Scientific Explanation and Others Essays in the Philosophy of Science", Free Press, (New York, 1965).
6. M. Lehman, "A Further Model of Coherent Programming Process", *IEEE Proceedings of Software Process Workshop, UK Feb 1984*, (IEEE C. S.,1984).
7. G. Polya, "How to Solve it: a new Aspect of the Mathematical Method", Princeton Univ. Press, (Princeton, 1957).
8. W. Stegmüller, "Probleme und Resultate der Wissenschaftstheorie und Analytischen Philosophie" **2**, Springer-Verlag, (Heilderberg, 1970).
9. A. Tarski, "On The Calculus of Relations", *Journal of Symbolic Logic* **6**, 73-89 (1941).
10. W. M. Turski and T. S. E. Maibaum, "The Specification of Computer Programs", Addison-Wesley, (Wokingham, 1987).
11. P. A. S. Veloso, "Outlines of a mathematical theory of general problems" *Philosophia Naturalis* **21**, 354-362 (1984).