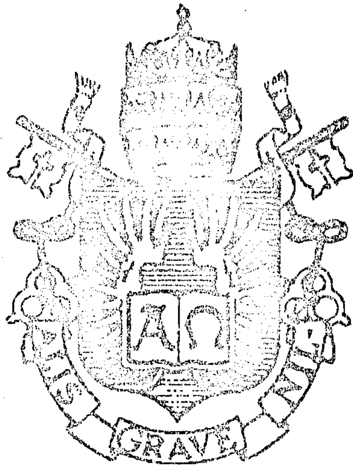

PUC

Monografias em Ciência da Computação
nº 18/92

Towards Formal Coherent Meta-Models for the Software Development Process

Armando M. Haeberer
Paulo A. S. Veloso
Thomas S. E. Maibaum

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 18/92
Editor: Carlos J. P. Lucena Abril, 1992

Towards Formal Coherent Meta-Models for the Software Development Process *

Armando M. Haeberer
Paulo A. S. Veloso
Thomas S. E. Maibaum #

Dept. Computing, Imperial College of Science, Technology and Medicine; London

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:

Rosane Teles Lins Castilho

Assessoria de Biblioteca, Documentação e Informação

PUC Rio — Departamento de Informática

Rua Marquês de São Vicente,225 — Gávea

22454970 — Rio de Janeiro, RJ

Brasil

Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645

E-mail: rosane@inf.puc-rio.br

techrep@inf.puc-rio.br (for publications only)

TOWARDS FORMAL COHERENT META-MODELS FOR THE SOFTWARE DEVELOPMENT PROCESS

Armando M. HAEBERER[†] Paulo A. S. VELOSO[†] Thomas S. E. MAIBAUM[+]

ABSTRACT

A theoretical basis and a unifying conceptual framework are essential for a coherent model of the processes involved in software development. Here we analyse basic needs of such formal coherent meta-models, which lead to several levels, for distinct objects and for relating them. These needs stem from formal reasons, epistemological considerations and heuristic convenience. The formal reasons arise from a precise analysis of the goal of software development, the epistemological considerations are connected to non-reductionistic explications, and the heuristic convenience has to do with instantiating the meta-model to cover various paradigms. We also outline a multidimensional meta-model of the software development process, which is based on these general ideas and centred around logical concepts, such as interpretations between theories, and an extended calculus of binary relations.

Key words:

Software development, formal methods, development methods and paradigms, coherent meta-models, formal logic, software development process, formal modeling, program derivation calculi.

---

† Dept. Informática, Pontifícia Universidade Católica; 22453 Rio de Janeiro, Brazil (Fax: (55) (21) 511 5645; e-mail: armando@inf.puc-rio.br., veloso@inf.puc-rio.br).

+ Dept. Computing, Imperial College of Science, Technology and Medicine; London SW7 2BZ, UK (Fax: (44) (71) 581 8024; e-mail: tsem@doc.ic.ac.uk).

RESUMO

Uma base teórica e um arcabouço conceitual unificado são essenciais para um modelo coerente dos processos envolvidos em desenvolvimento de software. Aqui se analisam as necessidades básicas de tais meta-modelos formais coerentes, o que conduz a vários níveis: para objetos distintos e para relacioná-los. Essas necessidades decorrem de razões formais, considerações epistemológicas e conveniência heurística. As razões formais advêm de uma análise precisa dos objetivos do desenvolvimento de software, as considerações epistemológicas se relacionam a explicações não-reducionistas e a conveniência heurística tem a ver com a possibilidade de instanciar o meta-model de modo a cobrir vários paradigmas. Também se apresenta um meta-modelo multidimensional do processo de desenvolvimento de software, o qual se baseia nessas idéias gerais além de ser centrado em torno conceitos lógicos, como interpretações entre teorias, e um cálculo estendido de relações binárias.

Palavras chave:

Desenvolvimento de software, métodos formais, métodos e paradigmas de desenvolvimento, meta-modelos coerentes, lógica formal, processo de desenvolvimento de software, modelagem formal, cálculos de derivação de programas.

CONTENTS

I. INTRODUCTION 1

II. THE SOFTWARE DEVELOPMENT PROCESS 1

II.1. THE GOAL OF THE PROCESS 1

II.2. THE ACTORS AND THEIR ROLE 2

III. NEEDS IN MODELING THE SOFTWARE DEVELOPMENT PROCESS 2

III.1. THE BASIC LEVELS 2

III.1.1. The Observable Level 2

III.1.2. The Program Text Level 3

III.1.3. The Fundamental Relation 3

III.2. FORMAL REASONS FOR MORE LEVELS 4

III.2.1. The Need for Theoretical Levels 4

III.2.2. The Need for Transversal Planes 5

III.2.3. The Need for The Plane of I/O Relations 5

III.3. HEURISTIC CONVENIENCE OF THE MULTIPLICITY OF LEVELS 6

III.3.1. Development within Theoretical Levels 6

III.3.2. Passage between Levels 7

III.3.3. Tests, Proofs, Derivations, and Actions 7

III.4. THE BASIC NEEDS IN MODELING THE SOFTWARE DEVELOPMENT PROCESS 9

IV. A MULTIDIMENSIONAL (META-)MODEL OF THE SOFTWARE DEVELOPMENT PROCESS 10

IV.1. THE STATIC PART 10

IV.1.1. The Static Dimensions 10

IV.1.2. Relations between Static Planes 12

IV.2. THE DYNAMIC PART 13

IV.3. THE WELTANSCHAUUNG OF THE META-MODEL 13

V. CONCLUSIONS 14

REFERENCES 14

ACKNOWLEDGEMENTS

Research reported herein is part of an on-going research project. Partial financial support from British and Brazilian agencies is gratefully acknowledged, as are the hospitality and support of the Dept. of Computing, Imperial College of Science, Technology and Medicine, and the Dept. of Informatics, Pontifícia Universidade Católica do Rio de Janeiro. Helpful conversations with Professor M. M. Lehman are gratefully acknowledged.

observable level a binary relation, like **A**, may be regarded as consisting of infinitely many atomic sentences of the form $\delta \mathbf{A} \rho$. Hence,

> establishing a property of A or a relationship between A and $\mathbf{m}_p$ or any other object, involves inspecting each one of the infinitely many atomic sentences of the form $\delta \mathbf{A} \rho$. (#)

Since we are regarding both the application and virtual machine as sets of I/O pairs, we can clarify the correctness relation between them. We shall say that

> virtual machine $\mathbf{m}_p$ is correct with respect to application **A** if and only if it halts for every appropriate datum (belonging to the domain of the application A) and the I/O relation realised by $\mathbf{m}_p$ is included in the I/O relation corresponding to A (+)

One may distinguish between the observable plane, with applications and virtual machines, and the plane of the I/O relations, with the denotations of specifications and of programs. The latter, called the *C-observable plane*, is an abstraction of the former and they are connected by *correspondence rules* C [Sup77]. But, for simplicity, we shall often neglect to distinguish between the objects x and C(x). Likewise, we shall generally not make explicit the relationships connecting data and results for the application and the program.

III.1.2. The Program Text Level

A *program* (text) *p* is a syntactic object written in a formal language, with precise formation rules. We thus introduce the *executable* (*linguistic*) *plane* consisting of these syntactic objects, which are rendered executable by interpretation by the target machine at hand.

The virtual machine $\mathbf{m}_p$ is the device obtained by interpreting a program *p* on the given target machine. This connection between *p* and $\mathbf{m}_p$ is established by an *interpreter* H, which is a part of the correspondence rules enabling one to establish connections between properties of *p* and the observable behaviour of $\mathbf{m}_p$. We can express this by $\mathsf{H}[p] = \mathbf{m}_p$, which is, by definition, what we mean by "virtual machine generated by a program text; correctness of the interpreter is not an issue.

III.1.3. The Fundamental Relation

We can try to formalise the notion of correctness in III.1.1 as a precise definition, by translating it into formal logic [H+V90]. For this purpose, it is convenient to introduce some notation related to the actors of II.2. We have been using

A for the application and

$\mathbf{m}_p$ for the virtual machine generated by program *p*.

Now, let us denote by

$\delta \mathbf{A} \rho$ that the pair $\langle \delta, \rho \rangle$ belongs to application **A**;

$\delta \mathcal{D} \mathbf{A}$ that δ is a possible datum of application **A**, i. e., δ belongs to the domain of **A**;

$\mathbf{m}_p \mathcal{A} \delta$ the fact that $\mathbf{m}_p$ has been fed datum δ;

$\mathcal{H}(\mathbf{m}_p, t, \delta)$ the fact that $\mathbf{m}_p$, when fed δ, halts at instant t; and

$\mathcal{R}(\mathbf{m}_p, t, \delta)$ the result produced at instant t by $\mathbf{m}_p$ applied to δ.

We can now formalise explanation (+) as

$$m_p \angle A \leftrightarrow \forall\delta\ (\ \delta\mathcal{D}A \wedge m_p\mathcal{A}\delta \rightarrow \exists t\ (\ \mathcal{H}(m_p, t, \delta) \wedge \delta A\mathcal{R}(m_p, t, \delta)\ )\ ) \qquad (1)$$

But, definition (1) presents two fundamental problems, to be discussed in the sequel. The very acceptance of expression (1) as definition of correctness allows us to show the need of some "common places" of Software Engineering, for instance, the interpolation of a theoretical object in the sdp, thereby recognising two - non-independent - phases [H+V90]

III.2. Formal Reasons for more Levels

We shall now discuss the problems presented by definition (1), for whose solution one is forced into the explicit adoption of an appropriate world view, leading naturally to the need for theoretical levels [H+V90].

III.2.1. The Need for Theoretical Levels

The first problem concerning definition (1) has to do with decidability on the (C-)observable level. The need for theoretical levels arises from an analysis of the expression $\exists t\ (\ \mathcal{H}(m_p, t, \delta) \wedge \delta A\mathcal{R}(m_p, t, \delta)\ )$.

Due to the quantification $\exists t$, the above expression is not decidable on the (C-)observable level, being non-refutable in principle [Sup77]. In intuitive terms, the problem lies in refuting, on the basis of observations, the eventual halting of m_p: if m_p, when fed δ, has not halted yet, how can one know that it will not halt after a few instants more? Also, m_p will produce an output only if, and when, it has halted.

In order to overcome this problem, we can use the fact that m_p is the virtual machine "generated", via H , by program *p*, the latter being a formal object whose termination is amenable to proof. If we prove $\mathcal{T}(p)$ (termination of *p*), then we know that m_p will halt. So we can Skolemise $\exists t\ \mathcal{H}(m_p, t, \delta)$ and introduce the functional symbol ζ so that $\zeta(\delta)$ denotes "an instant when the machine, after being fed datum δ, has halted". Thus, we can write:

$$\mathcal{T}(p) \vdash_H m_p \angle A \leftrightarrow \qquad (2)$$
$$\forall\delta\ (\ \delta\mathcal{D}A \wedge m_p\mathcal{A}\delta \rightarrow (\ \mathcal{H}(m_p, \zeta(\delta), \delta) \wedge \delta A\mathcal{R}(m_p, \zeta(\delta), \delta)\ )\ )$$

So, the problem of non-refutability in principle can be overcome by means of a formal proof, which ensures the eventual halting of m_p. But, the proof of termination of *p* relies on theoretical reasoning about the syntactic object *p*. This formal proof takes place on a theoretical level, rather than on the (C-)observable one. Hence, overcoming the problem of non-refutability in principle leads us naturally to the need for theoretical levels.

Upon a closer look, one realises that one actually establishes that *p* terminates over a precondition ϕ: one proves that *p* terminates for all data satisfying ϕ [B+W82]. And such a precondition comes from a specification *Spc*. Such a (formal) *specification* is a declarative text, written in a specification language with precise syntax and semantics, like first-order logic. We thus introduce a *declarative*

(*linguistic*) *plane*, containing these specifications.

Both p and *Spc* are linguistic objects, but they should be distinguished because of their intended interpretation. For, p is supposed to be executable, being intended to compute outputs from inputs, whereas *Spc* is intended to denote a relation between inputs and outputs.

III.2.2. The Need for Transversal Planes

The second problem concerning definition (1) is connected with the so-called *operational definitions* [Sup77]. We shall see this will lead naturally to the need for new planes.

Expression (1), being based on observing the behaviour of the virtual machine when fed appropriate data, is a typical example of an operational definition. At first sight this might appear unproblematic, because one tends to think of the verbs involved in the subjunctive mood: "if one were to take a datum δ and feed machine $\mathbf{m}_p$, then.....". But, if the connective $\rightarrow$ in (1) represents the material conditional, then the verbs are to be thought of in the indicative mood. In this case one will face the problem encountered by Carnap in trying to define dispositional concepts in this manner: any virtual machine that is fed no datum whatsoever is correct with respect to an application.

Carnap tried to solve this problem, within extensional logic, by replacing operational definitions of the form $Q \leftrightarrow (P \rightarrow R)$ by reduction sentences (conditional definitions) of the form $P \rightarrow (Q \leftrightarrow R)$. Unfortunately, Carnap's proposal presents some weak points and does not actually solve the problem of counterfactual conditionals [Sup77].

Thus, the logic of the correctness relations on the (C-)observable level must be a conditional, rather than classical, one [Hae91]. We shall represent the counterfactual conditional by $\Rightarrow\!\!\!\rightarrow$ in order to distinguish it from the material conditional $\rightarrow$. So, (1) shall be rewritten with $\Rightarrow\!\!\!\rightarrow$ in lieu of $\rightarrow$, to stress its reading in the subjunctive, rather than indicative, mood.

Thus, a new, transversal plane, for *the logic of counterfactuals*, appears. In fact, the need for another transversal plane, had already been foreshadowed. Indeed, in III.2.1 we saw that termination is a relation between syntactic objects, but on distinct planes. In fact, termination is part of total correctness [B+W82], and we shall denote the latter by $p \sqsubseteq Spc$ (p is an *implementation* of *Spc*). We, thus, need a further plane for reasoning about the connection between p and *Spc*, a *plane for the logic of program verification.*

III.2.3. The Need for The Plane of I/O Relations

The PW model [Leh84].of the sdp splits it into two legs: abstraction (from **A** to *Spc*) and reification (from *Spc* to p) Of course, the reification process can be carried out in a stepwise manner (see III.3.1). But, as already observed by Turski,

> the *verification* that a "more concrete" specification is an implementation of a "more abstract" one is not sufficient for the acceptance of the former as a basis for a new development step; in any case, after the formal *correctness proof*, one still needs an *acceptance test* with respect to the application . (*)

A formal justification for this factorisation into two legs can be provided [H+V90]. If we use $Spc \Leftarrow \mathbf{A}$ for "*Spc* is a (correct) specification for **A**", we can show

$$\mathcal{T}(p) \vdash_H \mathcal{S}pc \Leftarrow \mathbf{A} \wedge p \sqsubseteq \mathcal{S}pc \rightarrow \mathbf{m}_p \angle \mathbf{A} \tag{3}$$

Nevertheless, $p \sqsubseteq \mathcal{S}pc$ and $\mathcal{S}pc \Leftarrow \mathbf{A}$ in (3) are not on the same level, for $p \sqsubseteq \mathcal{S}pc$ relates two linguistic objects, albeit of distinct kinds, whereas $\mathcal{S}pc \Leftarrow \mathbf{A}$ relates a linguistic object to an observable one. According to (#), the only way to study the correctness of $\mathcal{S}pc$ with respect to **A** is by means of validations (tests). For these tests one needs to refer to the I/O behaviour of A, on the plane of I/O relations.

A precise definition for $\mathcal{S}pc \Leftarrow \mathbf{A}$ would involve a universal quantifier of the form $(\forall\delta)(\mathcal{D}\mathbf{A}\delta \rightarrow ...)$ as in (2). Now, except in the rare cases where $\mathcal{D}om(\mathbf{A})$ is finite, it is not finitely examinable, and $(\forall\delta)(\mathcal{D}\mathbf{A}\delta \rightarrow ...)$ does not terminate in finite time. Thus, in generating tests one replaces $(\forall\delta)(\delta\in \mathcal{D}om(\mathbf{A}) \rightarrow ...)$ by $(\forall\delta)(\delta\in \mathcal{W} \rightarrow ...)$, where $\mathcal{W}$ is a finite subset of $\mathcal{D}om(\mathbf{A})$. Then, the implication $\leftarrow$ of the biconditional in (2) presupposes an induction, which introduces non-monotonicity into the sdp. So, from the Weltanschauung we were able to derive the formal need of Turski's assertion (*) about the non-independence of the factorisation.

III.3. Heuristic Convenience of the Multiplicity of Levels

Software development can proceed within the theoretical levels, as in cases modeled by LST (see III.3.1), or on various levels, as in prototyping paradigms (see III.3.2). We have already discussed formal needs for the introduction of several levels. We shall now consider heuristic convenience and epistemological reasons, which, in modeling, can be as important as formal needs.

III.3.1. Development within Theoretical Levels

Some program construction paradigms, like the transformational method of Bauer and Wössner [B+W82], are based on linguistic transformations. The LST model of the reification process [Leh84] is particularly appropriate for such cases.

According to the LST model, one tries to obtain a series of "canonical (development) steps" - implementations of specifications - until a level that is directly supported by the target machine of the process. It is based on a theoretical explication of (part of) the reification process [VMS85; T+M87]. The central ideas are regarding a specification as a presentation of a (logical) theory and an implementation as an interpretation into a conservative extension, which yield the compositionality of implementations.

Clearly, several models, like those based on the refinement of an operational specification [Agr86], can be easily modeled by the LST model. Also, in the CIP method [B+W82], both the objects and the transformations themselves are already of linguistic nature.

In principle, one could model several paradigms on the basis of the LST model by placing all the objects on a linguistic level and viewing transformations as canonical steps. But then, one would be confounding a program specification with the program itself. Failure to distinguish between these objects, and their logics, would lead to logical reductionism; and one should not base modeling on reductionistic explanations.

III.3.2. Passage between Levels

One could, in principle, model prototyping paradigms via a linguistic (meta-)model. But any explication based on the idea of transformations between linguistic theories is clearly biased.

Let us examine first the classical prototyping paradigm [Agr86]. Here, after developing, exercising, correcting and accepting a prototype, a process of refinement proceeds on the same plane until attaining the desired program. In this case one needs two planes, namely an observable plane (containing the application and the virtual machine) and the plane of the prototype. And the process of formal construction will take place on the latter plane, rather than on a linguistic plane, which is the only one offered by linguistic (meta-)models, like LST and PW.

In prototyping, the central concept is that of “simulation”. A prototype is an object that behaves as the virtual machine one is trying to build, a crucial characteristic being that of iconic model: paraphrase of the behaviour of the object being modeled [Sup77]. So, in developing a prototype one has an I/O relation in mind.

Thus, a prototype can be best regarded as a term over basic I/O "building blocks", rather than a linguistic object. We thus see the convenience of introducing a *plane for terms over binary relations*.

Indeed, consider now the hybrid prototyping paradigm [Agr86]. In this paradigm, after accepting a prototype, one extracts from it a specification *Spc* and continues the process within a transformational paradigm. One sees the convenience of having three planes [Hae91]:

- a linguistic one, where the transformational part of the process is carried out;
- a plane for the I/O relations, containing the denotations of *Spc* and of the program; and
- a plane for the prototype, which is a term over basic I/O "building blocks".

So, if one insists on modeling prototyping paradigms by linguistic (meta-)models, then one would be forced to view a prototype as an a axiomatic theory, which would mask the central characteristics of the prototype in favour of a formal theory of the behaviour of the application. And it is exactly the difficulty in constructing such a theory that leads one to building prototypes in the first place. A serious drawback of this manner of proceeding is that it would provide only a reductionistic logical explanation (without preserving the structure of the object being modeled), rather than an epistemological explication, which would be much more enlightening [Sup77].

III.3.3. Tests, Proofs, Derivations, and Actions

Software development often proceeds in a stepwise manner from one version to the next one. Each such refinement step can be based on heuristics or calculi. When relying on heuristics, once obtained the new version, one must ascertain its acceptability, whether by formal proofs (verification) or by means of tests (validation).

Reliance on heuristics + tests is probably most common in the so-called "handcraft" approach to software development. Some software development methods (like M. Jackson's method) do provide a set of rules for helping in this process, but are not strong enough for supporting a proof by construction of the satisfaction of the correctness relation.

The usage of heuristics + proofs is usual with methods based on linguistic planes with enough power for supporting the development of correctness proofs, but without a calculus for developing

proofs by construction. This case is typical of the software development methods based on heuristics but including obligations of verification.

For instance, in the hybrid prototyping paradigm [Agr86], one does not generally have a calculus for constructing a formal specification from the prototype. So, the a-posteriori acceptance of the specification must be based on verification, if there exists an appropriate formal language, or on validation.

When one can count on a calculus, one can, and perhaps should, still use the strategy heuristics + proofs, a simple, but pervasive, example being the introduction of eurekas in calculi. Nevertheless, this usage should be reduced to a minimum: the greater part of the heuristics is formalised and most proofs are by construction.

A calculus, for the linguistic or the transversal planes, provides still other benefits. For, one can specify (or model) development methods by relying on the formal manipulations of these calculi. For instance, assume that one has a calculus of binary relations with sufficient expressive and manipulative powers. Then, one can develop a software method based on the hybrid prototyping paradigm, in which the plane of linguistic development is represented by the CIPL language [B+W82] and the plane of the prototype by this calculus. In this case, one can pass from the prototype to the specification by means of calculations, much in the same manner as in the development of programs in CIP. Of course, this possibility hinges on some assumptions about the calculus, which are satisfied in our example meta-model (see IV.1).

We will now briefly discuss how one can ascertain the satisfaction of some of the correctness relations between objects of the sdp, as well as the role of actions in it.

The relation $p \sqsubseteq \mathcal{S}pc$ is one of interpretation between specifications (theory presentations), perhaps in distinct logics, and its satisfaction can be ascertained in the meta-language (see III.2.2 and III.3.1)

A calculus with sufficient expressive and manipulative powers provides two possible approaches. On the one hand one can directly prove the satisfaction of the correctness relation between the objects, on the other hand one can construct a new object from the given one so as to satisfy correctness relation.

In contrast (see III.2.3), the "verification" of $\mathbf{m}_p \angle \mathbf{A}$ involves an induction on atomic sentences for a finite subset of $\mathcal{D}om(\mathrm{A})$. Hence, such "verification" is not a real proof in the usual sense. It is rather an experimental test for which the world view selected should provide a procedure, based on the hypothetico-deductive model [Sup77; H+V90], or on "bootstrapping" [Hae91], etc. But, such tests involve actions.

In III.1.3 we have already seen an action, of feeding virtual machine $\mathbf{m}_p$ with application datum δ (represented by $\mathbf{m}_p \mathcal{A} \delta$), which is inescapable in modeling the sdp. This is due to the expressive power and form of proof of the (C-)observable level (see (#) in III.1.1).

But, action $\mathbf{m}_p \mathcal{A} \delta$ is not the only one to be taken into account in the sdp. This action is just an "obligation" in the "proof" of $\mathbf{m}_p \angle \mathbf{A}$. Now, the proof itself is an obligation and, hence, presupposes an action, much as any other verification or validation obligation in software

development. Thus, "developing p from Spc", "verifying $p \sqsubseteq Spc$" and "testing p with respect to **A**" are examples of actions that should be modeled.

Hence, we need formalisms for reasoning about actions. We would like to deal with tests within a formal framework similar to that of proofs proper. Since tests involve actions, a condition for the coherence of a model of the sdp is the inclusion of a dynamic dimension that allows one to talk, and reason, about actions in a precise manner, much as one deals with formal proofs.

III.4. THE BASIC NEEDS IN MODELING THE SOFTWARE DEVELOPMENT PROCESS

We are interested in a **theoretical basis** and a **unifying conceptual framework** that permit combining software methods and paradigms into a **coherent process model.** We have seen that such a framework requires several levels, as well as formalisms for relating objects of distinct planes and supporting reasoning about actions, which enable it to place itself in a position "meta" with respect to the processes.

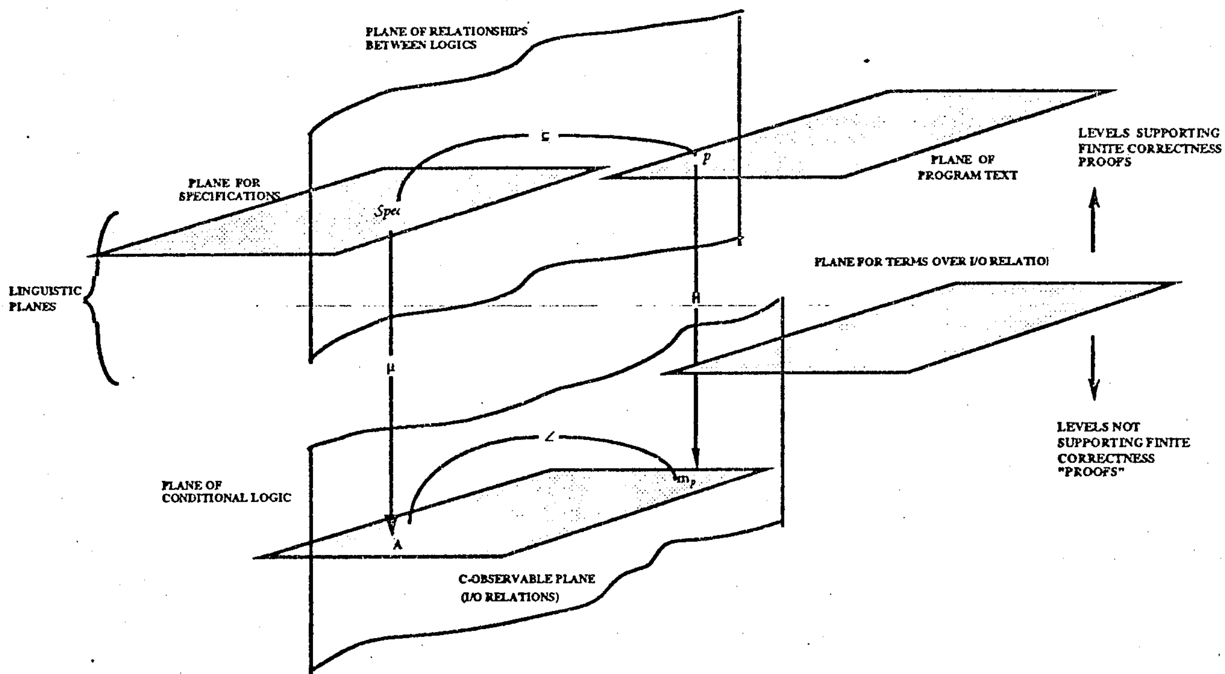

Figure 1: Multiple levels

Our argument, which we shall stress in the sequel, on the basis of results form ongoing research, is that there are are some minimal requirements for this. In particular, one should have the possibility of including, as indicated in figure 1, the following basic planes:

- a (C-)observable plane of I/O relations(see III.1.1 and III.2.3),
- a plane for the program text (see III.1.2 and III.2.1),
- a plane for formal specifications (see III.2.1),
- a plane for terms over I/O relations (see III.3.2);

as well as some transversal planes (see III.2.2):

- a plane for relationships between logics: a logic of program verification (see III.2.1).

- a plane of a logic for the counterfactual conditional (see III.3.3);

In this section we have discussed some basic needs in modeling the sdp, which led to the introduction of several levels. These needs stem from distinct sources, namely formal reasons, epistemological considerations and heuristic convenience. We have seen the importance of placing the various objects on distinct planes and of creating new planes for accommodating the transformations between objects on distinct planes. In the next section we shall outline a multidimensional meta-model of the sdp, under elaboration, which is based on these general ideas.

IV. A Multidimensional (Meta-)Model of the Software Development Process

As a concrete exemplification for our general considerations, we shall now briefly present a multidimensional meta-model of the sdp that is under elaboration. This meta-model, which is centred around general concepts of interpretations between theories [VMS85] and an extended calculus of binary relations [H+V91; V+H91], can be naturally divided into into two parts: a static and a dynamic one. The former will be concerned with the objects involved in the process and the correctness relations, whereas the latter will concentrate on the actions along the process (see III.3.3). Within the static part one establishes, for instance, the obligation of satisfying a certain correctness relations, without worrying about how this is carried out (by construction, verification, etc). The dynamic part will be concerned with the prescription, within MAL [Kho88] of how such a "proof" is to be carried out.

IV.1. The Static Part

The static part of this meta-model consists of the observable and theoretical levels together with the relationships among planes.

IV.1.1. The Static Dimensions.

The observable level comprises applications and virtual machines (or their abstractions as I/O relations: the C-observable level). On the theoretical level one finds programs and specifications, which are formal objects one can reason about without having to deal with infinite objects. This two-level reasoning is inevitable (see III.4), if one is to construct an object that is to satisfy $\mathbf{m}_p \angle \mathbf{A}$.

The observable level will be represented by the C-observable plane of I/O relations. This will provide a semantics for the various linguistic planes of the theoretical level, as well as a semantic "hinge" for connecting planes (see IV.2.2).

The theoretical level consists of four linguistic planes:

- the plane of the programming language,
- the plane of first-order logic,
- the plane of the extended calculus of binary relations,
- the plane of the n-ary relations.

in addition to the transversal planes (see figure 2)

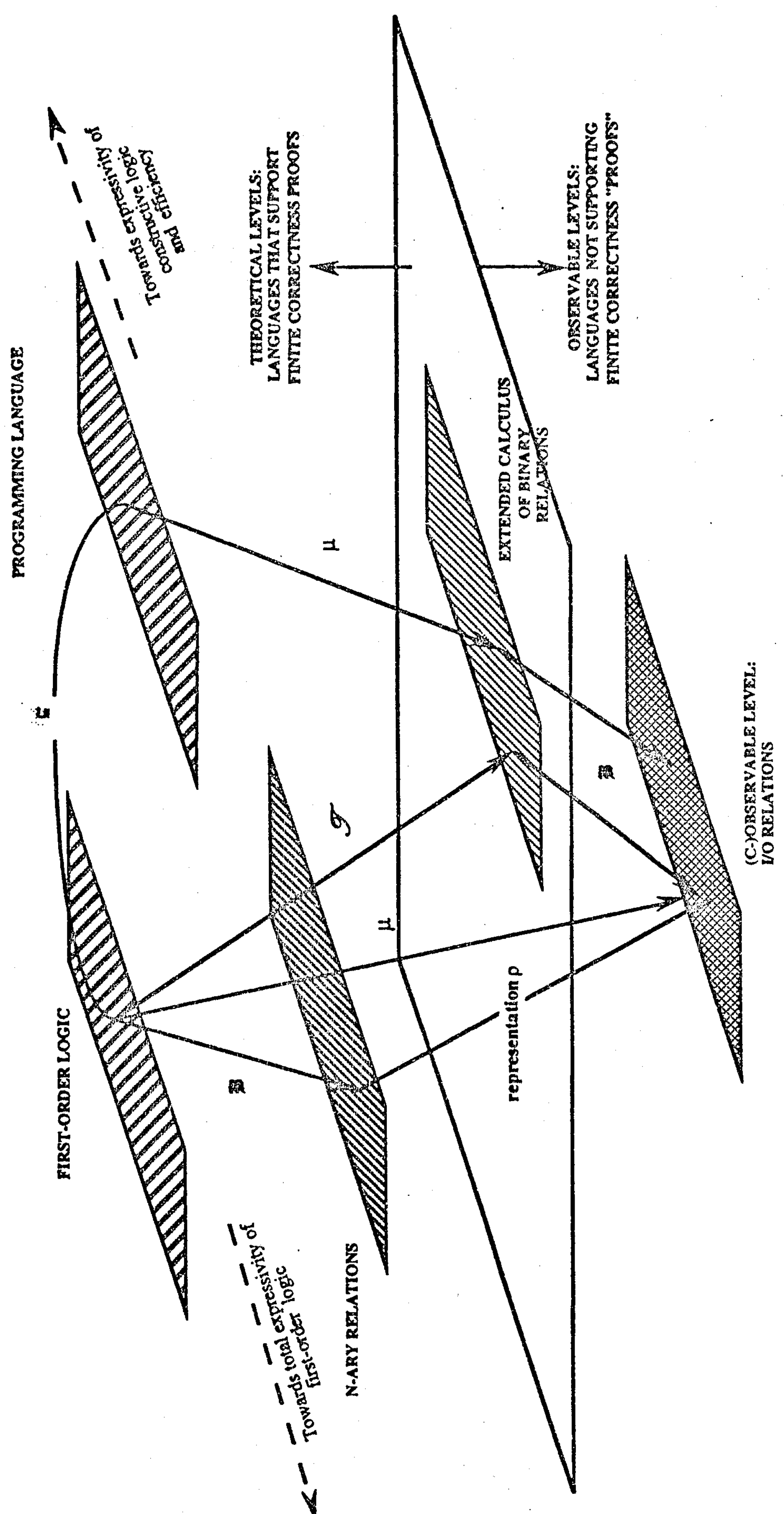

Figure 2: Example meta-model

We have already argued for the need of the plane of the programming language in III.1.2 and

III.2.1. So let us examine the remaining ones.

A rationale for the plane of first-order logic is allowing one to accommodate the logical theory of data types [VMS85; T+M87], which provides the theoretical basis for the LST model of the sdp.

The plane of the extended calculus of binary relations comprises a calculus of (partial) binary relations [Hae91; H+V90, 91; V+H91] developed by extending Tarski's calculus of binary relations [Tar41] by a few finitary operations, which are natural because the universe is now structured. An advantage of this plane is its enabling one to regard a prototype as a term over I/O relations (see III.3.2); for then the refinement of a prototype may become a process of calculation. The standard denotation of a term over binary relations is a set of I/O pairs, as those on the C-observable plane, and the preservation of the I/O concept throughout algebraic manipulations renders this calculus very adequate for capturing the idea of iconic models in prototyping paradigms.

The plane of the n-ary relations provides the usual semantics for first-order logic as well as for logic programming languages (like Prolog). Thus, in order to cover paradigms involving logic programming, we include this plane. Another reason for this plane is our desire of covering Möller's calculus of (higher-order) n-ary relations [Möl91], as well as Codd's algebra of n-ary relations; the former for exploring hybrid software paradigms and the latter for investigating paradigms involving relational data bases.

IV.1.2. Relations between Static Planes.

If one wishes to model hybrid software paradigms, involving distinct languages and logics, then one must establish formal relationships among the objects on the various planes. In our case, we should analyse figure 3, which is a "transversal section" of figure 2.

In figure 3 we have:

a first-order formula φ, with n free variables,

the n-ary relation $\varphi^{\mathfrak{B}}$ defined by formula φ on structure $\mathfrak{B}$,

the denotation $\mu[\varphi]$ of formula φ on structure $\mathfrak{B}$: the binary relation (set of I/O pairs) which is the representation $\rho(\varphi^{\mathfrak{B}})$ of $\varphi^{\mathfrak{B}}$;

a term t over binary relations,

the denotation $\mu[\mathsf{t}]$ of term t on structure $\mathfrak{B}$, an I/O relation.

In the extended calculus of binary relations one can express quantifiers by terms. This has two important consequences:

the quantifiers become decomposable, which is of importance in program derivation [H+V91];

a translation $\mathcal{T}$ of formulae into terms is constructed [V+H91] so that $\mathcal{T}(\varphi)$ is a term denoting $\mu[\varphi]$.

Thus, the diagram in figure 3 commutes, which allows us to deal with correctness relations on and between distinct planes. For instance, on the the plane of the extended calculus of binary relations: $\mathsf{t}_1 \leftarrow\!\bullet\, \mathsf{t}_2$ iff $\mathsf{t}_2 \subseteq \mathsf{t}_1$ and $\mathcal{Dom}(\mathsf{t}_2) \subseteq \mathcal{Dom}(\mathsf{t}_1)$ [H+V91]. Moreover, this enables the analysis of transversal planes, like the one in figure 3, as well as the definition of functions, relations, restrictions and diagrams within such planes, which shows that our meta-model can place itself in a position "meta" with respect to the processes, as required in III.4.

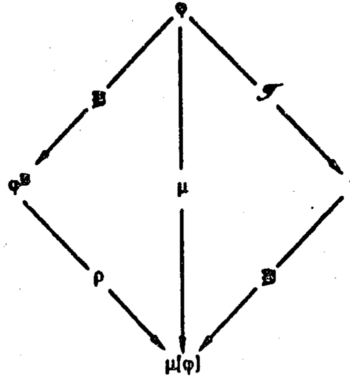

Figure 3: Translation of formulae into terms

IV.2. The Dynamic Part

We need formalisms for reasoning about the counterfactual conditional ⇻ and about the actions occurring in the sdp, in order to to deal with them in a precise manner. Given the restricted nature of our universe of discourse, where cotenability [Nut83] is not relevant (see (#) in III.1.1), the counterfactual character of these conditionals may perhaps be captured by a logic that is more studied and established than that of the conditionals

Let us consider a semantics of similarity of worlds for these conditionals [Nut83]. In our restricted universe, it is reasonable to expect that the counterfactual character is due merely to the fact that, in evaluating the conditional only in the real world, some actions (like $\mathbf{m}_p\mathcal{A}\delta$) may have had no effect. Then, it suffices to evaluate it also in the worlds similar to the real one where these actions have already been realised. Thus, we can envisage that the counterfactual character of our conditional ⇻ may be captured by a modal logic of actions, like MAL [Kho88]. Hence, we can rewrite (1) as:

$$\mathbf{m}_p \angle \mathrm{A} \leftrightarrow \forall\delta\ (\ \delta\mathcal{D}\mathrm{A} \rightarrow [\mathbf{m}_p\mathcal{A}\delta]\ \exists t\ (\ \mathcal{H}(\mathbf{m}_p, t, \delta) \wedge \delta\mathrm{A}\,\mathcal{R}(\mathbf{m}_p, t, \delta)\ )\ )$$

Moreover, the same formalism will enable us to handle actions like the ones mentioned in III.3.3.

IV.3. The Weltanschauung of the Meta-Model

One of our theses is that the "world view" (Weltanschauung) of models like our example, is in part based on a generalisation of Carnap's verificationistic theory of meaning [Sup77]. The basis for this thesis is the view that the objects that purport to represent software artifacts, on any one of the planes of the theoretical level, should be connected by a "chain" of interpretations, implementations, translations, extensions, etc. The purpose of such a "chain" is the establishment of test procedures of this artifact of software with respect to the application **A**. Thus, such world view, presupposes, at least, a logic of actions and a meta-language, where one can talk about diagrams like the one in figure 14, as well as the C-observable plane of the I/O relations.

The meta-language should be formalised in order to provide a formalism with which one can reason about proofs, tests, interpretations, implementations, and manipulations of theories in general.

Then we shall be heading towards the establishment of a general framework for the construction of models like the one exemplified here. Also, we shall then have a general concept of coherence, which may serve as a bench-mark for the coherence of particular models of the sdp.

V. CONCLUSIONS

We have analysed some basic needs of a **theoretical basis** and a **unifying conceptual framework**, which are essential for **coherent meta-models** for the software development process. They arise from formal reasons (stemming from a precise analysis of the goal of software development), epistemological considerations (related to non-reductionistic explications) and heuristic convenience (connected to instantiating a meta-model to cover various paradigms).

These needs lead to several levels, for distinct objects and for relating them, as well as to a formalised meta-language, where one can reason about proofs, tests, interpretations, implementations, and manipulations of theories in general. This analysis also provides a general concept of coherence, which may serve as a bench-mark for the coherence of particular models of the software development process.

We have also outlined a multidimensional meta-model of the software development process, based on these general ideas and centred around general concepts of interpretations between theories [VMS85], an extended calculus of binary relations [H+V91; V+H91] and a modal logic of actions [Kho88]. This illustrates how the various formal and practical tools, as well as the various models, methods and paradigms of programming could be organised into a coherent process, which is essential for the design of **integrated programming support environments.**

REFERENCES

[Agr86] Agresti, W. (ed) - New Paradigms for Software Development. IEEE Comp. Soc. Press, 1986.

[B+W82] Bauer, F. L.; Wössner, H. - Algorithmic Language and Program Development. Springer Verlag, Berlin, 1982.

[Hae91] Haeberer, A. M. .- Fundamentos para um Metamodelo Descritivo e Prescritivo do Processo de Desenvolvimento de Software. D. Sc. diss. , Pont. Universidade Católica, Rio de Janeiro, 1991.

[H+V90] Haeberer, A. M. ; Veloso , P. A. S. - Why Software Development is inherently non-monotonic: a formal justification. Trappl, R. (ed.) Cybernetics and Systems Research, World Scientific Publ. Corp., London, p. 51-58 , 1990.

[H+V91] Haeberer, A. M., Veloso P. A. S. - Partial Relations for Program Derivation: adequacy, inevitability and expressiveness. Möller, B. (ed.) Constructing Programs from Specifications; North-Holland, Amsterdam, p. 319-371, 1991.

[Kho88] Khosla, S. - System Specification: a deontic approach. PhD diss., Imperial College of Science, Technology and Medicine, Dept. Computing, London, 1988.

[Leh84] Lehman, M., M. - A Further Model of Coherent Programming Process. Proc. Software Process Workshop, IEEE Comp. Soc. Press, Feb. 1984.

[Möl91] Möller, B. - Relations as a Program Development Language. Möller, B. (ed.) Constructing Programs from Specifications; North-Holland, Amsterdam, p. 373-397, 1991

[Nut83] Nute, D. - Conditional Logic. Gabbay, D.; Guenthner, F. (eds) Handbook of Philosophical Logic, Vol. II, p. 387-439. Reidel, 1983.

[Sup77] Suppe, F. - The Structure of Scientific Theories. University of Illinois Press, Urbana, 1977.

[Tar41] Tarski, A. - On the Calculus of Relations. Journal of Symb. Logic, 6(3), p. 73-89, 1941.

[T+M87] Turski, W., M.; Maibaum, T., S., E. - The Specification of Computer Programs. Addison-Wesley, Wokingham, 1987.

[V+H91] Veloso , P. A. S.; Haeberer, A. M. - A Finitary Relational Algebra for Classical First-Order Logic. Bull. Section on Logic (Polish Acad. Sciences), 20(2), p. 52-62, Jun. 1991.

[VMS85] Veloso, P. A. S., Maibaum, T. S. E., Sadler, M. R. - "Programme development and theory manipulation". Proc. Intern. Workshop on Software Specification and Design, London, p. 155-162, Aug. 1985.