THE VERY IDEA OF SOFTWARE DEVELOPMENT ENVIRONMENTS:

A CONCEPTUAL ARCHITECTURE FOR THE ARTS* ENVIRONMENT PARADIGM

Armando M Haeberer♣
and
Thomas S E Maibaum♥

Abstract

During the last three years we have been building an instantiation of a system's development paradigm, called ARTS. The paradigm consists of a view of what a system development environment is, in general terms, and a methodology for instantiating the paradigm for particular and specific domains of application.
The motivation for and the explanation of the paradigm are derived from extant epistemological models of the method of Natural Science. We assert that these models are directly applicable to the domain of software and systems construction, and that, from them, we can derive principles and explanations for what a software development environment should be.
We present a brief description of the Statement View of scientific theories, a conceptual architecture for software development environments whose rationale is given in terms of the Statement View and some examples of how the present instantiation of ARTS realises this conceptual architecture.

1 INTRODUCTION

An environment to support construction of software based systems from specifications, be they requirements specifications or architectural specifications[1], should provide facilities for, amongst others: the manipulation of specifications; their validation; the construction of the initial architectural description of the system from the corresponding requirements specification; the verification of the latter against the former; the identification of components, be it because they exist in "real life", or as an artifact to deal with complexity[2]; the composition of a system from such parts; the interactive refinement of specifications – interactive with the software engineer and, via validation, with real life; the reification of specifications to transform them from descriptions of the components of the original system, in its own domain, into descriptions of software components that "simulate" the original ones; the verification of refinement relations; the construction of programs from detailed specifications; the validation of components and systems of software (testing); and the correction of specifications taking into account the negative results of tests.

If we begin to think about the underlying activities supported by the facilities described in the above paragraph, we see that we have in hand a collection including: *elucidation*, the activity of finding a requirements specification; *illumination*, the activity of finding a constructive architectural specification satisfying the requirements; *systematic observation*, the activity of validating through experiments, by either explaining the behaviours or predicting them; and *calculation* using the underlying formalism, the activity of formal derivation of validation and verification tests, of refinements, etc.

The parallel between these activities and what is informally called 'the scientific method' is too strong to be considered merely a coincidence. The idea of using this method as a sound and systematic foundation for the activity of constructing software systems is further strengthened and well founded when we start thinking, for instance, of the activity of testing. The fact that a positive result of a test does not confirm the correctness of a program, while a negative one refutes it, is obviously a late restatement of the hypothetical-deductive method in its simple

* This projet was partially founded by Siemens Telecomunicações (Brazil).

♣ The Laboratory of Formal Methods. Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro – Marquês de São Vicente 225. 22453. Rio de Janeiro. RJ. Brazil. tel +55 21 512-8325 – fax +55 21 512-8045 – e-mail armando@lmf-di.puc-rio.br

♥ Department of Computing. Imperial College – tel +44 171 594-8274 – fax +44 171 581-8024 – e-mail tsem@doc.ic.ac.uk

[1] For the purposes of this document, "architectural specifications" are those specifications that define not the behaviour of a system, but its structure and, in some appropriate level of granularity, the bahaviour of the parts. Thus, in an architectural specification, the bahaviour of the system is inferred (when possible)from the behaviour of the parts and from the system's structure. See also [Zave and Jackson] for related work on architecture for requirements.

[2] We refer to complexity in the Kolmogorov-Chaitin sense.

version – stated by Newton and refined by Carnap, Hempel and others – which is at the heart of the method of Natural Science and Engineering ([Stegmüller, Suppe]).

The goal of this paper is not to present simply an epistemological discussion on the scientific basis of software engineering or its methodology, nor to justify the reason for the use of a particular epistemological conception as the foundation for the 'conceptual architecture' of a paradigm for software development environments. The objective of this paper is the description of such a 'conceptual architecture' exemplified by one of its realizations currently under development, namely the ARTS paradigm, with some justification in relation to this epistemological and scientific basis.

The paper is organised as follows. In section 2 we outline the progress of ideas in epistemology leading to the so-called Statement View of scientific theories and relate this to concerns of computer science and software engineering. In section 3, we address the ramifications of these ideas for the design of software development environments and illustrate them with specific examples from the environment actually under construction.

2 A LITTLE BIT OF EPISTEMOLOGY WON'T HURT ….

The conceptual architecture of the ARTS environment is based principally on the epistemological model of the method of Natural Science known as the *Statement View* ([Suppe]), or, alternatively, *the two level theory of the language of science*, with some adaptations due to the later work of Mary Hesse and the structuralist programme led by Ulises Moulines and Joseph Sneed, amongst others ([Hesse, Balzer])

The so-called Statement View of scientific theories has been strongly criticized as an explicative model of the method of Natural Science ([Suppe]). In most cases, the critics ask up to what point this model sheds light on the nature of the real methods used by the working scientist for acquiring knowledge, as opposed to providing, a posteriori, an artificial rational reconstruction of such a method. These criticisms vanish when using the Statement View in software engineering either because we are not trying to do epistemology, but rather using its constructs, or because of the particular domain of application to which we refer. For instance, this happens in the case of the criticism known as 'the Putnam challenge' ([Suppe, Putnam]), which will be treated succinctly below. Other criticisms disappear when using versions of the Statement View strongly modified by epistemologies motivated by and post dating the original Statement View, as, for instance, those cited in the previous paragraph.

The conception of the scientific method derived from the discussions and work of the group known as the Vienna Circle. The Statement View of scientific theories arose when Rudolf Carnap, in the 1930's, tried to introduce the so-called *dispositional terms* into the empiricist language of science, which he had been building since the 20's. The empiricist language of science (*LE*) would be a language in which the only syntactically constructable statements would be those with scientific meaning. That is, it should be impossible to construct in *LE* a metaphysical statement (in the sense of Hegel) ([Suppe, Stegmüller]).

When trying to construct the empiricist language, Carnap recognised two kinds of terms denoting their corresponding types of concepts or properties: the so called *plain* terms, such as red, blue, cold, or hot, which are intersubjectively and immediately understood, and the so-called *dispositional* terms that require systematic observation in order to be recognised intersubjectively.

In Computer Science, terms such as 'a machine halts on input *i*' (in program construction) or 'algorithm such and such takes time *t* on input *I*' (concrete complexity) are dispositions. Indeed, these dispositions are only apprehended through systematic observation. As we will see below, the problem is not that both these statements are 'observationally undecidable'. Clearly the halting problem is observationally undecidable, but checking the time taken for execution of the program is decidable. The problem arises when we use such dispositions to conclude that a deleted program always halts on any input and has whatever algorithmic complexity we might wish! Notwithstanding the available examples in software engineering, for explaining the question of the introduction of dispositions in the language of science, we will use an example

from another area; otherwise, we would be risking falling into circularity by exemplifying with the intended object of discourse.

To illustrate what *is* a disposition and the problem of introducing dispositional terms into the language of science, we will try to define 'solubility in water of the object *x*'. Let *Dx* be an abbreviation for '*x* is soluble in water', *Axt* an abbreviation for '*x* is introduced into the water at time *t*', and *Sxt* be an abbreviation for '*x* dissolves in the water at instant *t*'[3]. The permanent disposition '*x* is soluble in water' must be definitionally equivalent to 'whenever *x* is introduced into water, it dissolves'. In other words, the following definition should hold[4]:

$$Dx \leftrightarrow (\forall t)(Axt \rightarrow Sxt) \qquad (1)$$

The problem is that this definition does not represent the intuitive meaning of the dispositional predicate 'soluble in water'.

Suppose we introduce the object *a* into water to verify its solubility, i.e., Aat_0 is true. There are two possibilities.

The first is that *a* dissolves, and, therefore, Sat_0 will also be true. If we assume that a series of such experiments at the instants of time $t_0,\cdots,t_n$ provide a 'sufficient basis' $\bigwedge_{i=0}^{n}\{Aat_i \wedge Sat_i\}$ for inducing $(\forall t)(Aat \rightarrow Sat)$, then, according to definition (1), *Da* will also be true.

The second possibility is that *a* is not dissolved in some experiment, namely in the instant t_j. Thus, Sat_j will be false. Because $(\neg Aat_j \wedge \neg Sat_j) \vdash \neg Da$, in this case definition (1) *proves to be adequate*.

Now, suppose that the substance *a* is not introduced into the water, that is, the solubility of *a* is never put to the test at any moment of its existence. Suppose that *a* is a log that was burnt yesterday without ever having been introduced into water. Then, it is the case that $(\forall t)(\neg Aat)$. Since from $\neg Aat$ it logically follows that $\neg Aat \vee Sat$, and from $(\forall t)(\neg Aat)$ the universal statement $(\forall t)(\neg Aat \vee Sat)$ also follows, the latter being equivalent to $(\forall t)(Aat \rightarrow Sat)$, by definition (1) we have that *Da* holds. In this way, the log that burnt yesterday (i.e., it no longer exists and thus cannot be put to the test) is soluble in water ([Stegmüller]) !

As a consequence, *while the formal definition of solubility in water does not appear to put in place too narrow a concept, in contrast, the concept introduced by the definition is manifestly too wide*.

Definition (1) is of the kind known as an *operational definition*. The problem with operational definitions is the use of the material conditional (because what we actually want to say is 'if I were to introduce this object into the water, then …'). A potential approach could be to use a subjunctive conditional, i.e., a counterfactual. The problem is that conditional logic is not much more than wishful thinking, given the fact that counterfactuals are not truth functional.

In [Carnap36], Carnap tried to solve the problem by introducing *reductive statements.* For example, Carnap transforms the operational definition of solubility in water, formula (1), into the reductive statement[5]:

$$(\forall t)(\forall x)\big(Axt \rightarrow (Dx \leftrightarrow Sxt)\big) \qquad (2)$$

This is known as a bilateral reductive statement. From the logical point of view, a reductive statement is nothing but a conditional definition. Nevertheless, from the philosophical point of

[3] Note that we are abstracting away the delay between introducing an object into water and the actual time taken for it to dissolve.
[4] Note that the definition of the 'correctness-like' relation (4) (below) for the executions of a program *p* on some computer *H* realising a specification *Spc* is exactly of this form.
[5] Note that the corresponding version of definition (4) (below) would be $(\forall \delta)\Big(F\delta m_{pH} t_0 \rightarrow m_{pH} \leq Spc \leftrightarrow (\exists t)(\exists \rho)\big(t_0 \leq t \wedge H\delta m_{pH} t \wedge \delta\ m_{pH} \rho \wedge \langle \delta, \rho \rangle \in \mu[Spc]\big)\Big)$.

view, the difference between (1) and (2) is enormous, given that (2) is a partial axiomatisation of a dispositional concept ([Stegmüller]).

The introduction of reductive statements such as (2) resolves the inadequacy mentioned above. In case *Aat* is false at some time *t*, we simply cannot consider the biconditional $Dx \leftrightarrow Sxt$, which defines solubility in water in the case in which the object *a* has been introduced into the water, i.e., in the case in which *Aat* is true. In this way, the concept of solubility will remain undefined for those objects that are never submerged in water.

However, the concept of the solubility *Dx* is not necessarily substitutable in all cases by its definiens *Sxt*, as required by the criterion of eliminability ([Shoenfield]).

On the other hand, Carnap encountered a philosophical problem in the use of reductive statements ([Carnap50]). Consider the bilateral reductive statement $Ba \wedge \neg Ba$. If the experimental procedure described by this statement establishes a negative result for an object *a*, this signifies that, according to the observations, we must attribute to the object *a* the predicate $Ba \wedge \neg Ra$. From $Ba \rightarrow (Da \leftrightarrow Ra)$ then follows, purely logically, that $\neg Da$. But, in reality, it is frequently the case that a researcher detects the negative result of an experiment and, nevertheless, affirms that the predicate *Ra* still holds. This is because, for instance, the experimental procedure in question is not absolutely secure, as it is only valid under the assumption that there are no perturbing factors present. Thus, from the point of view of the researcher, the operational procedure must be understood with the proviso of an exception clause. If we consider as valid the existence of such exception clauses, we must accept that reductive statements are completely inadequate[6].
Hence, Carnap abandoned this way of resolving the problem of the introduction of dispositions in the empiricist language of science *LE*. In doing so, he also abandoned the argument for the very existence of *LE*.

2.1 THE TWO LEVEL THEORY OF THE LANGUAGE OF SCIENCE

The new proposal of Carnap, which came to be called the Statement View of scientific theories ([Suppe, Stegmüller, Carnap56]), or the theory of the two levels of the language of science, consisted of abandoning *LE* and introducing two languages in its stead. One, the equivalent of *LE*, is that which we will call below the observational language (*LO*), understandable on its own. The other, which we call the theoretical language (*LT*), in which we formulate a scientific theory *T*, is not understandable on its own, nor completely, if only because its empirical interpretation is only partial. This (empirical and partial) interpretation is given by means of correspondence rules *C*, which link some, but not all, the extra-logical expressions of *LT* to expressions of the observational language. *The dispositional terms must be constructed as theoretical concepts, which absolutely do not appear in the observational language, but only in the theoretical language.*

By means of this transfer from *LO* to *LT*, we can consider what we called above an exception clause. Given *M*, a dispositional concept constructed as a theoretical term, suppose that we deduce an observational consequence *SO* from: certain theoretical hypotheses concerning the presence of *M* (abbreviated as *SM*); some other hypotheses *SK*; some descriptive observational statements, assuming for example conditions which pertain in the laboratory (abbreviated *SO**); and the use of the theory *T* and the correspondence rules *C*. We then obtain the deduction:

$$SM, SK, SO^*, T, C \vdash SO \qquad (3)$$

Suppose now that we do not obtain the expected observational consequence *SO*, i.e., $\neg SO$ holds. From the meta-theoretic statement (3) we can certainly derive the following: $\neg SO, SK, SO^*, T, C \vdash \neg SM$. Nonetheless, in the verification of $\neg SO$, even maintaining theory *T* and correspondence rules *C*, we are not forced to accept $\neg SM$ (as was the case when using

[6] So, what exactly is the moral of this story, so far, for the software engineer? Recalling again Dijkstra's famous dictum about testing only showing the presence of errors, not their absence, we see it needs to be reconsidered. Firstly, we do not know if what we want to observe is sound with respect to our intention and, secondly, we do not know if our testing procedure is faulty or not unless we have a way of reasoning about it.

reductive statements). Instead, we can suppose the falsity of the theoretical auxiliary hypothesis *SK*, or of the empirical hypothesis *SO**, or of both. Under the appropriate circumstances, we will admit the supposition as properly confirmed. This way of taking into consideration exception clauses through the transference of the definition of dispositional terms of the observational language *LO* to the theoretical one *LT* is what is known as *the hypothetical deductive method in complex version* ([Stegmüller]).

The accepted definitive version of the Statement View of scientific theories ([Suppe]), construes them as having a canonical formulation satisfying the following conditions:

1. There is a first-order language *L* (possibly augmented by modal operators) in terms of which the theory is formulated, and a logical calculus *K* defined in terms of *L*.

2. The non-logical or descriptive primitive constants (i.e.,, the "terms") of *L* are bifurcated into two disjoint classes:

   VO, which contains just the observable terms and must contain at least one individual constant;

   VT, which contains the nonobservable or theoretical terms.

3. The language *L* is divided into the following sublanguages, and the calculus *K* is divided into the following subcalculi

   - The observational language, *LO*, is a sublanguage of *L* which contains no quantifiers or modalities, and contains the terms of *VO* but not from *VT*. The associated calculus *KO* is the restriction of *K* to *LO* and must be such that any non-*VO* terms (i.e.,, nonprimitive terms) in *LO* are explicitly defined in *KO*; furthermore, *KO* must admit of at least one finite model.

   - The logically extended observational language, *LO'*, contains no *VT* terms and may be regarded as being formed from *LO* by adding the quantifiers, modalities, and so on, of *L*. Its associated calculus *KO'* is the restriction of *K* to *LO'*.

   - The theoretical language, *LT*, is that sublanguage of *L* which does not contain *VO* terms; its associated calculus, *KT*, is the restriction of *K* to *LT*.

   These sublanguages together do not exhaust *L*, for *L* also contains mixed sentences – i.e., those in which at least one *VT* and one *VO* term occur. In addition, it is assumed that each of the sublanguages above has its own stock of predicates and/or functional variables, and that *LO* and *LO'* have the same stock which is distinct from that of *LT*.

4. *LO* and its associated calculi are given a semantic interpretation which meets the following conditions:

   - The domain of interpretation consists of concrete observation events; things or thing-moments; the relations and properties of the interpretation must be directly observable.

   - Every value of any variable in *LO* must be designated by an expression in *LO*.

   It follows that any such interpretation of *LO* and *KO*, when augmented by appropriate additional rules of truth, will become an interpretation of *LO'* and *KO'*. We may construe interpretations of *LO* and *KO* as being partial semantic interpretations of *L* and *K*, and we require that *L* and *K* be given no observational semantic interpretation other than that provided by such partial semantic interpretation.

5. A partial interpretation of the theoretical terms and of the sentences of *L* containing them is provided by the following two kinds of postulates: the theoretical postulates *T* (i.e., the axioms of the theory) in which only terms of *VT* occur, and the correspondence rules or postulates *C* which are mixed sentences. The correspondence rules *C* must satisfy the following conditions:

   a) The set of rules *C* must be finite.

   b) The set of rules *C* must be logically compatible with *T*.

   c) *C* contains no extralogical term that does not belong to *VO* or *VT*.

   d) Each rule in *C* must contain at least one *VO* term essentially or nonvacuosly.

Let *T* be the conjunction of the theoretical postulates and *C* be the conjunction of the correspondence rules. Then, the scientific theory based on *L*, *T*, and *C* consists of the conjunction of *T* and *C* and is designated *TC*.

2.2 THE STATEMENT VIEW AND SOFTWARE ENGINEERING

Briefly, the global scientific language *L* is divided into two partial languages: the observational language *LO* with vocabulary *VO* and the theoretical language *LT* with vocabulary *VT*. The terminal symbols of *VT* only receive an indirect and partial empirical interpretation via the correspondence rules *C*. The *pure* theory *T* is formulated completely within the language *LT*, whereas the interpreted theory, which consists of the conjunction $T \wedge C$, contains expressions originating in the two languages.

Even though a more detailed discussion of the Two Level Theory of the Language of Science would be necessary to understand the fine points and minutiae of the conceptual architecture which occupies us, this is beyond the scope of the present paper.

What we will actually use here is a variation of this theory in two senses.

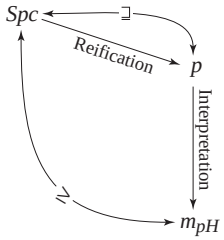
The first is that we do not consider simply an observational language and a purely theoretical one. The characteristic of the observational language being 'understandable in its own right' motivates us to analyse the concept of the *empirical basis*. We use *empirical basis* to denote the set of observable objects whose denotational terms form the vocabulary *VO* of the observational language and are, therefore, intersubjectively comprehensible. There exist two different empirical bases relevant to our present discussion. The first, called the epistemological empirical basis, has denotational terms corresponding to the vocabulary *VO* of the Statement View. The second is the so-called methodological empirical basis. This is formed by the denotational terms of the (joint) vocabularies $VO^{T'}$ and $VT^{T'}$ describing the observation instruments being employed.

If we consider some theory based on optics and for this we use a microscope, we do not have the right to say that we observe the actual object which is on the microscope platform immediately under the objective. If we would do this, we would be considering as observables the images formed in the eyepiece of the microscope basing our considerations on an explanation given by the theoretical terms of the theory we are using. If, on the contrary, we are discussing, for instance, a biological theory, we can affirm the following, without risk of using theoretical terms inadequately. "We are observing a cell which is in a tissue deposited on the platform of the microscope and not simply a light spot generated by the reflection of the light by the object on the platform through the optical system of the microscope." This concept of the methodological empirical basis is the one which generates the concept of T-theoretical terms, introduced by Joseph Sneed. (A complete exposition and, for the moment, a definitive one can be found in the structuralist manifesto "An Architectonic for Science" ([Balzer]).) For Sneed, there does not exist an absolute notion of theoretical term, but, instead, a relative concept of 'theoretical term with respect to the theory *T*'. Thus, 'cell' is an observable term with respect to the biological theory in question. However, it is theoretical with respect to the theory of thick lenses optics and the theory of microscope systems, extended by the biological theory in question.

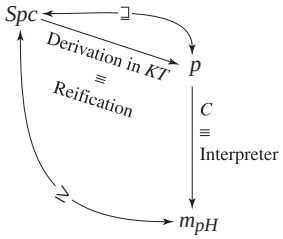
In the Figure 1 below, we will see that the abstract machine m_{pH} realising our program *p* is an observable term. However, it is theoretical with respect to the acceptance of the well behavedness of the underlying computer, operating system, etc.

The second sense in which we depart from the Statement View, already referred to above, is that we admit theories which are not axiomatised in first order logic. They may be presented via some other mathematical medium, as for example, '*x* is an *S*', where *S* is a set theoretic structure. This modification is also due to Sneed.

Let us now return to the criticism of the Statement View called 'the Putnam challenge'. The challenge posed by Hilary Putnam was that the adepts of the Statement View must provide a positive justification for the existence of theoretical terms ([Putnam, Stegmüller]). She claimed that all the existing justifications were negative, that is to say, a term is theoretical because it is not observational. This criticism, together with the methods for eliminating theoretical terms due to Hempel (using Craig's Theorem) ([Hempel]) and to Ramsey (the so-called Ramsey sentence) ([Ramsey, Stegmüller]), was never treated satisfactorily by the adherents of the Statement View. However, these problems are not manifested in the particular domain in which we want to apply this formulation of the scientific method.



(a)



(b)

Figure 1

Consider Figure 1(a). We have here a typical diagram depicting the construction of a program *p* by reification from a specification *Spc*. The interpreter I_{Lp} of the programming language *Lp*, in which the program *p* is written, creates, on the given hardware/software platform *H*, a realisation of the program *p* as a virtual machine m_{pH}.

The correctness-like relations in Figure 1(a) can be defined as:

$$m_{pH} \le Spc \leftrightarrow (\forall \delta)\Big(F\delta m_{pH}t_0 \rightarrow (\exists t)(\exists \rho)\Big(t_0 \le t \wedge H\delta m_{pH}t \wedge \delta\ m_{pH}\rho \wedge \langle \delta, \rho \rangle \in \mu[Spc]\Big)\Big) \quad (4)$$

and

$$p \sqsubseteq Spc \leftrightarrow (\forall \delta)\Big(Term(\delta, p) \rightarrow (\exists \rho)\Big(\langle \delta, \rho \rangle \in \mu[p] \wedge \langle \delta, \rho \rangle \in \mu[Spc]\Big)\Big) \quad (5)$$

where:

$F\delta m_{pH}t_0$ is a predicate meaning "the datum δ is fed into the machine m_{pH} at the time instant t_0";

$\mu[Spc]$ is the relational denotation of *Spc*;

$H\delta m_{pH}t$ is a predicate meaning "machine m_{pH} halts at the instant of time *t* after being fed with datum δ";

$\delta\ m_{pH}\ \rho$ means that the pair $\langle \delta, \rho \rangle$ is an input/output pair of machine m_{pH};

$Term(\delta, p)$ is a predicate meaning "program *p* terminates with input δ";

$\mu[p]$ is the relational denotation of program *p*

Notice that, in (4), the expression $(\exists t)\Big(t_0 \le t \wedge H\delta m_{pH}t\Big)$ is observationally undecidable, as it expresses the halting problem.

The correspondence rules C_{Int} realized by the interpreter (see Figure 1(b)) connect termination, in the theoretical language, with halting, in the observational one. So, we can write:

$$T_{Prog}, C_{Int} \vdash (\forall \delta)\Big(Term(\delta, p) \rightarrow (\exists t)\Big(H\delta m_{pH}t\Big)\Big) \quad (6)$$

We can do even more; we can Skolemize (6) and obtain:

$$T_{Prog}, C_{Int} \vdash (\forall \delta)\Big(Term(\delta, p) \rightarrow H\delta m_{pH}\xi(\delta)\Big) \quad (6)$$

Here $\xi(\delta)$ is the particular time instant in which m_{pH} will halt after being fed with the datum δ, on the assumption that we have proved the validity of $Term(\delta, p)$ in the theoretical level. So, we can write further:

$$T_{Prog}, C_{Int}, (\forall \delta)\Big(Term(\delta, p)\Big) \vdash (\forall \delta)\Big(H\delta m_{pH}\xi(\delta)\Big) \quad (7)$$

So, we can now write (4) as:

$$T_{Prog}, C_{Int} \vdash m_{pH} \leq Spc \leftrightarrow (\forall \delta)\Big(\big(F\delta m_{pH} t_0 \wedge Term\big) \rightarrow (\exists \rho)\big(\delta\ m_{pH}\ \rho \wedge \langle \delta, \rho \rangle \in \mu[Spc]\big)\Big) \quad (8)$$

This is observationally decidable because the disposition "machine halting" was introduced in the theoretical language as "program termination".

After the Putnam challenge, if we have no positive reasons for maintaining the theoretical-observational dichotomy, we should accept that this is artificial because the introduction of ways to eliminate theoretical terms. For instance, given an expression of the interpreted theory *TC* introducing a disposition, the method based on Ramsey sentences consists in replacing every theoretical term with a variable and introducing an existential quantifier binding this variable. That is, one can un-Skolemize the given expression, obtaining in this way a new expression which only contains observable terms.

Notice that if we apply the Ramsey method to (8), we will obtain again an expression like (4), which is observationally undecidable. The reason that we cannot apply the Ramsey sentence without loosing decidability is that the Putnam challenge is not applicable in our particular domain, or, even better, we can answer the challenge.

In our domain, we have a given dichotomy, which is: given that the computability of a function is a property of itself, as an object, we cannot decide if a function is computable by only observing it, for instance, by observing its graph. For affirming that a function is computable, we need to find a recursive expression denoting it. Thus, the very fact of the existence of a machine m_{fH} computing the function f is inherently related to the existence of a program p such that $m_{fH} = \mu[p]$. Hence, in the realm of Computer Science, the theoretical language in which programs, termination, complexity, etc. are expressed, and the observational one, in which machines and computation duration are expressed, are inherently given and inherently different[7].

The assertion that formal specification and formal concepts of refinement are necessary parts of software development here receives full, formal, and complete justification.

The method for deducing an observational consequence *E* from a set of interpreted theories TC_1, …, TC_m and a set of statements of antecedent conditions about particular facts can be explained by means of the so-called Hempel-Popper deductive-nomological (D-N) systematization model of explanation ([Hempel, Popper]).

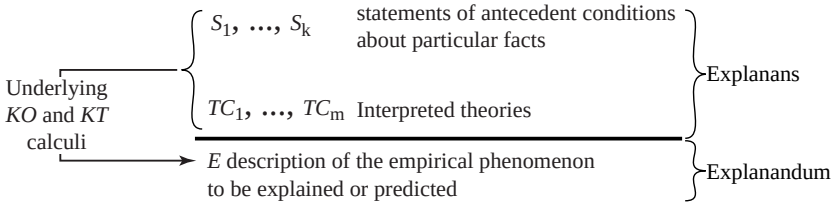


Figure 2

According to Hempel, an explanation of the event *E* (whose description is known as the *explanandum)* consists of a suitable *argument*, wherein the explanandum "correctly follows" from the premises (known as the *explanans*) of the argument. The explanandum must be a logical consequence of the explanans, i.e., the latter should be derived from the former using the appropriate calculi (i.e., *KO'* and *KT*). Thus, D-N explanations take the form shown in Figure 2, where, as was said above, TC_i are interpreted theories and S_i are statements of antecedent factual conditions. The explanans must contain essentially at least one interpreted theory (a universal law-like generalisation). Statements S_i should either be true or else 'highly confirmed'. The explanandum *E* may be either a description of an event or a law or a theory.

[7] For a complete discussion of these issues in relation to the philosophy of science see [Haeberer and Veloso 91a, 91b, 90a, 90b, 90c, 89, Veloso and Haeberer 89].

The difference between prediction and explanation is of pragmatic character. If E is given, i.e., if we know that the event described by E has occurred, and a suitable set of statements and interpreted theories $S_1,\cdots,S_k,TC_1,\cdots,TC_m$ is provided afterwards, we speak of an explanation of the event in question. It may be said, therefore, that an explanation of a particular event is not fully adequate unless its explanans, if having been taken account of in time, could have served as a basis for predicting the event in question. Consequently, whatever will be said in this article concerning the logical characteristics of explanation or prediction will be applicable to either, even if only one should be mentioned.

So, the above description of the hypothetico-deductive method in complex version (expression (3)) can be written in the form of a D-N systematisation asdepicted in Figure 3.

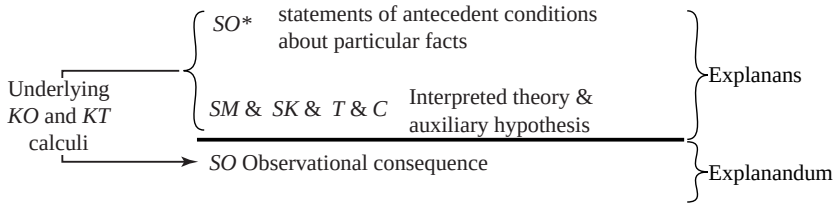


Figure 3

If we think a little about the way we validate specifications and programs in any software development process, we should accept that we have two correct ways of doing it. Suppose we already know an event SO of the real world (system, problem, etc.), in the presence of a theoretical sentence *SM*, derived from specification *Spc* and hopefully inducing *SO*, and other particular theoretical hypotheses *SK*, valid for this particular circumstance. We should try to derive *SO* from *SM*, *SK*, *Spc* and the correspondence rules *C* – giving real world meaning to *Spc* – and a set of behavioural (requirements) properties *SO** holding[8] in this particular circumstance[9].

Why did we say "a correct way of doing…"? This was because when carrying out specification, validation and program testing we are confronted with the hypothetico-deductive method in complex version. If the result of the observation does not occur as predicted, we must banish either the interpreted theory, the auxiliary hypothesis *SK*, or the a priori factual requirement properties, i.e., the scenario, we assume. If we decide to banish the interpreted theory, we have two alternatives, i.e., rejecting the specification *Spc* itself, or its denotation in the real world, i.e., the correspondence rules that interpret it. We can also have doubts about the formal derivation of the description of *SO*, but this is not a matter of opinion, it is simply a matter of reviewing the formal calculations which drive us from $S_1,\cdots,S_k,Spc$ to *SO*.

Hence, we should accept that the D-N systematization fits very well as a meta-explanation of validation and testing.

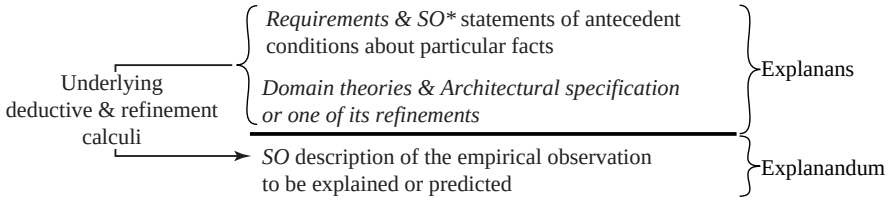


Figure 4

Figure 4 shows a simplified version of the D-N systematization applied to the domain of software engineering and the activities of validation and testing ([Haeberer and Veloso 89]).

[8] Here, since the statements S_i are factual, the meaning of *holding* can be that of a logical truth, or, more often, an intersubjective truth, or even a not yet refuted (highly confirmed?) hypothesis.
[9] Compare this description with that of a scenario in object oriented design.

3 THE ARTS ENVIRONMENT AND ITS PARADIGM

There is a growing interest in providing methods and tools to support the development of real-time systems. There are now a plethora of tools available and we must ask ourselves the question: Why make yet another attempt? The above description of the Statement View provides us with a safe and satisfactory basis for our rationale: safe because we are reusing a conceptual framework that has been demonstrated to be applicable to problems such as ours, and satisfactory because we can rationalise and explain what it is we are trying to do and why. Below we describe the conceptual architecture and some implementation details of the ARTS (formal Approach to Real-Time Systems[10]) environment, trying to explain its characteristics and intended use in terms of the Statement View. First we rehearse some of the discussion about the nature of engineering from [Maibaum 97].

According to [Rogers], "engineering refers to the practice of organising the design and construction of any artifice which transforms the physical world around us to meet some recognised need". Hence, the very nature of engineering is informed by the observational vs theoretical dichotomy: the physical (observable) world vs the language/medium of design (and the expression of 'recognised need'). "[…] The essence of technological investigations is that they are directed towards serving the process of designing and [...] constructing particular things whose purpose has been clearly defined."

"We have seen that in one sense science progresses by virtue of discovering circumstances in which a hitherto acceptable hypothesis is falsified, and that scientists actively pursue this situation. Because of the catastrophic consequences of engineering failures - whether it be human catastrophy for the customer or economic catastrophy for the firm - engineers and technologists must try to avoid falsification of their theories. Their aim is to undertake sufficient research on a laboratory scale to extend the theories so that they cover the foreseeable changes in the variables called for by a new conception. The scientist seeks revolutionary change - for which he may receive a Nobel Prize. The engineer too seeks revolutionary conceptions by which he can make his name, but he knows his ideas will not be taken up unless they can be realised using a level of technology not far removed from the existing level." Again, [Rogers] identifies the role of validation and testing in relation to the design of required artifacts and distinguishes this role from the corresponding (related) role in science.

According to [Vincenti] (and in accordance with the latter part of the above statement by [Rogers]), the day to day activities of engineers consist of *normal design*, as comprising "the improvement of the accepted tradition or its application under 'new or more stringent conditions'". He goes on to say: "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has good likelihood of accomplishing the desired task."

[Jackson] discusses this concept of 'normal design', although he does not use this phrase himself. "An engineering handbook is not a compendium of fundamental principles; but it does contain a corpus of rules and procedures by which it has been found that these principles can be most easily and effectively applied to the particular design tasks established in the field. The outline design is already given, determined by the established needs and products." … "The methods of value are micro-methods, closely tailored to the tasks of developing particular well-understood parts of particular well-understood products."

An implied but not explicitly stated view of engineering design is that engineers normally design devices as opposed to systems. A *device*, in this sense, is an entity whose design principles are well defined, well structured and subject to normal design principles. A *system*, in this sense, is an entity that lacks some important characteristics making normal design possible. "Systems are assemblies of devices brought together for a collective purpose." Examples of the former given by [Vincenti] are airplanes, electric generators, turret lathes; examples of the latter are airlines, electric-power systems and automobile factories. The software engineering equivalent of devices may include compilers, relational databases, PABXs, etc. Software engineering examples of systems may include air traffic control systems, internet banking systems, ... It would appear

[10] Acronym construction is dangerous to your health!

that systems become devices when their design attains the status of being normal, i.e., the level of creativity required in their design becomes one of *systematic choice*, based on *well defined analysis*, in the context of *standard definitions* and *criteria* developed and agreed by the relevant engineers. The implications of this conception of design are clear: the design methods of engineers are specific and heavily sytematised. The implication is that the design environment and supported methods are highly specific and heavily domain dependent. (An obvious conclusion may be that environments designed to be very general in their application are likely to be less than effective in any particular domain, not being able to deal directly with the concepts, notations and methods of that domain.)

We now proceed to describe the ARTS environment paradigm and its present instantiation. We relate, briefly, the discussion to the above description of the Statement View. We will use Figure 5 (and the part depicted in Figure 6) as points of reference during the discussion. The numbering of the paragraphs below corresponds to the numbering in the Figures identifying component aspects of the environment.

An environment for software development should support at least the following activities and notations:

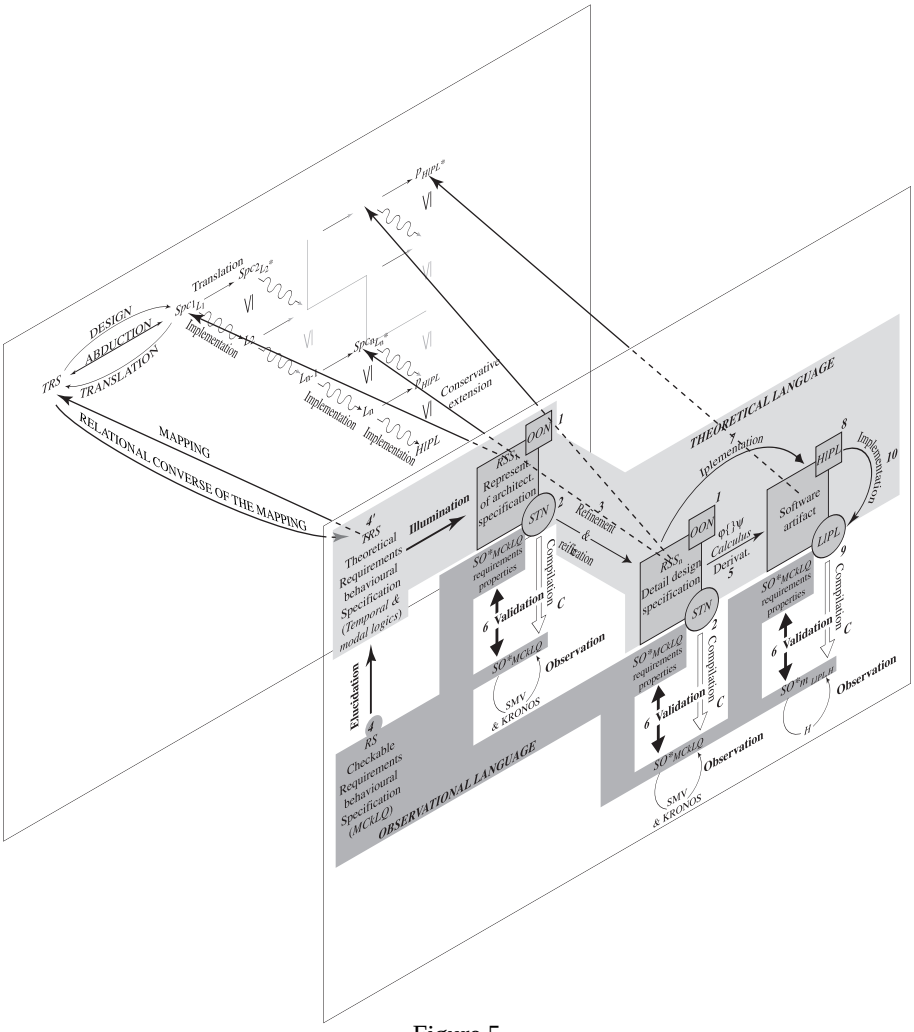


Figure 5
(Notice that each mapping should have its relational converse; they are not shown for the sake of clarity in the figure.)

1. The engineer performing normal design requires a standardised notation of proven utility as a medium in which to develop the design of the required artifact. This notation forms part of the theoretical language of the engineer. Since normal design is a 'standard', from which an engineer is not easily allowed to depart (having to suffer the penalty of not being able to explain easily

their design and have it accepted), it should be supported by 'standardised and accepted design notations'. In the present state of development of software engineering, Object Oriented languages would seem to provide (at least a starting point for) such a notation. Hence, in ARTS we adopt one of the standard OO notations (historically that of Syntropy ([Cook and Daniels]), but now migrating to UML ([OMG])). Because we are interested in supporting a well founded method, we choose restrictions and adaptations to such an OO notation which relate to both the domain of application (where, for example, real-time constraints and their treatment are very important) and to the intended semantics of the engineering notation (and the restrictions/alterations to the interpretation of the notation by the engineer that this imposes). The resulting language (which is often referred to as the 'high level design notation', is the theoretical language which is referred to by [Vincenti] as the engineering notation. It is important to note that it is not fixed across all instantiations of ARTS, but is designed, de novo, to be suitable for the specific instantiation. (For example, in another project, Mensurae, addressing process definition, improvement and metrication, a different set of languages and tools are assembled for supporting the work of the process engineers.)

2. One of the adaptations to the engineering notation that is required by the domain of application (with ramifications for the intended semantics of the notation) is the need to deal with so called hard real-time constraints in the implementation domain of PABXs. In order to support the activities of validation and verification, we need a notation that enables the engineer to express the required design and to check that this design satisfies some required or predicted properties. We have identified *timed automata* ([Alur and Dill, Alur and Henzinger 94, 96]) and *hybrid automata* ([Henzinger, Daws et. al.]) as satisfactory notations for representing the timed transitions of system components. These automata are not the class allowed by the standard OO notations and so there is a need to adapt the notations. Furthermore, the method related to this integrated notation must be explicated so as to provide an effective design tool for the engineers.

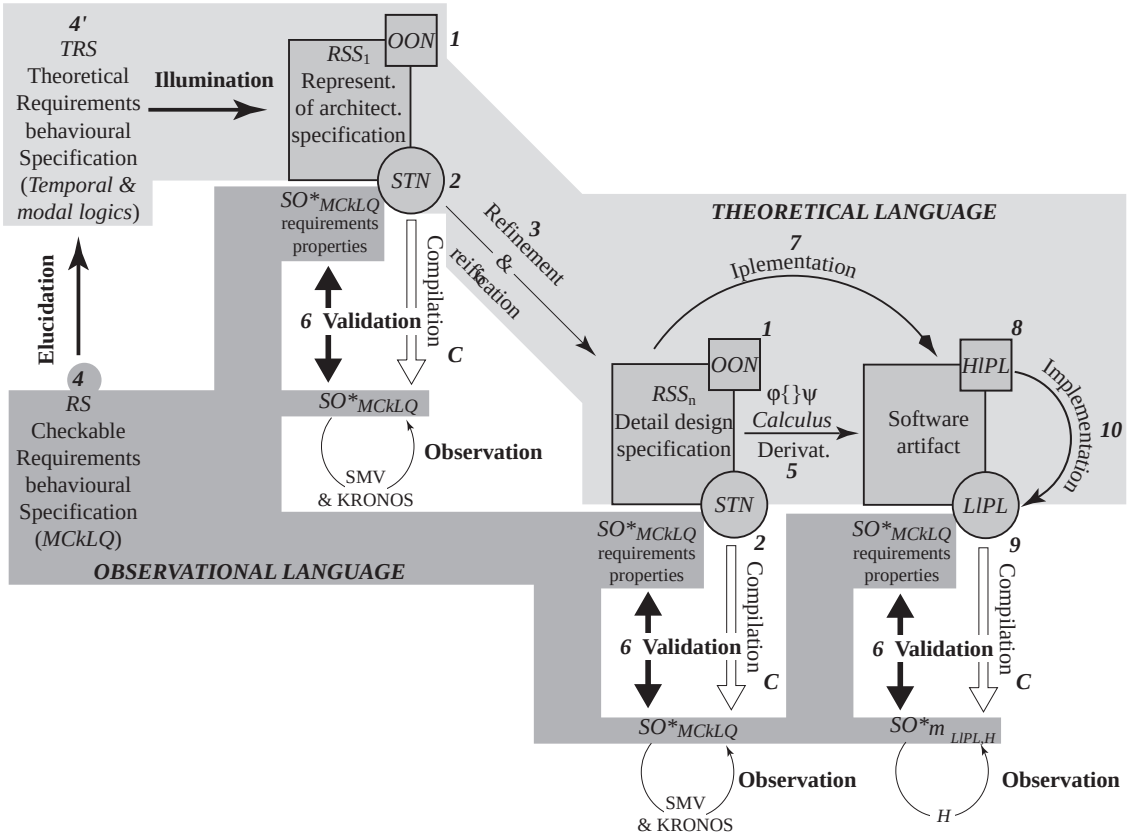


Figure 6

The automata have a representation in the language (a temporal logic) of a model checker. (Several model checkers and their languages will do. Presently we use *SMV* and *KRONOS* ([Clarke et.al., Daws et. al.]).) Such a model checker is essential as a tool of analysis to help the engineer determine if the present design satisfies required or predicted properties. Of course, the engineer is not expected to formulate queries to the model checker directly, as this would require knowledge of the underlying temporal logic beyond the capabilities of most engineers using the environment. Instead, a palette of useful patterns is available. As time goes on, we will develop a larger palette and augment this simple query mechanism with useful automated analysis tools.

3. Software development environments must support the activity of *refinement* or *reification.* This activity involves verification (of a more detailed design against the preceding more abstract one). This process is one that takes place purely at the theoretical level of language (but see the discussion of validation in 6 below.) There are at least two languages involved in this activity. Firstly, there is the engineering language with OO notations as the medium of expression. Behind this plane of language, there is the formal language and calculus in terms of which the engineering notation is attributed precise semantics[11]. Why have these two languages? The answer is simply one of utility. The engineering notation is 'native' to the engineer and is an effective tool in his/her hands. The underlying mathematical notation allows us to explain exactly what we mean by concepts such as refinement steps and their correctness. (This may not be possible or easy at the level of the engineering notation.) This then helps us justify what we mean at the level of engineering notation by phrases such as 'applying a pattern to transform (a part of) a representation into a more detailed one'. Such a transformation is perhaps chosen from a suite of such patterns of refinement transformations that have proved useful in the design of a particular class of systems, such as PABXs. Occasionally, a sophisticated user may want to use directly the formal notation to justify a new refinement pattern or reason about some property which is important, but not effectively represented in the engineering notation. There is no problem in principle with such use of both languages as they are both intended to support reasoning at the theoretical level of discourse[12], i.e., they are both part of the theoretical level of discourse. We foresee that, in a well- specified domain, such as that of PABXs, the 'normal design' method assumes a fixed number of refinement levels to be used. Hence, there is not normally a need to support arbitrary kinds of design steps; the environment can be seen to be enforcing a particular method to the extent that it disallows departures from this norm.

The engineer will have to convince himself/herself that a pattern of refinement is applicable. There will often be formal conditions to check (being the equivalent of an engineering calculation to enable a choice in design). The engineer may be convinced simply by thought experiments or may use available tools, such as the model checkers or even theorem provers. Thus, the tools required for validation and testing, although not in principle necessary for reification, may still prove useful.

That we need a calculus of refinement (and a representation of it in the engineering notation and method) is an artifact of the domain. We are unable to describe a direct connection between problem and solution and must have recourse to the theoretical level to develop (albeit indirectly) such a solution. This recourse to the theoretical level of discourse has been commented on in many works on software development, with [Lehman et al] being a notable example. Seen in this context, it is simply a restatement of a universal truth, rather than being a new 'revelation'.

4. As noted by [Rogers], an engineer designs an artifact to fulfill a stated need. This 'need' is what is referred to in software engineering as the 'requirement'. The requirements specification, so called, may consist of two parts. The first is an observational language in which we may state expected observational properties of our intended artifact. Such properties may include logical

[11] So the engineering notation had a precise semantics after all! But we observe that the engineering notation had a life of its own before meeting up with precise mathematics. However, what we must end up with is a pair of related formal languages. More importantly, the semantics of a notation is itself a *theoretical* notion in this conceptual framework.

[12] There may be a problem, of course, of displaying in the engineering notation some refinement defined and justified in the formal notation.

generalisations, such as 'for all …' or 'at all time instants …'. However, we also want to state properties that correspond to scientific generalisations, an example being liveness properties[13].

5. At some point in the refinement process, an engineer might wish to hand over to an automatic tool the problem of synthesising an executable program to implement a required method. The state at which this happens will depend on the technology available (and on the predilections of the engineer or the organisation). In the present instantiation of ARTS, we assume that sequential programs without loops are synthesisable and we are building tools to implement this synthesis. As we gain more and more experience in a domain of design, we will systematise more and more the standard design procedures, to the extent that they may become automatable[14].

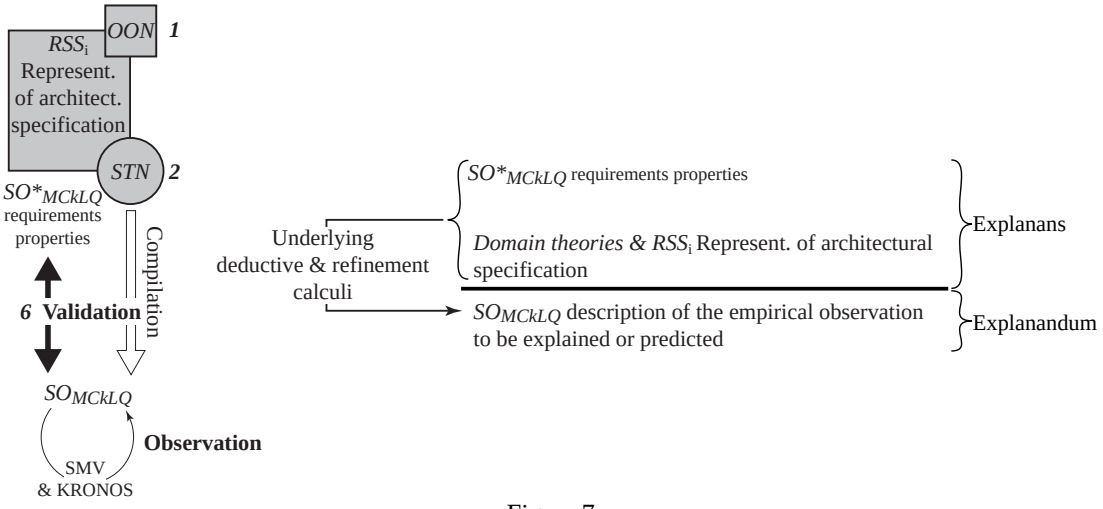


Figure 7

6. The left part of Figure 7 is a validation activity as depicted in figures 5 and 6, whilst the right part is an instantiation of the D-N systematization applied to the domain of software engineering (Figure 4) for the context of ARTS conceptual architecture. What we do in ARTS is to state a set of requirements properties *SO** that hold in the real world as given by the requirements – these properties are stated in a language directly translatable to *MCkLQ*, i.e., the language of the model checkers (in our case, *SMV* and *KRONOS*). From these properties and the "compilation" (by transformations) of *STN* into *MCkLQ*, we derive by means of an ad-hoc constraint solver (called *DEXVAL*) a description (in the form of a trace with acceptable ranges of valus for the relevant variables) $SO*_{MCkLQ}$ of the meaningful[15] empirical observation we want to explain or predict. In the latter case, the corresponding model checker, i.e., *SMV* or *KRONOS*, model checks $SO*_{MCkLQ}$ against our design, giving either an affirmative answer or a counterexample, in the form of a trace, carrying out what in scientific experimentation would be the empirical observation. Notice that, in order to consider *MCkLQ* as an observational language, we must rely on the concept of being T-theoretical. In this case, we are considering the model checkers as part of the methodological empirical basis, i.e., they are our instruments (whose correct performance we choose to accept).

[13] Liveness is not an observationally decidable property. It requires 'too many' (an infinite!) number of observations. Hence, it requires theoretical treatment, i.e., proof.
[14] This is, of course, the ultimate aim of all software development managers: no more engineers to get in the way!
[15] Anyone who has been involved with software testing, specification validation or experimental design, in general, knows very well the difficulty of deriving meaningful experiments, i.e., experiments which really have (in principle) the power of refuting the hypothesis. In Science, this "derivation" relies on the skill of the scientist. In our case, due to the fact that our specifications and properties are stated in a really formal way, we have the advantage of being able to rely on formal procedures for guaranteeing the meaningfulness of the experiments. The *DEXVAL* validation experiment deriver is a first approach we have tried to address this crucial aspect.

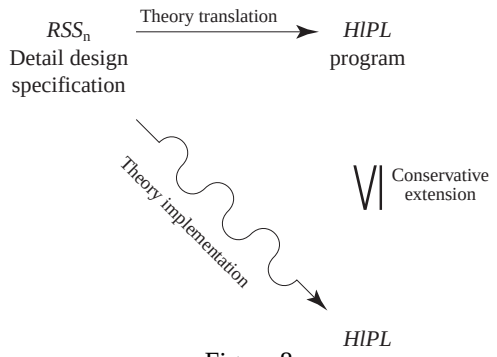


Figure 8

7. The synthesis activity described in 5 above may be seen as a part of a more general *implementation* activity which realises the detailed design called RSS_n in Figure 6 by means of the 'Software artifact'. This relationship is an instance of what is described in [Turski and Maibaum, Maibaum et. al. 85, 91] as an implementation of RSS_n in terms of *HlPL*, the high level programming language which is used to code the detailed designs of the OO method, via the 'mediating specification called *HlPL* above. See Figure 7. What is going on here is that we use our synthesis engine to provide code for individual methods in our design, but we still require a programming language representation of the structural part of our design (i.e., all parts of the detailed design not corresponding to methods). (Whether this step encompasses the synthesis tasks in 5 or is invoked after these steps are accomplished depends on the method being imposed on the engineer.) The supposition behind this step is that there is a standard way of translating OO structuring from detailed design to a programming language representation. This may be considered part of the synthesisable part of the design to be implemented in a standard way by the designers of the environment.

8. We referred above to the high level programming language, *HlDL*, which may simply be a theoretical artifact used to act as an intermediary between the various executable languages to be used in realising designs and the native languages required to actually execute the designs in real environments. This language may itself be executable, but, its role in design is important. Given this standard representation in a high level executable form, it may well be easier to generate executable code in various standard languages than directly from low-level designs in OO notation. Hence, *HlPL* serves a simply utilitarian role. *HlPL* is, perforce, part of the theoretical level of discourse. It describes a theoretical artifact, i.e., the program we have designed.

9. We wish, eventually, to generate executable code that will be used to realise the application we have in mind. In the present instantiation of ARTS, this target language, i.e., our *LlPL* is C++. As for 8, C++/*LlPL* forms part of the theoretical plane of discourse, requiring interpretation via a (physical) machine to form its observational counterpart. This interpreter forms part of the correspondence rules relating the program *p*, a theoretical artifact, with its observational counterpart (the m_{pH} of Figure 1), the program (with specific 'inputs') executing on some software/hardware platform *H*. (Note that the *HlPL* may have its observational counterpart, the abstract machine formed by the program in *HlPL* and the machine (and related software) on which we execute the program in *HlPL*. This implies that the language *HlPL* is executable! In the present instantiation of ARTS, the role of *HlPL* is played by *DDL*, the Detailed Design Language ([Carvalho]) which is <u>not</u> executable.)

10. Here we have another instance of the implementation relation referred to in 7. We have the 'abstract' description in *HlP*L (DDL) and we wish to realise this in terms of the 'concrete' level of description *LlPL*(C++). This is a standardised part of design and may be (and is) automated.

11. We have many instances of the concept of 'translation' used in the above architecture. As a matter of policy, whenever possible we use transformational methods to affect this kind of translation ([Garcia et. al.]). The reason for this is methodological. We wish to have certain properties of such translations: correctness (the result of the translation bears some defined relationship to the source), traceability (certain properties of the translation can be related structurally to the corresponding parts of the source), optimality, etc. We have found that the

transformational engine *TXL* ([Cordy et. al.]) provides in this instantiation of ARTS a highly effective engine for realising such translations. For example, over and above enabling us to address these properties effectively, the resulting C++ code is highly efficient in execution[16].

12. We are interested in such environments in using multiple languages/representations, defining various translations and relationships, formulating queries to model checkers, possibly proving properties of our artifacts, etc. We require an interface generator to facilitate the generation of such interfaces. For the present instantiation of ARTS, we have a tool called RECOPLA which serves this purpose. It is a meta-editor for synthesising diagrammatic/textual editors with facilities for associating semantics based manipulation with syntactic entities and operations. For example, the transformational translation of one notation to another is generally handled in this manner.

13. The environment requires persistency properties and a supporting architecture for system integration. In the present instance of ARTS we use ObjectStore for the former and CORBA for the latter. A discussion of this aspect is beyond the scope of this paper.

14. The environment integrates at a conceptual level various tools and languages to support software development. The tools need integration not just in the sense of 13, i.e., being able to 'talk' to each other, but also as component arts of an integrated method. This involves building and understanding user interaction models, but, although this aspect is very interesting in its own right, it is unfortunately beyond the scope of the paper.

15. There is a whole area of research and experimentation with automated and user assisted tools to support the various activities in software development. The choice of appropriate model checkers, theorem provers, type checking tools, other automated forms of analysis (possibly based on abstract interpretation principles), etc., is a long term task which is also likely to be highly domain influenced.

4 CONCLUSIONS

During the last three years, we have been building an instantiation of a system's development paradigm, called ARTS. The paradigm consists of a view of what a system development environment is, in general terms, and a methodology for instantiating the paradigm for particular and specific domains of application. The specific domain of instantiation is for the design of PABXs, with the work being supported by Siemens Telecomunicações (Brazil) who want to apply it to improve their productivity and the quality of their products.

The motivation for and the explanation of the paradigm are derived from extant epistemological models of the method of Natural Science. We assert that these models are directly applicable to the domain of software and systems construction, and that, from them, we can derive principles and explanations for what a software development environment should be. Although these approaches to the method of Natural Science have not had universal acceptance amongst scientists and philosophers, the specific criticisms put forward appear not to be applicable to our domain of discourse, i.e., software engineering. The distinctions introduced by the Statement View help us software engineers disambiguate different languages and levels of discourse, providing systematic explanations for many of the problems pre-occupying us in the process of building software. By using these principles in the design of a conceptual architecture for software development environments, we hope that we have rationalised many of the choices and dilemmas facing us.

Although the discussion of the Statement View and of the specific instantiation of this architecture, called ARTS, was perforce brief, we nevertheless hope that we have conveyed some of the essential ideas and the 'spirit' of the exercise. We hope to report further in the future on other instantiations of the conceptual architecture (such as that for Mensurae, mentioned above, also funded by Siemens Telecomunicações (Brazil)) and on successful experience of using the instantiations of the paradigm[17].

[16] This is, of course, an observational observation.
[17] Of course, this is a theoretical statement whose observational consequences will have to be confirmed in the future!

5 REFERENCES

[Alur and Dill] Alur, R. and Dill, D.L. (1994) A Theory of Timed Automata, *TCS* 126, 183-235.

[Alur and Henzinger 94] Alur, R. and Henzinger, T.A. (1994) A Really Temporal Logic, *JACM* 41(1) 181-204.

[Alur and Henzinger 96] Alur, R. and Henzinger, T.A. (1996) Reactive Modules, *Proc. Of 11th LICS*, IEEE Computer Society Press, 207-218.

[Balzer et. Al.] Balzer, W., Moulines, C. U. and Sneed, J. D. (1987) An Architectonic for Science. The Structuralist Program, Synthese Library, Volume 186, D.Reidel Publishing Company.

[Broy and Pepper] Broy, M. and Pepper, P. (1981) Program Development as a Formal Activity, *IEEE Trans. Software Engin.*, SE-7 (1) 14-22.

[Carnap50] Carnap, R. (1950) Empiricism, Semantics, and Ontology, Revue internationale de Philosophie, 11, 208-228; reprinted in Linsky (1952) and the enlarged 1956 edition of Carnap (1947).

[Carnap56] Carnap, R. (1956) The Methodological Character of Theoretical Concepts, pp. 33-76 in Feigl and Scriven.

[Carvalho] Carvalho, S., (1995) The Design Description Language, Project ARTS Technical Report 2, Formal Methods Laboratory of the Departamento de Informática of the Universidade Católica in Rio de Janeiro (LMF–DI),.

[Clarke et. al.] Clarke, E. M., Grumberg, O., and Long, D. E. (1992) Model checking and abstraction. In *Proc. Of POPL '92*.

[Cook and Daniels] Cook, S. and Daniels J. (1994), *Designing Object Systems*, Prentice Hall.

[Cordy et al] .Cordy, J.R, Halpern-Hamu, C.D. and Promislow, E., (1991) TXL: A rapid prototyping system fir programming language dialects, *Computer Languages*, 16(1).

[Daws et al] Daws, C., Olivero, A., Tripakis, S. and Yovine, S. (1996) The Tool KRONOS, in Alur, R., Henzinger, T.A. and Sontag, E.D. (eds) *Hybrid Systems III*, LNCS 1066, Springer-Verlag, 208-219.

[Enderton] Enderton, H.B. (1972) *A Mathematical Introduction to Logic*. Academic Press; New York.

[Feigl and Scriven] Feigl, H. and Scriven, M., eds. (1956) Minnesota Studies in the Philosophy of Science, Vol. II. Minneapolis: University of Minnesota Press.

[Garcia, et. al.] Garcia, A.V., Haeusler, E.H., Haeberer, A.M. (1997) A Semantic Approach to the Solution for the Legacy Code Problem. Formal Methods Pacific. Wellington, Nova Zelandia. 1997. Proceedings. 57-69. Victorian University of Wellington..

[Haeberer and Veloso89] Haeberer, A.M., Veloso, P.A.S. (1989) "The Inevitability of program testing - A theoretical analysis". Proceedings of the XVII Chilean Conference on Informatics. July 1989.

[Haeberer and Veloso90a] Haeberer, A.M., Veloso, P.A.S (1990) "Towards a Formal Analysis of the Software Development Process", Document 641-BUR-6, 41st Meeting of the IFIP Working Group 2.1 "Algorithmic Languages and Calculi"; Chester, England.

[Haeberer and Veloso90b] Haeberer, A.M., Veloso P.A.S. (1990)"Why Software Development is Inherently Non-monotonic A Formal Justification" - Proceedings of the 10th European Meeting on Cybernetics and System Research.

[Haeberer and Veloso90c] Haeberer, A.M., Veloso P.A.S. (1990) "Why Software Development is Inherently Non-monotonic - A Formal Justification"- 10th European Meeting on Cybernetics and System Research.

[Haeberer and Veloso91a] Haeberer, A.M., Veloso, P.A.S. (1991) "Some Epistemological Reflections on Software Development". Journal of Symbolic Logic, June 1991.

[Haeberer and Veloso91b] Haeberer, A.M., Veloso, P.A.S. (1991) "Epistemological Issues in Software Development" - Proceedings of the 9th International Congress on Logic, Methodology and Philosophy of Science, Uppsala, Sweden.

[Hempel] Hempel, C. G. (1965) Aspects of Scientific Explanation And Other Essays in the Philosophy of Science, The Free Press, New York

[Henzinger] Henzinger, T.A. (1996) The Theory of Hybrid Automata, *Proc. Of 11th LICS*, IEEE Computer Society Press, 278-292.

[Hesse] Hesse, M. (1966) Models and Analogies in Science, Notre Dame, Ind.: University of Notre Dame Press.

[Jackson] Jackson, M. (1997) Private Communication.

[Lehman et al] Lehman, M.M., Stenning, V. and Turski W.M. (1984) Another Look at Software Design Methodology *ACM Software Engineering Notes* 9(2) 38-53.

[Maibaum and Turski] Maibaum, T.S.E. and Turski, W.M. (1987) On What exactly Goes on When Programs are Developed Srtep-by-step, *Proc. 7th ICSE*, IEEE Computer Society Press, 528-533.

[Maibaum et al 85] Maibaum, T.S.E, Veloso, P.A.S. and Sadler, M.R. (1985) A Theory of Abstract Data Types for Program Development: Bridging the Gap?. In Ehrig, H., Floyd, C., Nivat, M. and Thatcher, J. eds. *Formal Methods and Software Development; vol. 2: Colloquium on Software Engineering*. Springer-Verlag, Berlin, 214-230.

[Maibaum et al 91] Maibaum, T.S.E, Veloso, P.A.S. and Sadler, M.R. (1991) A logical approach to specification and implementation of abstract data types. Imperial College of Science, Technology and Medicine, Dept. of Computing Res. Rept. DoC 91/47, London.

[Maibaum 86] Maibaum, T.S.E. (1986) The role of Abstraction in Program Development. In Kugler, H.-J. (ed) *Information Processing '86*. North-Holland, Amsterdam, 135-142.

[Maibaum 97] Maibaum, T.S.E. (1997) Software Engineers do not Take Engineering Seriously: an Academic('s') View of Software engineering Education. Proc. of ESEC/FSE, LNCS, Springer-Verlag.

[Nagel, et.al.] Nagel, E., Suppes, P and Tarski A., eds. (1962) Logic, Methodology, and Philosophy of Science: Proceedings of the 1960 International Congress. Stanford, Calif.: Stanford University Press.

[OMG] UML Proposal to the Object Management Group, Version 1.1, September 1997.

[Popper] Popper, K. (1959) The Logic of Scientific Discovery. London: Hutchinson.

[Putnam] Putnam, H. (1962) What Theories Are Not, pp. 240-251 in Nagel, et. al.

[Ramsey] Ramsey, F. P. (1931) The Foundations of Mathematics and Other Logical Essays. London: Kegan Paul; New York: Harcourt Brace.

[Rogers] Rogers, G.F.C., (1983) *The Nature of Engineering*, The Macmillan Press Ltd.

[Shoenfield] Shoenfield, J. R. (1967) *Mathematical Logic*. Addison-Wesley, Reading.

[Stegmüller] Stegmüller, A. (1979) Probleme und Resultate der Wissenschafstheorie und Analytischen Philosophie Band II: Theorie und Erfahrung, Springer-Verlag, Heidelberg.

[Suppe] Suppe, F. (1979) The Structure of Scientific Theories, University of Illinois Press.

[Turski and Maibaum] Turski, W.M. and Maibaum, T.S.E. (1987) *The Specification of Computer Programs*. Addison-Wesley, Wokingham.

[Veloso and Haeberer] Veloso, P.A.S., Haeberer, A.M. (1989) "The Inverted U Meta-model of the Software Development Process"- Proceedings of the HICSS-22. 22th Hawaii International Conference on System Sciences. January 1989.

[Vincenti] Vincenti, W.G. (1990)*What Engineers Know and How They Know It*, The Johns Hopkins University Press.

[Zave and Jackson] Zave,P. and Jackson, M. (1997) Four Dark Corners of Requirements Engineering, *ACM TOSEM* 6(1) 1-30.