Software Engineering Education: The Rôle of Formal Specification and Design Calculi

Dines Bjørner* Jorge R. Cuéllar†

October 1997, Revised May 1998

Tony Hoare on 'Formal Methods':
Maturity: *Use of a formal method is no longer an adventure; it is becoming routine.*
Convergence: *The choice of a formal method or tool is no longer controversial: they are chosen in relation to their purpose and they are increasingly used in effective combination.*
Cumulative progress: *Promise of yet further benefit is obtained by accumulation of tools, libraries, theories, and case studies based on the work of scientists from many schools which were earlier considered as competitors.*[1]

Abstract

This paper analyses current principles of software development: from domains via requirements to design. On the basis of this analysis we outline a structure and contents of professional software engineering. From this we extract some requirements to a university graduate (MSc.) curriculum in software engineering. We summarise the four software engineering axes that we wish to emphasize in this paper:

- software engineering as a responsible profession,
- abstraction, linguistics and logic,
- methodology, formal specification and design calculi
- domain, requirements and software-design engineering.

We view (i) engineering as 'walking the bridge between science and technology' — with engineers using mathematics as and when appropriate, (ii) methods as 'sets of principles for analysing problems and for selecting and applying techniques and tools in order efficiently to construct an efficient artifact (here software)'; and (iii) software engineering as consisting of 'domain engineering, requirements engineering and software design (engineering)' — with software development comprising all these stages and teams of engineers specially educated in sub-branches of software engineering.

Since software engineering produces and consumes descriptions and since professional engineers create varieties of abstractions we conclude that they make use of varieties of formal specification languages and design calculi — to represent abstract and concrete descriptions and to calculate over and between these.

The paper may be incomplete in not covering aspects of AI and knowledge based engineering. It also does not deal with the dimensioning and performance evaluation of hardware and software systems.

*Department of Information Technology, Bldgs. 343–345, Technical University of Denmark, DK–2800 Lyngby, Denmark; Fax: +45-45.88.45.30, E–Mail: db@it.dtu.dk, http://www.it.dtu.dk/~ db

†Siemens AG, Department of Research and Development, D-81730 Munich, Germany; E–Mail: Jorge.Cuellar@mchp.siemens.de

[1]Private communication, August 1997, received in connection with the ongoing preparation of FM'99: World Congress on Formal Methods (in the design of computing systems), Toulouse Congress Centre, 20–24 September, 1999 — an ACM, AMAST, EATCS, ETAPS, EU, FME, IFIP and IEEE sponsored event.

Contents

1 Introduction

1.1 Background

Some 30 years ago the term 'software engineering' was coined [173].

Today, 30 years later, there is still a great disparity in the education world-wide of software engineers, and there is still a great disparity among educators as to the contents of a proper education.

The practice of software construction is probably the only engineering profession left which is not regulated by standards such as followed in other branches of engineering.

1.2 Still an Immature Profession

We find this situation, after more than 30 years of university education and industrial practice, somewhat questionable.

We are ashamed that software products are put on the market although they are known to contain thousands of bugs! We believe it should be possible to provide some form of guarantee of proper software functioning — as we see it in other engineering fields.

With this paper we wish to outline facets of what we see as constituting professionalism, and how a university graduate education might achieve this. We do so through extensive, yet terse analysis and by advancing (i.e. proposing) a close-to-comprehensive view of software engineering.

1.3 The Thesis

The thesis of this paper is that formal techniques and tools based on formal understandings of software development should be main components in proper, professional software development.

By formal techniques we mean such that are built on formal specification and design calculi. By formal specification we basically mean a mathematics, including logic, and abstraction oriented specification which emphasizes properties rather than algorithms — even when we are abstracting properties of algorithms! — and by design calculi we mean formal, mathematical, in particular formal logical rules that apply to formal specification texts and allow us to deduce or calculate properties (as in other engineering disciplines).

The objects that can exist inside a computer are data and programs (or processes); but in order to create them, to reason about them, to understand their relationship to one another and to demonstrate that they really meet their objective we need more: models of application domains, specifications, properties of programs, abstractions, algorithms, implementation procedures, etc. Many people believe (or act as if they do) that it is not important to discuss semantical matters when talking about programs: the meaning of programs (and specifications, etc.) is "clear", a program performs a sequence of steps of checking conditions, jumping accordingly in the code, and setting variables to expressions of other variables. Many programmers become addicted to the illusion that they understand a program as soon as they know, instruction by instruction, how the instruction will be compiled. A low-level semantics based on the sequence of instructions seems sufficient. But as soon as the code is large or "tricky", or contains side effects or race conditions, this "understanding" of the program clearly becomes insufficient. Even within structured programs, it is unavoidable to understand large pieces of code as performing a certain clearly defined functionality. Function calls, for instance, should not be understood as "a

sequence of programming-language instructions" but, as the name indicates, as computing a function (say, $f(n) = n^2$) which is naturally represented in a syntax different from the program itself and at a different *abstraction level*. To reason about the relationship of a program to the function that it calculates, one must necessarily be talking about semantics at several abstraction levels.

One distinguishing feature of programs written in a given programming language, is that a program is on the one hand a syntactical object (a sequence of strings in a certain alphabet) but, on the other, its "meaning" is a semantical (mathematical) object, and it is this meaning that really concerns us. For instance, a program may be replaced by a semantically equivalent program in any context. This is also the case with mathematical formulas written in a given logic.

It is relatively easy to write discrete chaotic dynamical systems as short programs using simple arithmetic on real numbers. And it is of course very difficult to predict what this program is going to do in the next one or two seconds. If we do not want that our compilers, operating systems, transaction monitors, etc. act like chaotic systems (and *too many*, if not most, large systems in unusual conditions are quite unpredictable and do not satisfy the expected behaviour) we then need two things:

- structural rules on instructions/modules/programs that restrict the way they are composed to larger systems (preferably of syntactical nature, or at least, "easy" to check) and
- to take formal methods (specifications, semantics, etc.) seriously.

The first is what is called "structured programming": caution about go-to statements, type checking or programming conventions. We believe that this topic is still not completely mature, particularly for concurrent programming; still more research and experience is needed. The second point implies quite a drastic change in the way we teach software engineering today.

Since our presentation hinges very much on the distinctions between domain engineering, requirements engineering and software-design engineering, all of which we argue should be pursued also using formal techniques, we very briefly define these three concepts. In domain engineering we establish models of the application domain ideally without any reference to requirements of contemplated software and without any reference to the software design. In requirements engineering we establish models of the external properties that are expected from the software and ideally without any reference to how that software might achieve those properties. Finally by design engineering we establish models of the externally observable software interfaces (the software architecture), of the internally observable interfaces (the program organization), and stepwise refine the program organization into executable code. We summarise the four software-engineering axes that we wish to emphasize:

- software engineering as a responsible profession,
- abstraction, linguistics and logic,
- methodology, formal specification and design calculi, and
- domain, requirements and software-design engineering.

Although we sweep our analysis broadly we fail to analyse a proper rôle for AI and knowledge based engineering. That is a great pity and shame, and renders this paper a torso. We apologise.

1.4 Curriculum Experience

Since approximately 1984 the department of the first co-author has been moving in the direction of software-engineering education here being proposed. In the first five years (mid 1992 — mid 1997) of the existence of UNU/IIST, the United Nations' University's International Institute for Software Technology, under the UN Director–ship also of the first co-author, vigorously pursued the formal specification and design calculi as well as the domain engineering, requirements engineering and software design engineering paradigm here being brought forward. From the successes of Danish software houses (CRI (Computer Resources Intl.), DDC Intl., PDC (Prolog Development Center), etc.), and from the successful technology transfer of these paradigms also to many groups in some one and a half dozen developing countries on four continents (Asia, Africa, South America and Eastern Europe), we take the liberty of concluding that the ideas put forward here have been thoroughly tried & tested.

The second co-author has applied successfully the formal software techniques paradigm, especially in connection with real-time, safety-critical systems at Siemens. He has taught mathematics, computer science and industrial engineering courses in several universities, including Ohio-State, U. de los Andes (Bogota), Dortmund, Chemnitz and Munich.

2 Software Engineering as a Responsible Profession

Before delving into the specific message of this paper, to be found from section 3 on, we analyse the general position of our field: from science to technology via engineering.

We believe that it is important that the software engineer has a clear view of the professional standing of software engineering: vis-a-vis and hand-in-hand with science and technology.

2.1 Some Occupations

Traditional *scientists* study nature and abstracts — using mathematics — its mechanics, electricity (including electronics), hydrology, chemistry or astronomy.

Mathematicians study mathematics and some apply it to for example the above natural sciences.

We are taught some of these disciplines in school, with a view towards (i) understanding the world around us, (ii) appreciating science and mathematics as used in modelling nature, and (iii) preparing some of us for a further, academic or occupational career based on the sciences.

Engineers, be they mechanical, electronics, chemical or astrophysics engineers, build on scientific results, and deploy mathematics in modelling their technological artifacts with a view towards understanding them, being able to compute properties, and being able to predict their behaviours. Likewise with software engineers — as we shall see.

Engineers "walk the bridge between science and technology". Engineers create technology based on scientific insight. And engineers "act like" scientists when studying an otherwise insufficiently documented technology in order to understand it as a scientific and engineering artifact.

Based on scientific results and also the use of mathematics, engineers are able to construct technologies. The technologies that we focus on are those whose construction reflect an understanding of underlying theories, i.e., sciences.

Technicians manipulate technologies without necessarily deploying mathematics or understanding their background sciences.

Technology managers are typically former engineers. They typically manage engineers, projects and products, while pushing technologies to their limits, scientifically and commercially.

By *software* we understand the translated program for some application (the target code) as well as the program itself (the source code), its installation, maintenance and usage manuals, and all the documentation that was generated when the software was first developed.

By *software technology* we understand software that fits specific computing platforms.

2.2 Computer Science

By *computer science* we understand the science of what programs are, i.e., the study and knowledge of the properties of the artifacts that can exist inside computers: data and processes.

Although the borderline to computing science may be fuzzy, we do count as subjects of computer science those of computability and complexity theory [127], automata theory and formal languages [116], type theory [2, 88], foundations of algebraic semantics [20, 214], denotational semantics [191], operational semantics [183], axiomatic semantics [112, 13] and process algebras [110]. See also [96, 169, 213].

2.3 Computing Science — Programming Methodology

By *computing science* we understand the science of how programs can be developed, i.e., the study and knowledge of how to construct the artifacts that can exist inside computers.

We count as subjects of computing science, also called programming methodology, the disciplines of algorithms and data structures [156], of functional programming, logic programming, imperative programming, parallel programming and algebra programming, as well as those of formal specification and design calculi such as the broad-range techniques of VDM [31, 126, 75], Z [196, 197], Larch [100, 101], RAISE [92, 93] and B [6, 135, 217], etc., as well as the more calculi-oriented: *The Discipline of Programming* [64], *The Science of Programming* [90], *The Craft of Programming* [189] and *The Logic of Programming* [109, 91], including the refinement calculi-oriented [164]. A recent programming methodology discipline to receive seminal treatment is that of reactive systems [49, 146, 147]. The formal specification disciplines are best understood, as we shall claim, in a semantics setting [96, 169, 191, 60, 176].

Disciplines that straddle programming methodology and software engineering are typically covered in subjects such as operating systems [199], database systems [58, 59, 211], distributed systems [144], and data communication including protocols [193].

2.4 Software Engineering

Software engineering is characterised by disciplines, as we shall see it in this paper, which secures that the software fits the application domain (domain engineering), meets expectations (requirements engineering) and is otherwise correct (programming methodology). A central part of software engineering is programming. In addition software engineering is traditionally concerned with securing pragmatic issues such as efficient use of computing platform resources, trustworthy projects and products — including test-case generation and validation, version

control and configuration management, design-decision tracking, [adherence to] documentation standards, etc. Quality assurance and quality control is interwoven with all this.

2.5 Discussion

Individually many of the disciplines listed deserve separate courses. The way software engineering is being taught and covered in most text books (whose title predominantly contains the term 'software engineering'), does not harmonise with the expanding programming methodological topics of computing science. We usually find that most software engineers do neither practice formal specification nor design calculi, and that most — if not all — textbooks likewise skirt the issues prefixed 'formal'. If these text books do contain something about so-called "formal methods", then it is as a separate chapter tucked away towards the end.

The whole point of this paper is to show (i) that formal techniques (specification and calculation) are indispensable if one is to practice sound and professional engineering, (ii) that they — to us and to an increasingly growing number of professionals — are woven into almost all software engineering, (iii) that they are no harder than similar mathematics requirements of other engineering branches, and (iv) that there is no choice!

2.6 Professional Software Engineering

We believe that the following properties characterise the professional software engineer: (i) an education that covers the main concepts treated in this paper, (ii) a practice which apply the kind of techniques and tools covered in this paper and in an engineeringly sound and pragmatic manner, (iii) a reluctance to start on software development projects for which proper methods are either not mandated or for which no time has been set aside to ensure a reasonable adherence to correctness issues, and (iv) refusal to complete (sign off on) developments for which a reasonable minimum of correctness cannot be achieved.

We have used some 'hedges' above: sound and pragmatic, reasonable adherence, reasonable minimum.

The quotes brought in section 8.2 starting page 42 put the "ideals" of formal specification and design calculi into the context of the "reality" of software engineering.

3 Abstraction, Linguistics and Logic

3.1 Ambiguity

The first problem of specifications written in natural languages is *ambiguity*. The sources of ambiguity are diverse and sometimes very unexpected. Therefore, even if you are convinced that a sentence is *precise* (that is, not ambiguous), it is possible that a hidden ambiguity will later surface. A first obvious source of ambiguity is that some words have several, contradicting definitions. Two ambiguous words that we will be using in this paper are method and model. We will discuss them in a moment. When writing a specification, a usual solution is to include re-definitions of all terms, but those definitions themselves may be ambiguous, incomplete or circular. Sometimes it is not desirable to redefine the terms, but instead their use in different contexts can be systematically distinguished. A second common source of ambiguity is that sentences may be parsed in different ways. One well-known example is the sentence "time flies

like an arrow", which with some effort may be parsed as stating that a certain kind of flies has a predilection for an arrow. This illustrates how unexpected ambiguities can be. But even if a sentence is parsed in a unique way and all its terms are precise, its meaning may remain ambiguous. Just compare the signs (displayed at the foot of an escalator) "Shoes must be worn" and "Dogs must be carried". Does this mean that whenever I want to use the escalator, I must be wearing shoes and carrying a dog? Or is the meaning that whenever I have a dog and shoes with me and want to use the escalator I must carrying the dog and wearing the shoes? See [122] for a very good discussion of this example.

The words "model" and "methodology" are ambiguous. Since we will be using them often, let us make more precise. Model may be used in the sense of:

- Model: A model is an idea, a mental construct or a mathematical object (if you wish: a platonic object) that, perhaps tentatively, describes a system or presents a theory that accounts for some of the properties of the system. This is the sense in which the word is used, for instance, in "performance modelling" or "domain modelling".
- (Logical-theoretic) Model: In logics, more specifically in so called model–oriented semantics, a model of a formula or theory is a mathematical object that satisfies the formula or theory.

Notice that in a certain sense the two meanings are quite opposite: in the first one a model is a *generalization* an *abstraction* of the system, while in the second one the model is a conctrete instance where the formula or theory is true.

We avoid the ambiguity by using the adjective logical-theoretic, when a confusion may be possible. Usually we use the word in the first sense.

The word "method" will be discussed in section 4.8. We want to use the word in the strong sense of *a set of principles for analysing a problem, etc.* and not just for *away of doing something, a procedure*. In this loose sense of the word, formal methods are methods, but not in the strong one.

3.2 Abstraction

By *abstraction* we understand the human activity of constructing a partial description of an "object" (be it "reality" or a mathematical object, be it an actually constructed hardware- or software-piece or a set of expectations (requirements) that it should fulfill). Also, by abstraction we mean the result of this process of abstracting. The abstraction describes only some properties of the object, those that want to be emphasized as being important in the given context.

A program is quite a concrete representation of a certain mathematical object which may be viewed as a computable function, a discrete dynamical system, an abstract state machine, an algorithm, a (discrete) dynamical system or the like. The program itself may be compiled and it should run on appropriate hardware, but many interesting abstractions of the program may not be directly compiled into code.

We consider the ability to continuously find abstractions at all levels of development, in domain, in requirements and in software design engineering, as a most crucial property a leading software engineer must possess. We also find that most software development projects that we have witnessed fail to exhibit such clean abstractions. It seems to us that inability to abstract usually lead to severely limited software functionalities, to software with far too many "case distinctions", and therefore may be a first serious cause of erroneous software. That is: software

that fails to meet its requirements; requirements that do not fit to the domain description; and a description of the domain that misrepresents the 'real domain'!

3.3 Syntax

In programming and specification linguistics syntax is distinguished from semantics. The syntax is concerned with the structure of programs or specifications as strings, i.e., sequence of letters in an alphabet. Whether a certain string is a legal program or not and how to "connect" strings of legal programs to build larger legal programs is a matter of syntax. Semantics deal with what legal programs mean.

By *syntax* we understand the rules for forming inductively a language of finite strings on a particular alphabet from atoms. Depending on the use, the elements of the language are called *terms* or *well-formed formulas* or *programs*.

For instance, one could define two languages, **N** and **I**, in the following way. Define **N** as the smallest set with two properties:

- The string '0' belongs to **N**.
- If n is a string that belongs to **N**, then the concatenation of the string '$S.$' and n also belongs to **N**.

and define **I** as the smallest set with:

- Each string that belongs to **N** also belongs to **I**.
- If n is a string that belongs to **N**, then the concatenation of '$-$' and n belongs to **I**.

This may be written in Backus-Normal Form notation:

$$n ::= 0 \mid S.n \tag{1}$$
$$m ::= n \mid -S.n \tag{2}$$

These rules define the two sets of strings:

$$\mathbf{N} = \{0, S.0, S.S.0, \ldots\} \tag{3}$$
$$\mathbf{I} = \{\ldots, -S.S.0, -S.0, 0, S.0, S.S.0, \ldots\} \tag{4}$$

In this way, each element of **N** represents uniquely a natural number: the symbol 0 represents the zero and $S.n$ represents the successor of the number represented by the string n. Also in the same way, each element of **I** represents uniquely an integer number. But instead of thinking that the natural and integer numbers have nothing to do with strings, let us think of **N** as *being* the set of natural numbers and **I** the set of integer numbers.

In other words, we are identifying the meaning of the sequence '$S.S.0$' with the sequence '$S.S.0$' itself! We are identifying the syntax and the semantics of **N** and **I**. Later, we may think of 2 as an abbreviation of the sequence '$S.S.0$'. But this 'trick' of identifying syntax and semantics does not always work: firstly, on some domains, like **R**, not all values correspond to a syntactical expression, and secondly, because different syntactical expressions may mean the same thing. This is the case with programming languages.

Let us now introduce the syntax of a simple imperative programming language which we call **IMP**. Let first $\mathcal{V}ar$ be a set of letters that we want to call *variables*, and let X range over $\mathcal{V}ar$. (The variables X are intended to be integer valued). Let also m, m_1, m_2 range over integers.

The formation rules for **IMP** are:

$$a ::= m \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \tag{5}$$

$$b ::= tt \mid ff \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \tag{6}$$

$$c ::= skip \mid X := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c \tag{7}$$

This notation provides an inductive definition of three sets, which will be called, respectively, the *arithmetical expressions*, the *Boolean expressions* and the *commands* or *programs* of **IMP**. Actually, something is missing in the definition of the three sets of strings: parentheses. One says that the given Backus-Normal Form rules define the *abstract syntax* of the sets, while a second set of rules is still needed to complete the *concrete syntax*. For instance, orders of precedence between the operation symbols or rules for parentheses may be introduced to enforce that each string can be parsed correctly. Our convention is that we introduce enough parentheses to ensure that the expressions are built in a unique way.

For instance, the string of symbols:

$$Y := X; X := 1;\ \textbf{while } Y > 0 \textbf{ do } (X := X \times Y;\ Y := Y - 1)$$

is a legal program in the language **IMP**. Now we can ask ourselves what this program *means*.

3.4 Semantics

By *semantics* we understand rules for forming "meanings" of syntactic elements from the meanings of atomic parts.

There are many ways of presenting the semantics of programming or specification languages, the most common ones are denotationally [191, 96, 169, 213], axiomatically [112, 13], and operationally [183].

As an example, we will present the semantics of a set **IntExp** of integer expressions. Recall the definition we gave of **I**, the set of integers. If, instead, we would have let

$$m ::= 0 \mid S.m \mid -m$$

then we would have obtained a larger set of strings, including for instance '$-S.S.-S.S.0$'. This language (set of strings) is **IntExp**.

The basic idea in denotational semantics is simple: atomic syntactic constructs are ascribed some mathematical value (or function) as their denotation, while composite syntactic constructs have their semantics expressed as a function of the semantics of each of the constructs.

Although we can not *identify* **IntExp** with the set of integers, we may think of each element of **IntExp** as denoting an integer: 0 denotes the zero, $S.m$ denotes 1 plus whatever m denotes and $-m$ denotes the negative of what m denotes. This mapping $M : \textbf{IntExp} \to \textbf{I}$ is called the *denotation.* Thus '$-S.S.-S.S.0$' and '$S.-S.0$' denote both the integer zero: $M(\text{`}-S.S.-S.S.0\text{'}) = M(\text{`}S.-S.0\text{'}) = \text{`}0\text{'}$. The formal definition of M is easily given by induction.

Another way of giving semantics to **IntExp** by the set of rewrite rules [62]:

$$-0 \rightarrow 0 \tag{8}$$
$$-- \rightarrow \epsilon \tag{9}$$
$$S.-S. \rightarrow - \tag{10}$$

where ϵ is the empty string. The rules define a relation "$\rightarrow$" on strings. The rewrite rule $\alpha \rightarrow \beta$ may be applied to any string m containing the substring α by replacing α by β in m. If the resulting string is m_1, we write $m \rightarrow m_1$. Let $\rightarrow^*$ be the reflexive transitive closure of $\rightarrow$. Thus $m \rightarrow^* m_1$ iff m_1 may be obtained from m by (zero or more) applications of the rewrite rules. It is for instance easy to show that '$S.-S.S.--S.0$' $\rightarrow^*$ '$-S.S.0$' by applying the rules 9 and 10. It is not difficult to show that for each $m_0 \in$ **IntExp** there is exactly one $m_1 \in$ **I** such that $m_0 \rightarrow^* m_1$. In this sense, m_1 is precisely Mm. Moreover, the *structured operational semantics* of **IntExp** is to consider the rules 8-10 as defining a transition system where the vertices are elements of **IntExp**. There is a transition from n_0 to n_1 iff $n_0 \rightarrow n_1$. This transition system may be seen as a program which calculates the value of $n \in$ **IntExp** by rewriting it step by step until the calculation stops in an element of **I**.

Analogously, but somewhat more involved, is the definition of the semantics of **IMP**. First, let us ask ourselves what the denotation of program should be. There are several answers to this question, one possibility is to think of programs as changing the values of the variables. Thus, if we define a state as a valuation of $\mathcal{V}ar$ (a function from $\mathcal{V}al$ to **I**) then a program may be seen as a *partial function* from states to states. Thus if $prog \in$ **IMP**, $M(prog)$: $States \overset{partial}{\rightarrow} States$. (It is not a total function, since some programs starting on some states σ never terminate: the function $M(prog)(\sigma)$ is not defined). For example, $M(x := 3)(\sigma)$ is the state that differs with σ at most on the value of X, the first one being 3. We denote this state by $\sigma(X \leftarrow 3)$. Recall the program we had at the end of the last section: $Fac := (Y := X; X := 1;$ **while** $Y > 0$ **do** $(X := X \times Y;\ Y := Y - 1))$. Then, it is true that $M(Fac)$ is the function that maps the state $(X \mapsto n)$ (for $n \geq 0$) to $(X \mapsto n!)$, in other words, it is the factorial function.

To define an operational semantics, consider the set of configurations, i.e. pairs of the form $< c, \sigma >$ where c is a program **IMP** and σ is a state. The operational semantics gives a transition system on the set of configurations which describes how those configurations 'evolve'. For instance, $< X := 3; c, \sigma > \rightarrow < c, \sigma(X \leftarrow 3) >$. Of course, the interesting point is how to define the semantics for the **if** $-$ **then** $-$ **else** and, even more, for the **while** construct. Details can be found in [213].

An archetypical axiomatic semantics for a programming language like **IMP** is *Hoare logic*. Although today it is used more as a logic for proving correctness and properties of programs, it was also intended originally as a method to explain what a program actually does, in other words as a semantics.

Hoare logic, we think, plays a central rôle in the software education, since it can be taught from the very beginning, when students learn their first programs (even at high school). It is also possible to use informally (a version of) Hoare logic, just as programming languages are informally used to start with.

For instance, consider again the program *Fac* applied to an input $X = n \geq 0$. The program may be annoted as follows:

$\{X \geq 0\}\{X = n\}$ $Y := X;\ X := 1;$
$\{X = 1 \wedge Y = n\}$ **while** $Y > 0$ **do**
$\{n! = X \times Y!\}$ $X := X \times Y; Y := Y - 1;$ $\{X = n!\}$

Think of $\{b\}$ as asserting that when the program reaches this configuration the boolean expression b is true. The first one $\{X \geq 0\}$ is an assumption of the program. The next one $\{X = n\}$ binds the value of n to whatever the value of X is. To show that the next one is true we may rewrite $Y := X; X := 1$ as a relation on state and next state: $Y' = X \wedge X' = 1$. The proof obligation is then $X = n \wedge Y' = X \wedge X' = 1 \Rightarrow X' = 1 \wedge Y' = n$. This is trivial. Now, we need to show that the loop invariant $\{n! = X \times Y!\}$ is indeed true. Initially it is when we enter the loop, since $X = 1$ and $Y = n$. After one step, which may be rewritten in the form $Y > 0 \wedge X' = X \times Y \wedge Y' = Y - 1$, the invariant still holds, since $n! = X \times Y! = X \times Y \times (Y-1)! = X' \times Y'!$. After the last step of the loop, the invariant still holds and moreover $Y = 0$. Therefore, $n! = X \times Y! = X \times 0! = X$. A detail in the proof is missing: the loop *does* terminate. This is true since Y is always decremented by one within the loop; therefore, $Y = 0$ is true eventually.

The use of annotations, in the sense of pre- and post-conditions as well as in the sense of assertions on contracts, may be used, formally and informally, as a methodology for constructing programs from specifications (see [164, 90]).

We find that every software engineer must have a reasonable working understanding of each of the above-listed semantics definition styles.

The archetypical model–oriented semantics definition style is that of denotational semantics [198, 191].

Compilers can be systematically derived from denotational semantics descriptions [36].

3.5 Pragmatics

By *pragmatics* we mean: the reasons for choosing a certain framework (say, a formal language or a logic) over others, as well as the actual use of syntax. For instance, when representing a graph as an algebraic structure it is a matter of pragmatics to model a graph by (V, E) where V is a set of vertices and E (edges) as a relationship on V or as (V, E, I) where V and E are sets (of atoms) and I is an incidence relationship with certain properties.

Syntax and semantics can usually be formalized, but it is in the nature of pragmatics that it cannot be formalized.

Software engineers often dream the impossible dream of trying to construct programming or software concepts that "float" somewhere in a mixture of syntax, semantics and pragmatics. And many issues that the programmer would like to straight-jacket into a feature are of pragmatic nature and hence can basically not be 'incorporated' into the software.

It is therefore important that software engineers have experienced, during their education, enough examples of formal syntax and formal semantics in order to learn themselves why pragmatics is so elusive!

3.6 Type Theory

Type theory is one of the most important contributions that computer science has made to science.

Starting with the implicit or explicit simple and ("trivially") composite typing of variables and values of ordinary programming languages from Fortran to C, higher order types [88], such

as of functions (which may again take on functions as parameters), subtypes ordered in lattices and used for example in object-oriented systems (multiple inheritance etc.) [44, 43], and finally intuitionistic type theory [177], type theory is an exciting "universe". Getting the types right (determining the signatures) is "half the fun" and brings us a long way towards getting the software right.

3.7 Programming Languages for Structured Programming

Although few software engineers design new languages they are constantly confronted with new languages — often through hyped propaganda. It is therefore important that software engineers know of the various language paradigms: functional programming (Lisp [149], Standard ML [160, 195, 212], Miranda [205], Haskell [204]), logic programming [139, 125, 67], imperative programming [175, 107], parallel programming [113, 114, 158, 159, 119] and algebra programming [190].

In the area of imperative programming Dijkstra's concept of non-determinism, paired with his formulation of Hoare logic [112] in the concept of weakest pre– and strongest post-conditions, has become a de facto "standard" for thinking about programming and for devising semantics and certain refinement calculi [64]. Some 'programming' (called meta-programming by some) is done in terms of composing calls to various platform packages such as those provided by X11 Windows, Athena Widgets, ODP, OMG, etc. Even for their proper exploitation it is useful that these platforms be thought of a language interpreters.

"Real", commercial, so-called "industrial-strength" programming languages — such as CHILL [103, 11], Ada [36], C [129], C^{++} [61], Java [14] — as well as the more pleasing, equally powerful programming languages — such as Modula–3 [175, 107] and Oberon [215, 187, 216] — are all composed from various linguistic constructs: functional (expressions and procedures), logic (Boolean expressions, conditionals), imperative (variables, pointers, statements, gotos), and parallel (critical regions, semaphores, processes, synchronisation and communication).

Instead of training future software engineers in the commercial, "industrial-strength" programming languages, we advocate teaching them a set of paradigmatic languages now listed. Together with each language one then has the possibility of also teaching its foundations.

In addition to university teaching, each of the languages mentioned below has given rise to extensive research. New language paradigms emerge "continuously". In order for the future software engineer to be able to cope, during the active years after graduation, it is probably wisest to expose them at university to the latest "academic, paradigmatic" languages and their foundations. We find that doing so better enables the software engineer to teach themselves any new programming language (Ada, Java, whatever).

3.7.1 Functional Programming Languages

In functional programming one deals with variables (and constants) of various types, with general arithmetic, conditionals, patterns, function abstraction and function application.

Examples of functional programming languages are: LISP [149], Standard ML [160, 195, 106], Miranda [205] and Haskells [204].

Underlying theories include Type Theories [2, 88, 43, 44] and Recursive Function Theory (including the λ-calculus and Computability) [130, 17, 127].

3.7.2 Imperative Programming Languages

In imperative programming one sequentially "programs" a changing state: assigns values to variables, manipulate pointers, etc.

Most commercial programming languages, from Fortran via Cobol, to CHILL, Ada, C, C^{++} and Java are basically imperative. That is: are centered around a concept of statements, where sequential execution of simple and structured statements changes the state.

With Algol, Pascal, Modula (–3) and Oberon we had and have some rather elegant languages — and with Dijkstra's 'language' of non-deterministic constructs (and guarded commands) [64, 65] we have imperative languages which have spurred the academic interest in proof systems and techniques of proving properties of imperative programs [172, 76, 112, 13, 175, 63, 64, 65, 66].

Perhaps one of the most important and pervasive notions of imperative programming is that of Dijkstra's weakest pre-conditions and strongest post-conditions, etc. [64, 65, 66]. Their importance in language design and in refinement calculi cannot be over-estimated.

3.7.3 Logic Programming Languages

In logic programming one "programs with truth values" (through so-called resolution theorem-proving).

Prolog is the quintessential example of a logic programming language [139]. Additionally constraint logic programs can be processed by such systems as CLPR (Constraint Logic Programming / [IBM Yorktown] Research) [125], Chips [67], etc.

Basic logic programming centers around a first order propositional calculus, but teaching logic programming provides a good basis for teaching mathematical logic, say first-order predicate-calculus.

3.7.4 Parallel/Process Programming Languages

In parallel programming we define and use processes: we invoke and terminate them, put them in "parallel" (in various ways: "true parallelism", non-deterministic external choice, non--deterministic internal choice, etc.), synchronize them (and then are able to exchange data between them: communicate).

Examples of "academic, parallel languages" are CSP (Hoare's Communicating Sequential Processes) [113, 114], CCS (Milner's Calculus of Communication Systems) [158, 159], and (varieties of) Petri Nets.In the π-Calculus [162, 161] (the so-called mobile) processes are first class citizens that can be communicated!

CSP serves as the basis for the industrial-strength, but nevertheless very clean occam language [119].

Besides the CSP, CCS, π-Calculus and occam theories, there are also those more specifically regarded as process algebras [110].

3.7.5 Algebra Programming Languages

This kind of programming is less well known in comparison to those previously mentioned.

In algebra (not algebraic) programming one considers algebras as data types and compose and decompose them in various ways [190].

The underlying theory is basically that of category theory [111, 18, 190].

3.8 Natural Language Linguistics

But not everything must or can be formally expressed, and formal expressions should usually also always be annotated, i.e., explained informally.

Ability to command one's own language, and being able to express things succinctly, finding and using appropriate didactic devices, all these are important properties that leading software engineers must possess. Add to this the ability to analyse other persons ambiguous and fuzzy descriptions, identify the crucial verbs and nouns (i.e. the alphabet), separate pragmatic concerns from syntactic and semantic ones — the leading software engineer must be quite a person!

Since, as we shall see shortly, specifications, of domains, requirements and software designs, usually must be both informal, in natural language, and formal, in some well-founded specification language, and since the informal part usually ought contain a clear and terse terminology, it is important that the software engineers who are leading the development team master their language.

It is not only AI (artificial intelligence) that resort to linguistics and language philosophy. In domain engineering and in requirements engineering many linguistic issues (of phenomenology, epistemology and ontology [94, 53]) arise.[2] Related modelling often is expressed in some exotic logic or other [46, 150, 151, 155, 152, 78, 79, 80, 81, 188, 138, 137, 155].

Insight into linguistics, including computational linguistics, and related logics is therefore deemed indispensable: professional software engineers must be well educated in these areas.

4 Methodology, Formal Specification and Design Calculi

Pervasive to all software engineering is the use of compilers and hence of programming languages. Core activities of a software engineer is specification and programming. The next subsections will review these.

4.1 Compilers

The most basic kind of software technology is that of compilers. Since every kind of software technology is ultimately based on some programs, and since these must pass through a compiler it is of utmost importance that we get the compilers right and that the software engineer understands them properly.

Although only few software engineers actually develop compilers, to properly understand modern compiler writing is, we believe, an essential prerequisite for any software engineering.

Every software engineer must be thoroughly exposed to not only the semantics of programming and machine languages, but also to proper compiler development as it is not only of core importance, but also because compiler development illustrates a number of formal techniques and tools.

Section 5.6 brings one perspective of what we mean by getting it right. Later sections bring more.

[2] We refer to the Stanford University Centre for the Study of Language and Information (CSLI): `http://www-csli.stanford.edu/publications/` home page.

4.2 Specification

Small programs are *relatively* easy to program without errors because the (single) programmer knows exactly how each part of the program (each instruction) fits into the whole system. Also, the software engineer knows exactly what each instruction does. In large systems the situation is much different: the chief-designer may perhaps have an intuition, a mental model or even a description of how the different parts interact (say, as running tasks in a complex operational system) and the software engineer may also have a model of each of those individual parts, but the likelihood that a subtle error remains undetected increases with the complexity of the system. The purpose of software engineering is to provide the user a set of methods and tools to be able to design correct systems, even if they are large. How can this be done? Well, the problem resides in the facts that:

a) the description that the designer has of the parts of the system (and of their running environment) are informally described and *ambiguous*,

b) the developing team has no way of checking that a module indeed satisfies the informal description that it is supposed to satisfy,

c) the developing team has no infallible (i. e. sound) means of drawing conclusions from the ambiguous descriptions of the individual parts.

Therefore, one way of achieving the goal of software engineering is first to *formalize* the description ("specification") of the components and of their interaction and then to provide techniques (calculi) and tools to *prove* that the parts of the system satisfy their specification and that from those specifications and the model of the interaction it follows that the whole system satisfies its requirements. Also partial solutions are welcome. Even if a formal notation is not chosen, it is still possible to disambiguate the specifications.

Specifying is the process of writing those descriptions precisely. Precise, even informal specifications are already useful: the developers understand better their system and discover inconsistencies, errors and ambiguities. They also document the design and help to communicate with the customer. But, without tools, precise specifications may still hide incongruencies or errors. Practice has shown that any effort in specifying rigorously (or formally) helps. How much formalization is needed is in many cases debatable, in principle it is possible to have clean and precise mathematical models and to do mathematical proofs without choosing the formal notation or calculus to start with. This point of view is advocated with some success by the Gurevich's abstract state-machines approach (see [97, 98, 99, 39, 40]). In many cases, however, a computer supported verification method based on formal calculi seems to be absolutely necessary. Without tools, precise specifications and proofs may still hide incongruencies or errors.

We shall generally be using the term 'specification' to cover domain, requirements and software models. But for now let us just use the term 'specification' in connection only with requirements and software designs. A requirements specification describes **what** the software is expected to offer — not how. A software specification, in contrast, describes various levels of abstraction of **how** the software provides the offer. At the requirements level the software engineer does not have to bother with concrete things such as executability of specifications. Instead the software engineer is encouraged to abstract so as to capture as succinctly as possible

the **what**. In section 5 we shall have much more to say about abstract specifications, including those of domains.

Suffice it here to state that abstract, both informal and formal specifications related to software have come to stay and that todays professional software, like professional control engineers, chemical engineers, etc., must master many different specification languages and their associated design calculi.

Examples of specification languages (with associated design calculi) are: VDM [126, 75], Z [196, 197], Larch [100, 101], RSL [92, 93], Duration Calculi [49], B [6, 135, 217], Temporal Logics [146, 147, 49], and TLA [133].

4.3 Programming

Programming is here used in its broadest connotation and covers the act of contriving pleasing domain, requirements and software architecture specifications as well as the act of contriving efficient code.

Although it is in principle possible to write structured assembler code, even for large programs, as time passes this code tends to become unstructured. (Legacy code).

It is important that programs are right: meet their specifications functionally (including relate to previous development steps), and that software designs are otherwise efficient, easily modifiable, etc. Techniques of programming are therefore of crucial importance.

Programming, as well as domain and requirements-model development, typically proceeds in steps of so-called refinement: from abstraction towards concretizations. Programming at the software-design level usually implies the creation (discovery, invention, reapplication) of an efficient algorithm in order to solve a computational problem. Programming at the domain modelling or at the requirements modelling levels usually have to do with introducing, one-by-one concepts such that complexity is "conquered" through separation of concerns.

In a sense one can claim that it may be straightforward to specify a computational problem, but that it may require very special abilities to come up with an efficient algorithm and program to solve that problem. Using formal techniques is no substitute for algorithmic cleverness, but it is always wise to also formally document both the problem and its solution — in stages of derivation.

Some of todays software engineers seem to excel in being "clever" wrt. algorithmic matters, but to being "stupid" in failing to properly document the algorithmic ideas and that the algorithm indeed does solve the problem as posed.

Although we may claim that requirements and software architecture specification may yield higher returns on investments as compared to algorithm specification, we nevertheless advocate formal, or at least rigorous, stepwise refinement throughout! If you have to specify the "grand things" you might as well also specify the "nitty-gritties". It would be sad if a grandiose architecture failed because of an erroneous algorithm.[3] Which parts of the developemant process may or can be specified in which level of formality is a question of pragmatics. Given the current state-of-the-art, it is still quite costly to formalise and verify the complete design of a large system. This of course depends on the availability of tools and people able to use them.

Sections 5.3–5.5, and 6 deal with this central matter.

[3] A Kingdom was lost for the want of a horse. William Shakespeare, Richard III

4.4 Verification

To verify steps of development, whether within each of the three main software-development phases (domain, requirements or software-design engineering), or between them, and to otherwise prove properties of individual specifications, including executable programs, is important. If not already, future clients will increasingly demand that delivered software meets certain quality standards wrt. various forms of correctness. The only way we know how to assure this to highest attainable degrees of confidence is to deliver formal verification.

Let $\mathcal{D}$, $\mathcal{R}$ and $\mathcal{S}$ stand for the theories of the domain, requirements and software (or more generally, machine). Then verification means that $\mathcal{D}$ and $\mathcal{S}$ imply $\mathcal{R}$, i.e., that the designed software *satisfies* the requirements in the presence of a theory about the domain. This proof obligation, ($\mathcal{D}$, $\mathcal{S} \models \mathcal{R}$), is well known. For complex systems we are still far away from being able of proving this statement to a full extent. But it is sometimes possible to prove (with Model Checking or Theorem Proving) that for some abstract versions of $\mathcal{D}$, $\mathcal{S}$ and $\mathcal{R}$, the obligation indeed holds [54].

4.5 Model Checking & Model Checkers

Some formal methods (Z, VDM, Larch and others) are appropriate for programs with rich data types but a simple control structure (for instance, sequential while-programs), others (CCS, CSP, Temporal Logic, Automata, etc.) may be used for concurrent or reactive systems but support only simple state spaces. Still others (RAISE, LOTOS) combine methods for concurrent systems with complex data structures. In the case that the state space is simple and finite, or the data complexity may be abstracted away, it is possible to construct a finite transition system which serves as a formal representation of the system. By construction, the program *satisfies*, *implements* or *refines* the transition system in the sense that any behavior (trace) of the program may be viewed as a trace of the transition system (by appropriate abstraction of data and of "irrelevant" internal steps).

Then, this finite transition system is a formal specification. Model checking algorithms use an efficient space search over the states (or transitions) of the transition system to prove (check) that it satisfies (is a logical-theoretic model) of its specification. The specification may be formalized as a temporal logic formula or also as a sort of transition system, like automata or Kripke Structures.

In symbolic model-checking the state-space search is not done by inspecting states individually (i.e., valuations of all variables) but by inspecting at once sets of states (i.e., predicates on variables). Binary decision diagrams (BDDs) offer a compact data structure for representing predicates on finite sets.

In the following example we consider a pair of traffic lights on a street intersection. The crossing is sketched in figure 1 page 20, a). In figure 1 page 20, b) we describe a property that holds in $\mathcal{D}$, i.e., a property of the domain. It simply states how cars behave: a car may legally cross (event Car happens) only if the light is green, or more precisely, a car may cross from South to North (Car_1) only if the first light is green and cars may cross from East to West (Car_2) only if the second light is on. Do not think of this figure as telling what *must* happen, but only of what *can* happen in any state. In particular, a car does not have to cross, even if the light is green. For simplicity, we further assume that initially, when the system is installed, the cars in the South-North direction may cross, while the others must wait for the $Green_2$ event.

So much for properties that hold in $\mathcal{D}$.

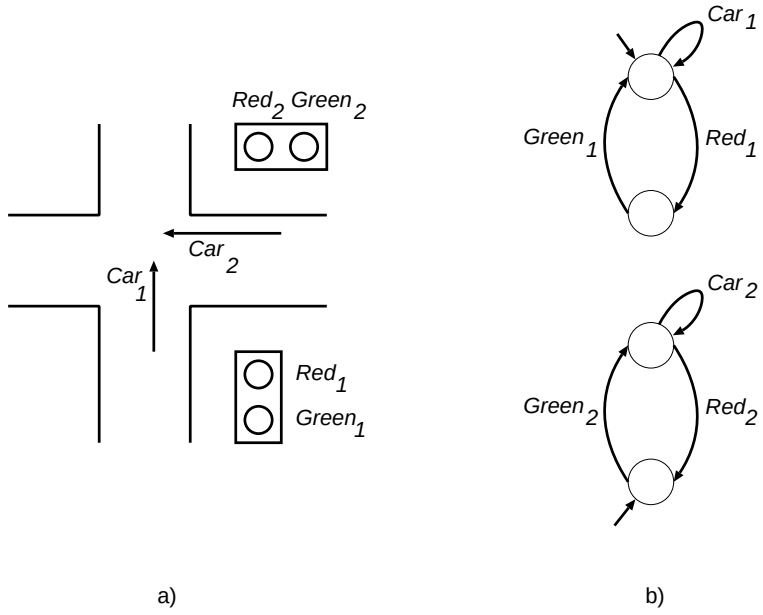


Figure 1: *a)* A Street Intersection and *b)* the Domain Specification

Let us now look at $\mathcal{R}$, the requirements. We want, among other things that will be irrelevant here, that both events Car_1 and Car_2 never happen simultaneously. Thus, for the sake of this example, this is all $\mathcal{R}$.

Let us now assume that we know $\mathcal{S}$, the specification of the machine, or as we call it, of the *controller* of the lights, and that it satisfies the properties that always $(Green_1 \Leftrightarrow Red_2) \wedge (Green_2 \Leftrightarrow Red_1)$, in words: each time the controller turns the first light to green it also changes the other one to red and vice-versa. Thus we assume that this property follows from $\mathcal{S}$.

It is quite obvious why in the composed system in no reachable global state cars in both directions may cross. This is true because if one of the two automata of figure 1 is in the first state then the other one is in the second state and vice-versa. This may be formalized for instance by the statement that $g_1 \Leftrightarrow r_2$ is an invariant of the system (i.e., it holds in all reachable states), where g_1 and r_1 (resp. g_2 and r_2) are the first and second state of the first (resp. second) transition system. This analysis, based on a search of the reachable states, can be efficiently mechanized, even for quite large systems with 10^{20} reachable states.

Notice a subtle point of the argument that we have presented. It may be argued that the description $\mathcal{D}$ of the domain that we gave is incorrect: indeed a car *can* cross when the light is red! But then if an accident happens it was not the hardware/software controlling the light to be blamed for it. It is the car driver. Well don't worry: the meaning of the event Car was not simply that a car crosses, it was that it crosses on a green light. We may see the triple $\{\mathcal{D}\}\mathcal{S}\{\mathcal{R}\}$ as stating a *contract* (see [16, 157]): we are to make sure that if the environment behaves as stated in $\mathcal{D}$, then the whole system will behave as stated in $\mathcal{R}$. If that is the case for all possible "environments" which satisfy $\mathcal{D}$, we say that the "Hoare triple" $\{\mathcal{D}\}\mathcal{S}\{\mathcal{R}\}$ is true.

4.6 Theorem Proving & Theorem Provers

Depending on the semantics that you associate with a program, and in particular with the logic (description calculus) that you choose, to reason in that semantical world, you may decide which theorem prover system meets your needs. If the formal model of the system is a (possibly infinite) transition system where the states are given by valuations of variables, a theorem prover may be used much in the same way that a symbolic model-checker is used. Instead of representing predicates on variables as binary decision diagrams (BDDs), first-order (or higher-order) logic predicates are used [182, 42, 180, 54].

4.7 Synthesis

Let us return to the system of two lights on a crossing, figure 1. Now regard the domain and the requirements ($\mathcal{D}$ and $\mathcal{R}$) to be given, and we search for a controller $\mathcal{S}$ with $\{\mathcal{D}\}\mathcal{S}\ \{\mathcal{R}\}$ We do not assume, as we did in Section 4.5 that the *Green* events of one light correspond to the *Red* events of the other. This was part of the theory of $\mathcal{S}$. Now we know nothing about $\mathcal{S}$, except that it should satisfy $\{\mathcal{D}\}\mathcal{S}\{\mathcal{R}\}$.

Let us now look at figure 2 as being a game board. Two players, the *controller* and the *environment*, take turns in playing a game.

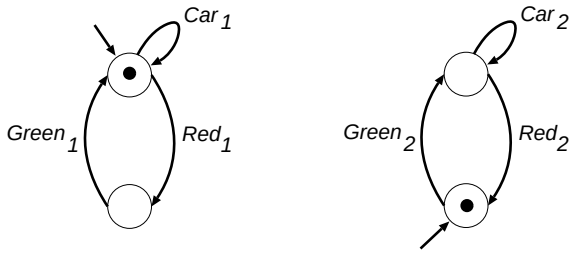


Figure 2: A Game between Controller and Environment

As we will see in a moment, each move of one of the two players, will define a set of events, subset of $\{Green_1, Green_2, Red_1, Red_2, Car_1, Car_2\}$. The score of the game at any finite time is a finite sequence of sets of events. After each move the new score is the result of appending the set of events to the current score of the game. Imagine the game as running until a winning condition is detected or else the game runs for ever. Even in this case, one of the players wins the game (at "infinity"). In our example, the controller looses the game if two cars in different directions cross at the same time, i.e. if both Car_1 and Car_2 happen simultaneously in one turn, or if one of the lights remains red forever, i.e. if never $Green_2$ happens or if after the occurrence of Red_i the corresponding event $Green_i$ never happens.

Formally, a winning condition for one of the players is a language of infinite strings. In our example the controller wins the *safety* game iff Φ_1 is true, and it wins the *safety-and-liveness* game iff Φ_2 is true:

Let:

$$\Phi_1 \ := \ \{< E_1, E_2, \ldots > | \text{ for all } \ i : \ \{Car_1, Car_2\} \not\subset E_i\}$$

$$\begin{aligned}\Phi_2 \quad := \quad & \Phi_1 \cap \{< E_1, E_2, \ldots > | \text{ there is an } \ i \text{ with } Green_2 \in E_i \text{ and} \\ & \text{for all } \ i: \text{ if } \ Red_1 \in E_i \ (resp., Red_2 \in E_i) \text{ then} \\ & \text{there is a } j \geq i \text{ with } Green_1 \in E_i \text{ (resp. } Green_2 \in E_i)\}\end{aligned}$$

where $E_i \subset \{Green_1, Green_2, Red_1, Red_2, Car_1, Car_2\}$ for each $i \in \mathbf{N}$.

In other words, the controller wins the safety game if never two cars cross simultaneously (in different directions), and it wins the safety-and-liveness game iff besides winning the safety game it never blocks a car forever by not giving him the opportunity of crossing the light. Of course, the controller perhaps wins a safety game because he was lucky: although an accident could have happened, it didn't (car drivers were more prudent driving than we were in constructing the software). This is not what we want: we do not want to win a game, we want to win *all* games (if the environment behaves as it should). We want a *strategy* for winning all games.

Place, initially, one token on the initial state of each graph. Now, the two players, controller and the environment, take turns moving the tokens: if a token is on a state s and there is a transition from s to s' labelled by E, then the player may decide to choose the transition, in which case the event E "happens" and the token is moved to s'.[4] In the case of several graphs, each token may moved independently of each other in the same turn: the union of the corresponding events happens. It is also possible that a player decides not to move one or several (or all) of the tokens.

Let us call the events Car_1 and Car_2 *uncontrollable*, and $Green_1, Red_1, Green_2, Red_2$ *controllable*. This means that the controller can only influence directly the occurrence of the events $Green_i, Red_i$, but the environment "decides" when the events Car_i will happen. (Of course, the event Car_i can only happen when a token is in the state which enables this action, that is when the corresponding light is green.)

Now, it is clear that the problem of constructing a correct controller is equivalent finding a strategy for the controller to win the game.

The most general strategy for the controller to win the safety game is to follow the recipe given in figure 3. In other words, let the controller start at the initial state of the figure, and let it decide (on non-specified way, perhaps based on the information given on extra sensors or general traffic considerations) when and in which direction it will move in the graph.

But observe that this strategy does not win the safety-and-liveness game: one possibility for the controller is to move back and forth turning the first light to red and green again. Although this controller is safe (two cars will not collide), it is not fair: a car may "starve" on the second light. To win also this game, "fairness" (or "acceptance") conditions have to be imposed to the graph. In our case they simply state that each of the two states upper-left and lower-right of figure 3 have to be visited infinitely often. Notice that this example shows that $*-$languages do not suffice for this type of applications: any finite sequence is unfair. This is why this sort of games is treated over $\omega-$languages (infinite sequences).

There are algorithms for finding these solutions (see [203, 186, 201, 202, 131]) that can be integrated into a model-checking/theorem-proving environment, as done in the TLT-project (Temporal Language of Transition, see [56, 57]) and in the subsequent SCSL-project (Synchronous Control Specification Language) at Siemens.

[4] In more general games, a transition may be labelled by a conjunction $\wedge E_i$; if the transition is taken, all the events E_i happen simultaneously. The type of automata considered here are labelled by elements of a Boolean algebra, as in [132] and [55].

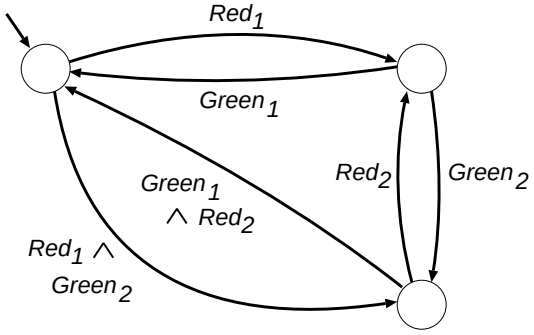


Figure 3: The most general control strategy

4.8 Methods and Methodology

Such terms as 'formal methods' are increasingly frequently used. Hence:

- **Method:**

  By a method we understand a set of principles for analysing a problem and for selecting and applying techniques and tools in order efficiently solve the problem.

The principles are presented in the form of natural (programming methodological) language statements such as:

> Example: **Principle of State Identification:**
>
> *In specification (say, of an aspect of the world or a computing system) one focuses on the observable, including the controlled variables. A finite, or at least an effectively enumerable alphabet representing these variables, call them the state components, must be identified.*
>
> *Each identifier of the alphabet must be given a type, i.e. the signature of the state must be established. Finally, as we can only know of the state through the observations we make when functions and operations are applied to it the alphabet and signature of these functions and operations (observers and generator) must likewise be established.*

or:

> Example: **Principle of Representational Abstraction:**
>
> *Components (say, of a state) are representation abstractly specified when the specification emphasizes properties of the components when subject to operations (rather than some arbitrary set-theoretic structuring).*

Model-theoretic, rather than, as just above, proof–theoretic (i.e. property-oriented) can also be representationally abstract if the mathematical type "fits" the desired properties.

Thus a telephone directory may be representation abstractly specified by some map (i.e. finite domain function), whereas a balanced binary tree representation may be too concrete.

Example: **Principle of Operational Abstraction:**

A function is operation abstractly specified if it emphasizes the function in intension rather than the function in extension, that is: the properties that relate function arguments to function results rather than how the function is computed.

Some principles, like the first of the above, if followed, are intended to help the developer identify state components through Analysis. Other principles, like the second and third above, provides techniques which are intended to help in abstractly expressing components, for example the state. These principles help to achieve efficiency in development.

Other principles are associated with specific techniques and tools and therefore identify these and gives guidance as to their use. Other principles are mere advice on whether or not to choose a certain emphasis, perspective or view in specification or design.

- **Methodology:**

  By methodology we understand the study and knowledge of methods.

No interesting software development (including domain, requirements and software-design development) can today take place without resorting to several methods (and any of these would, in an engineering environment be infeasible were it not for adequate, industrial strength tools). Methods (perhaps we should call them meta–) principles help us choose among varieties of methods.

- **VDM:** [31, 32, 126, 75]

  Perhaps the first so-called method was that of VDM: the Vienna (Software) Development Method. It emphasizes a model-theoretic approach together with the definition of functions and operations either explicitly or in terms of pre–/post-conditions. First applied to compiler development VDM has shown useful in a broad range of applications.

  We refer to the following `WWW` source of VDM information:

  – `http://www.ifad.dk/vdm/vdm.html`.

- **Z:** [196, 197]

  Perhaps the second specification method was that provided by Z (Z as in Zermelo, the mathematical set theoretician). Z provides rather elegant specification means.

  We refer to the following `WWW` source of Z information:

  – `http://www.comlab.ox.ac.uk/archive/z.html`

- **RAISE:** [92, 93]

  RAISE stands for Rigorous approach to Industrial Software Engineering, and covers a method, a specification language, RSL, and tool sets. RAISE is a further development of VDM and includes features and techniques for concurrency and modularization.

  We refer to the following `WWW` source of RAISE information:

  – `http://dream.dai.ed.ac.uk/raise/`.

- **Duration Calculi:** [49, 52, 51, 47, 48, 117, 50, 118]

  The duration calculi is a set of closely related formal systems for dealing with temporal notions: time, time intervals, time instances, etc. Applications focus on real-time and safety-critical systems.

- B: [6, 135, 217]

  Jean–Raymond Abrial, who initiated Z, has researched and developed another, very elegant approach called B (for B in Nicholas Bourbaki, the French group of set theory mathematicians).

  We refer to the following `WWW` source of B information:

  – `http://www.comlab.ox.ac.uk/archive/formal-methods/b.html`.

There are many other Methods: Larch [100, 101], STeP/React [146, 147], etc.

We refer in general to the extremely comprehensive set of information on formal methods given by the formal methods community home page:

- `http://www.comlab.ox.ac.uk/archive/formal-methods.html`.

Because of the different ranges of applicability it is important that software engineers have working knowledge of two or more of "the most diverse" (viz. RAISE, B and Duration Calculi)!

The above discussion is very simplified: The refinement calculi covered in section 4 can just as well qualify as methods.

4.9 Systematic, Rigorous and Formal Development

A formal notation may be used in many ways in a software development project. The software development may be characterised as proceeding in either a systematic, rigorous or formal manner — all depending on the extent to which the underlying formal notation is exploited in documenting or reasoning about properties of the evolving descriptions. Also it is possible to be more formal in one aspect (or view) of the system, for instance in the control flow, and to treat more abstractly some other views, for instance the data dependencies. Which parts of the system should be treated how formally is also a matter of pragmatics.

- *Formal Notation:*

  By a *formal notation* we understand a language with a precise, mathematically defined syntax and semantics and, eventually, a proof system. The first purpose of the formal notation is to describe or model real objects. Also the formal notation may provide means to express *properties* of the models of the real objects.

- *Formal Systems:*

  By a *formal system* we understand a formal notation together with a *design calculus* given by a set of syntactic *rules* for converting expressions of the formal notation into other 'derived' expressions. For instance, the expressions are interpreted as properties and the derivation as the construction of semantically equivalent or implied properties.

Traditional engineering fields typically use formal notation in the form of classical mathematics: calculus, differential and difference equations, linear algebra and matrix calculi, transformations, etc. Often the mathematics is presented graphically for ease of comprehension. The purpose of those languages is primarily to describe the physical reality. The difference between physical reality and model is quite clear. This gap is bridged when a design is used to construct a technological object.

In software engineering the "reality" that we want to understand and describe precisely is two-fold: on one side we have the application domain and the requirements specification and on the other the hardware/software system that we are designing. We start as in other engineerings by describing the application and the requirements mathematically. The novel aspect of software engineering is that those models themselves evolve to become programs, which are the technological objects that we want to construct. In this sense the clear distinction between model and reality is not exactly the same.

In all engineering fields the transition between mathematical models or engineering designs and actual technological constructions, i.e., the gap between models and reality, is impossible to bridge formally. In practical engineering projects this gap can be postponed sometimes until the end of the design phase. This is due to the fact that the corresponding engineering discipline has already a well established set of domain models, and engineers are trained in their use. In software engineering this gap between reality and model appears already at the beginning of the development process. It is precisely the biggest mental transition when we formally specify, i.e., abstract the application domain (see section 5.1.1) and when we "translate" user expectations into formal requirements (see also section 5.2).

Each of the many engineering fields has several sets of mathematics-based calculi. They are normally used to find correct dimensionings or to check designs. Those design calculi "model" physical quantities approximately.

The minimal use of formal notation is to use it as a description medium:

- *Systematic Use of Formal Notation:*

  By a *systematic use of formal notation* we understand a use of the notation in which we follow the precise syntax and semantics to formally describe (an abstraction of) a domain, requirements, design, etc.

Just formally specifying a part of the problem, or of the domain, or the requirements, or a software architecture, or a program organization, has shown to lead to cleaner developments with far fewer bugs, as has been documented in for instance [179] and in many papers of European formal methods conferences: [33, 38, 30, 185, 136, 171, 86, 74].

On the other extreme, formal notation may be used as a straight-jacket:

- *Formal Use of Formal Notation:*

  By *formal use of formal notation* we understand a systematic use in which we fully exploit the formality of the design calculi by actually proving properties using only the syntactical form of the rules in it. No appeal to the meaning or semantics of the formal notation is allowed to show the validity of a deduction.

In which aspects of the design process you really need this "straight-jacket" is a matter of careful evaluation. Without appropriate tools this can be too expensive when the complexity is

large. The intended meaning of the design calculi may sometimes be used in a more informal way, just as in common mathematical arguments:

- *Rigorous Use of Formal Notation:*

  By a *rigorous use of formal notation* we understand a systematic use of notation in which a mathematical understanding of the semantical models behind the scene is used to state and prove properties of the system. Although the derivations are expected to be correct, their correctness does not immediately follow the syntactical form of the rules in a design calculus, but rather by a mathematical, semantical argument.

4.10 Software Development

To us software development consists of three major components: domain engineering, requirements engineering and software design. Together they form software engineering.

Perhaps the term 'software engineering' is too restrictive. Since any implementation of especially a larger software system entails procurement also of hardware, development will also include configuration and acquisition of hardware components. That larger concept that includes the development, procurement, installation, performance tuning, operation and disposal of computing systems (hardware + software) is *systems engineering.* Thus, software engineering is part of systems engineering.

The aim of software development is to create software that is to function on some hardware. Together we call the software/hardware the machine ([122, 218]). Since domain engineering and requirements engineering aim at descriptions that may eventually lead to procurement of both software and hardware we shall refer to software development leading to a machine.

4.11 Discussion

In this section we just summarise what has been put forth in the preceding text, and which will be assumed for all the succeeding text also: Namely that development, from domain, via requirements to software design, in addition to being recorded in succinct, informal synopses, terminologies and narratives, will also be formally recorded, and that these formal specifications be subject to calculations (proofs of properties etc.). The term design calculi may be unfortunate in that it not only applies to software designs but also to domain and requirements models.

The best way we know today to achieve trustworthy software is through the use of formal techniques such as formal specification and design calculi. It is the same in other engineering fields. They also use mathematics. But the case of software is special in that we are not producing tangible artifacts based on laws of natural sciences. We are producing descriptions "galore": of domains, of requirements, of software architectures, of program organizations, and of increasingly concrete, "executable" code. The only laws they can possibly satisfy are those of mathematics and in particular mathematical logic. Other engineering design artifacts may satisfy laws of natural sciences. Software engineering is unique in that "all it produces" are textual descriptions and verifications, which are also textual structures.

Perhaps an essence of software engineering is the repeated construction, manipulation and analysis of very large structured texts.

5 Domain, Requirements and Design Engineering

In this section we outline a main message of this paper: namely that software development has three main phases: the development of domain models, of requirements models, and of software (designs and code).

In the first three subsections first we look at these three phases. To drive home this point we then present a classical example, namely that of a compiler for a given source "high-level" programming language and a target, say machine language.

5.1 Domain Engineering

5.1.1 Domain Engineering Concepts

In domain engineering we wish to understand first the application domain in which the software is to serve.

Two approaches seem current in today's 'domain engineering': one which takes its departure point in model–oriented, mathematical semantics specification work (and which again basically represents the 'algorithmic' school), and one which takes its departure point in knowledge engineering — an outgrowth from AI and expert systems. In this paper we focus on the former approach.

By an *application domain* we understand an area of activities, that contains a more or less clearly defined world of conceptual objects and actions on those objects. The application domain is the stage where the customer presents his requirements and the developer presents his solution. Examples of domains are: railways, air traffic, road transport, or shipping of a region; a manufacturing industry with its consumers, suppliers, producers and traders.

Since we are developing software packages that serve in these domains it is important that the software developers are presented with, or themselves help develop precise descriptions (models, see later) of these domains.

The domain typically includes the *system*, the *environment* and the *stake-holders*.

In the sense we use the word, a *system* is a set of interacting elements organized or created by people in order to provide some functionality. A system can be for instance an enterprise. Within a system, the purpose of a software program is to provide part of this functionality. And in many cases, the software/hardware machine is intended to replace a part of the old system.

A railway System consists of the railway net (lines, stations, signalling, etc.), the rolling stock (locos, passenger waggons, freight cars, etc.) and trains, the time tables and train journey plans, etc.

The *environment* is that part of the perceived world which interacts with the system.

The environment of a railway system includes the weather and the topology of the geographical areas.

By *stake-holder* we mean any of the many kinds of people that have some form of "interest" in the (delivered) machine: enterprise owners, managers, operators and customers of the enterprise. The *client* is the legal entity which procures the machine to be developed. A financial enterprise client is usually the appropriate level executive who specifically contracts some software to serve in the enterprise. The *staff* is the group of people who are employed in, or by the system: who works for it, manages, operates and services the system. The *customers* are the people or

companies who enter into economic contracts with the *client*: they buy products and/or services from the client.

Domain engineering is the art of establishing models of the domain. It is roughly divided into *domain acquisitions* and *domain modelling*. The first part is characterised by discussions with the stake-holders, or with other specialists on the domain. The second part is the process of writing down, in both informal and formal notations, the domain model.

The domain capture process, when actually carried out, often becomes confused with the subsequent requirements capture process. It is often difficult for some stake-holders and for some *developers*, to make the distinction. It is an aim of this paper to advocate that there is a crucial distinction and that much can be gained from keeping the two activities separate. They need not be kept apart in time. They may indeed be pursued concurrently, but their concerns, techniques and documentation need be kept strictly separate.

The informal description typically consists of a synopsis of the model, a terminology in which every professional term is defined, and a narrative which — in a readable style — describes how the terms otherwise relate. The formal model is then expressed in some formal specification language and can be subject to calculations using a design calculi of that notation.

By *domain analysis* we understand informal and formal analyses of the domain and of the resulting model — whether informal or formal. The purposes of the analyses can be to ascertain whether a component and/or its behaviour qualifies as a component of the domain, and for such included components analyses may reveal model properties not immediately recognized as properties of the domain. Note the distinction being made here: the domain as it exists "out there", and the model as an abstraction thereof and which "exists" on the (electronic) "paper" upon which the model is represented. The goal of domain analysis is to also establish a theory (theorems, properties) of the domain, or rather, of the models purported to represent the domain.

5.1.2 Domain Engineering vs. Logical AI

In logical artificial intelligence (AI) [154] "an agent can represent knowledge of the world, its goals and the current situation by sentences in logic and decide what to do by inferring that a certain action or course of action is appropriate to achieve its goals". In domain engineering we gather and represent knowledge about the world. In requirements engineering we establish goals. And in software design we facilitate inference, perhaps using AI techniques, perhaps using algorithmic, more classical means.

Many of the concerns of logical AI: epistemology (study of knowledge, its form and limitations),circumscription (a technique of non-monotonic reasoning) [150, 80], ontology (the value range of the alphabet of our observations about the world), phenomenology (on phenomena in the world: rails, timetables, trains, lines, stations, etc., their types, and types of classes of phenomena, etc.), bounded informatic situation and common sense (what people know about the world without reference to laws of physics, chemistry, etc.) [152], speech act theories, situation calculus (formalism for causal reasoning) [87], frame problems (how to express facts about the effects of actions without having to state what remains unchanged) and non-monotonic reasoning (non-monotonicity arises when facts are true in one world but become false when additional facts are considered) [155], are also concerns of domain engineering — and it can be expected that domain engineering and logical AI will share many research topics as well as engineering

techniques.

5.1.3 A Domain Engineering "Process Diagram"

We can summarise domain engineering by the upper half of the diagram shown in figure 4.

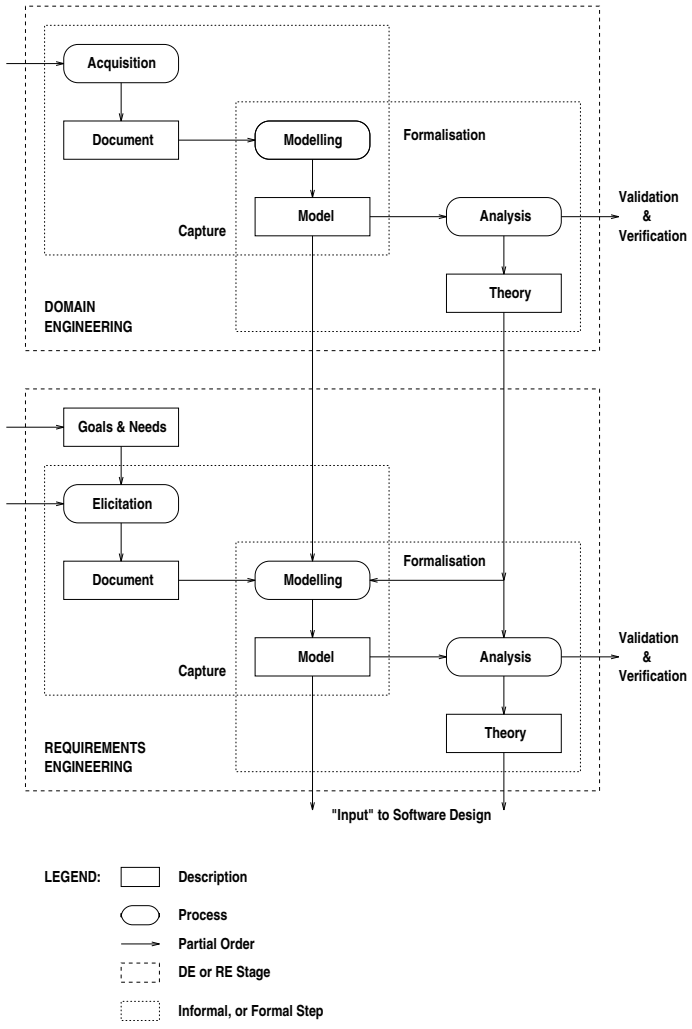


Figure 4: A Software Engineering Paradigm

The idea is that the domain engineer performs many tasks and that we wish to relate classes of these.

The upper box shows a diagram for the domain engineering "process". The lower for the requirements engineering "process". The left dot-framed boxes, one for domain engineering, another for requirements engineering, gather the mostly informal capture steps of domain engineering, resp. requirements engineering. The right dot-framed boxes, one for domain engineering, another for requirements engineering, gather the mostly formal modelling steps of domain engineering, resp. requirements engineering.

The arrow entering (from the left) to the dash framed domain engineering box (and incident upon the rounded acquisition box) designate the flow of information from stake-holders to developers.

The three arrows, the first (leaving from the rounded *analysis* box) to the right, and, the last two (leaving from the rectangular *theory* box) to the dash framed requirements engineering box designate the flow of information from developers to stake-holders, respectively to *requirements engineer*s.

Rounded boxes basically show development processes, rectangular ones show development documents, and internal arrows indicate the flow of information.

You may think of all rounded boxes as processes executing in parallel. That is, as human endeavours taking place potentially concurrently.

The triples *(arrow,rectangle,arrow)* can now be considered synchronous or synchronous communications between processes — in both directions: In those of the arrows communication brings information that can be used in the models, in the opposite direction the communication we may think of acquisition and elicitation (clarification) requests.

5.1.4 A Warning against Process Diagrams

The "process diagram" of figure 4 should not be taken too literally. Each problem that we face as software engineers usually poses new challenges.

The diagram as shown does not reflect any specific problem, and hence its usefulness is rather limited. "Real" software development problems tend to have their "process diagrams" be far more detailed wrt. to any of the boxes shown in the figure.

5.2 Requirements Engineering

Requirements, as we have seen, form a bridge between the larger domain and the "narrower" software which is to serve in the domain.

Requirements issues are either such which concern:

- The Domain:

  The mapping of domain concepts and facilities onto the machine: a projection of domain states, functions, operations and behaviours.

  Michael Jackson emphasizes: *Requirements reside in the domain.*

- The Interface:

  That is: the MMI (Man-Machine Interface) and the interfaces of the new software to other parts of the system,

- The Machine:

  Or they are requirements on the machine that do not constrain the (formal) interfaces of the machine, but refer to its physical properties (size and robustness to movement or temperature changes), internal dimensioning or velocity.

Requirements describe the system as the stake-holders *would like to see it.*

The process of capturing the requirements, writing them down in documents using a formal notation and analysing them is similar to the process of specifying the domain. Requirement models are formally derived from and extensions of domain models.

We are much inspired by Michael Jackson [121, 122, 218, 123].

5.3 Software Design Engineering

Given a design specification, the developer needs ingenuity and creativity to stepwise refine it into several design levels until executable code. Typically, two of the most basic steps in the design are *software architecture specification* and *program organization specification*.

A *software architecture description* specifies the concepts and facilities offered to the user of the software — i.e. the external interfaces.

A *program organization description* specifies internal interfaces between program modules (e.g., processes, platform components).

The concept of software architecture as treated in [7, 85, 4, 8, 9, 83, 82, 3, 84] more follows our definition of program organization than that of our definition of software architecture.

Since development of a software architecture from a requirements definition proceeds in stages of refinement that alternate between such which emphasize the external offerings (i.e. architecture) and the internal interfaces (i.e. organization), that difference is basically an academic one.

There are other, different 'schools' of software architecture: the Stanford Univ. *Rapide* (D.C. Luckham) [140, 128, 143, 142, 141], and the SRI Intl. (M. Moriconi) [165, 168, 167, 166]. [5]

There is also a "movement" on Domain Specific Software Architecture (DSSA) [6], and at SEI (the US DoD Software Engineering Institute at CMU) there is a larger software architecture effort [19]. We refer to:

- `http://www.sei.cmu.edu/technology/architecture/`

Other software architecture papers are [170, 89, 145, 70, 194]

5.4 Documents

All stages of software development result in *documents*. We document domains, requirements, software architectures, program organizations, etc. We sometimes also, again synonymously, refer to these descriptions as definitions (as for example for a domain model or a requirement model), sometimes as specifications (as f.ex. for a software-architecture model), yes even as designs (as for example for a program organization model).

The intended meaning of a document, that is, the semantical counterpart to this syntactic object, is what we call a *model* (see also 3.1). Thus, if the teams of developers and customers agree on the meaning of a document, this document presents a model (i.e., an abstraction) of the domain, of the requirements, of the software architecture, or of the program organization. Those models can be functions, relations, predicate transformers, labelled transition systems, Petri Nets, Event Structures, sequences of operations (functions) on data, relationships between entities (entity-relationship diagrams), abstract state machines, etc. It is not always obvious

[5] See also: `http://www.csl.sri.com/ moriconi/mmprojects.html`

[6] See `http://www.owego.com/dssa/faq/faq.html`

how to relate several of those models (even if you always use, say, labelled transition systems) because of the different levels of granularity and atomicity that the different models use at the different abstraction levels. Thus, there must also exist documents that relate those different abstraction levels (see, for instance, [1]). Now, if the properties of a set of models are logically coherent, we say that the corresponding documents are *concordant*. In other words, a set of documents is *concordant*, if the union of the documents is a consistent description of a model, each document emphasising a different aspect of the whole [218].

There seems to be a number of concordance issues:

- *Domain Perspectives & Aspects:*

  When structuring a domain specification one can either focus on reasonably separated issues within the domain such as pertaining to different groups of stake-holders (i.e. perspectives), and/or one can focus on the intrinsics, or the support technologies, or the rules & regulations, or the staff behaviour, etc. (i.e. aspects). A resulting domain description will probably be structured according to both principles.

- *Requirements Facets:*

  We have briefly mentioned this concept earlier. That is the (i) projection of domain states, functions, operations & behaviour form one facet; (ii) the focus on expectations about the machine itself, independent of the application to a first approximation, is another facet; and (iii) focus on the interface between the domain (users, equipment, etc.) and the machine is a third facet. This operational definition of requirements facets should be compared to [178, 72, 69, 73].

- *Software Views:*

  Here we follow the definition of 'views' put forward by Daniel Jackson [120]. Essentially a software view is a partial specification of components, functions, etc. Two or more views may then "view" models of desired components rather differently. The approach, as advanced by [120], then stipulates that the different views by correlated through invariants on states.

'View' concepts are today found in proposals for both requirement definitions and software designs.

As pointed out by Michael Jackson [122] the informal language of domain descriptions is indicative: "what there is", that of requirements descriptions is optative: "what there should be", and that of software design descriptions is imperative: "do this, do that — how to do it!". We could also use the terms *descriptive* and *prescriptive theories* in lieu of indicative and optative descriptions.

In contrast, the languages of formal descriptions are mathematical, and in mathematics we cannot distinguish between indicative, optative and imperative moods. Such distinctions are meta-linguistic, but necessary [218].

All stages and steps of the software development process involves creation: domain acquisition & domain modelling, requirements elicitation & requirement modelling, and design ingenuity. This human [210, 174] process of invention leads to the construction of informal as well as formal descriptions.

5.5 Validation vs. Verification

The domain acquisition and requirements elicitation processes alternate with domain modelling and requirements modelling, respectively, and these again with validation and verification securing satisfaction to the customer and to the developers.

This paper does not describe the crucial process of interactions between software developers (i.e. software engineers, which we see as domain engineers, requirement engineers and software designers) and the stake-holders. *Validation* is the act of securing, through discussion, with the stake-holders that the domain model correctly reflects their understanding of the domain, that the requirement model really corresponds to their expectations, that the assumptions on the behaviour of the environment are correct, that the rules to which the staff and operators are properly modelled, etc.

Verification was discussed in section 4.4.

5.6 A Classical Example: From Programming Languages to Compilers

5.6.1 The Domain: Language Semantics

In order to develop any compiler we must first fully document the source and the target languages. Hence we first establish or, if already existing, we study, the abstract and concrete syntaxes of the languages and their formal, abstract semantics. We cannot convince anyone that we have a sound semantics understanding of a programming language unless we can explain its formal semantics — which is assumed to be consistent and complete.

The *application domain* in this example is given by the two language syntaxes and semantics as well as the machine(s) where the compiler and the target code are going to run and, perhaps, (for purposes of optimizing), the type of optimizations that have a favourable impact on the efficiency of the target code. Part of the domain knowledge has been attained through "experimentation", i.e., practise, simulation or measurements.

5.6.2 The Requirements: Compiler Expectations

Given the two languages (the source programming and the target, usually machine language), we cannot just start developing a compiler before we have stated very clearly what is expected from the compiler.

Indisputably we need to express, as part of a requirements, that some notion of "executing a source program on an abstract machine with data" corresponds to "executing the compiled program on the concrete target machine with similar data".

In addition we may express a number of other expectations: either (i) that the compiler compiles fast, or (ii) that it delivers extensive both compile-time and run-time (interactive) diagnostics while allowing for "on-line" corrections (editing) of the originally submitted source program, or (iii) that the compiler generates highly optimized code, i.e., results in efficient use of run-time resources: time and/or space.

We expect that all these requirements are expressed in some formal manner — but recognize that we may not today have industrial strength techniques for expressing for example compile and run-time resource consumption!

Usually requirements include such non–functional expectations as the platform on which the compiler is to run or the form and conceptual contents of diagnostics, etc. Other difficult-to-

formalize requirements may be: The compiler must compile "indefinitely large source programs" although the platform on which the compiler executes may have a limited main store. And: The compiler must generate code for such programs to execute on a target machine with limited store.

5.6.3 The Software Design: Compilers

Now we may start the compiler design.

A first main task of compiler design is to establish a precise formalization of the compiling algorithm as well as a precise formalization of which analyses the compiler must make — that is: specifications of the back– and front–ends.

To do this compiler design proceeds from the domain formalizations of the source language semantics by making these more concrete. The domain models' static semantics then results, after some stages of design refinement, in a static analysis. And the domain models' dynamic semantics results, again after some stages of design refinement, in an abstract compiling algorithm. The former prescribes all the checks that the compiler must perform before it can accept the source text for code generation. The latter prescribes exactly which sequence of target language constructs each source language construct is to be compiled into.

A second main task of compiler design is to establish the structure of the compiler itself. This can only be done after the first task has been basically completed. The so-called static analysis specification, and the abstract compiling algorithm determine much of the structure of the compiler.

Typically one may seek a multi-pass compiler such that one can honour an "indefinitely large programs to be compiled" requirement. Analysis of the static analysis specification and the abstract compiling algorithm then reveals how many passes are needed. Here a pass is defined as a linear traversal of source program parse trees (in either direction and either from the leaves up to the parse tree root, or vice versa).

A final main task of compiler design is now to code the multi-pass administrator as well as all the passes which were individually identified (specified) during the second main task.

The refinement of the domain models modulo the requirements, the determination of an exact multi-pass structure, and the coding of it and all the front and back end passes constitute the software design.

It is likewise easy to see that software design is an altogether different activity whose complexity can only be mastered if both the domain and the requirements have been carefully and formally modelled.

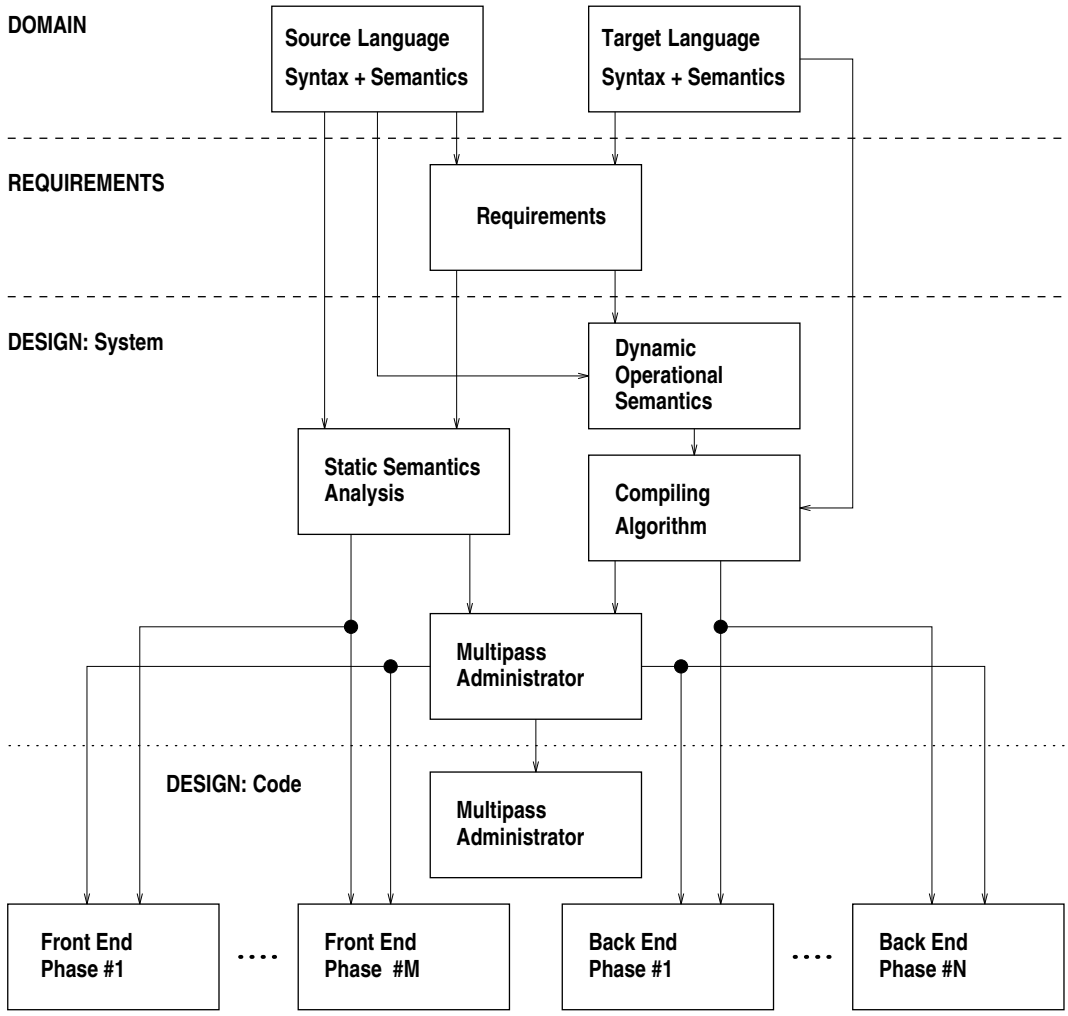
5.6.4 A Compiler Development "Process" Diagram

We can "picture" the above compiler development. See figure 5 page 36.

We refer to [23, 24, 25, 35] for more on software development graphs as mentioned above.

Each box is an "explosion" and "specialisation" of the 'Model' boxes of respective layers of figure 4. Especially the software design layer, not shown in that earlier figure, becomes detailed in this compiler development "process" diagram.

Figure 5: A Compiler Development "Process" Diagram



We refer to the referenced literature for further details. Suffice it here to say (i) that each box in the present diagram corresponds to an entire specification, (ii) that subsequent boxes are derived from preceding ones, and (iii) that there rests a proof of refinement (correctness) obligation. Issues (ii–iii) are designations of the arrows.

5.6.5 Discussion

The purpose of bringing the example of this section (5.6) was to hint that compiler development can be made much more easy and "safe", fast and economic, when developing the compiler as illustrated: (1) two language semantics, (2) formalization of functional requirements and precise expressions of non-functional requirements, and (3) task, stage and stepwise refinement of the compiler design itself as indicated.

We believe that the first industrial strength compilers that were developed using this approach were the CHILL and the Ada compilers developed at Dansk Datamatik Center in the early 1980's [103, 11, 102, 36, 179].

We know that today, 1997, more industrial compilers are still being developed informally, without any recourse to formal semantics, without any refinements, etc.

5.7 Software Support for Infrastructure Systems

An area much in need of research and poorly handled in practice is that of large scale software systems development for infrastructure systems. Examples of infrastructures are: (i) the financial service industry — of banks, insurance companies, securities brokers and traders, etc. — or just an entire bank (with all its branches), (ii) a railway system — in fact any transport system: airlines, shipping, a metropolitan transport system, etc., (iii) a manufacturing industry (of consumers and suppliers, producers and traders), (iv) air traffic, etc.

Current software for these infrastructure systems was developed piecemeal, basically without any form of formal model, let alone precise, reasonably exhaustive narrative description of the domain in question. Trying to get interoperability between separately (in space, time, etc.) developed program packages is a horrendous task that will effectively block serious progress.

Much research need be done [28, 29, 26, 27, 37]. Normative as well as example instantiated domain descriptions need be experimentally developed. Properly done it will take years.

No matter how many resources one throws after software architecture research etc., one must first understand the domains, then understand how requirements are derived from and relate to domains, before one may finally be able meaningfully to derive software architectures from and relate them to requirements definitions.

A good way to structure R&D into infrastructure software systems is to work within Michael Jackson's 'Problem Frame' paradigm — next!

6 Problem Frames

Based on the ideas of Polya [184], Michael Jackson's delightful book [122, 124] introduces the concept of problem frames. We now relate some of Jackson's frames to our quest: to move towards sensible, professional software engineering education programmes based on domain engineering, requirements engineering and software design engineering.

6.1 Translation Frame

Compiler and interpreter developments are classical examples of translation problem frames, as already described in sections 5.6 pages 34 to 35.

In summary we can say: we have the basis for a translation frame problem when its **domain** primarily revolve around two languages and the possibility of congruence between respective programs of these languages. We have a translation frame problem if that congruence is a major facet of **requirements**. **Design** now amounts to applying all the theories, techniques and tools of compiler (or interpreter) development: automata and lexical scanner generators, formal languages and possibly error correcting parser generators, code optimization and generation and attribute grammar processors, etc.

The translation problem frame is perhaps the best scientifically understood (researched), and technology-wise supported problem frame. There is however, it appears, still a long way to go before all the benefits of past research have become standard practice in the industry.

6.2 Information System Frame

The development of database (management) systems (DBs, resp. DBMSs) are typical examples of solutions (and technologies) for information systems problem frames.

We have the basis for an information systems problem frame if the **domain** presents itself in terms of well delineatable subsets of data "abstractions" of a "real world information" and of various data vetting procedures applicable to such information in addition to various more or less procedural treatments of such data. We have an information systems problem frame if the **requirements** stipulate that the machine (software + hardware) is to support the registration, vetting, storage and manipulation of such data. The **design** typically involves such considerations as mappings onto existing database management system schemes (DDLs, DMLs, etc.).

The database (etc.) problem is likewise a seemingly well understood problem frame [58, 59, 206, 207, 208, 211].

Current emphasis in this area is on federated databases i.e. the co-existence of different, heterogeneous databases — even GISs (geographical information systems) and DISs (demographic information systems). Spatial queries of remotely sensed images is currently en vogue.

6.3 Reactive Systems / Control Frame

The development of the digital computer control & monitoring of mechanical or chemical processes are prime examples of reactive systems problem frames — usually understood by software engineers as real-time, safety critical, etc. problems.

We have the basis for a reactive systems frame if the **domain** can be characterised in terms of outputs of a system which can be abstracted (i.e. modelled) by sets of typically differential equations over inputs, states and time. That is, if the problem can be identified as some — usually complicated — state which, over time undergoes changes in reaction to inputs and or (just) time, and whose output can be observed. We have a control frame problem if the **requirements** express that the reactive system state and output are to be subject to certain constraints.

Control theory and engineering have over the years produced respectable theories of control (viz. Bang-bang, Direct Digital, Adaptive, Stochastic, Fuzzy, etc., Control [15, 68, 77]).

The systems being controlled now, by digital computers, are, however, so complex and typically requires change of controllers during operation. Decisions as to controller changes are often best done by logical reasoning and in a way that has yet to be captured in control theoretic terms.

Software-based process-control (see the case study book [5]) is commonly a cycle-based reaction: the controller in each cycle first reads the inputs and then computes the reaction producing the corresponding outputs. *During* the cycle no input events from the process (or the console) are "allowed" to happen, or more properly, are not presented to the controller until the beginning of the new cycle. In this way, we may neglect the reaction time and consider the reaction as instantaneous. At first sight, this queueing of input events may seem to be a dangerous

delay (systems should react *fast*), and this *perfect synchrony hypothesis* as unrealistic (in the implementation it will take time). But, on the contrary, this simple trick introduces a higher level of abstraction which simplifies the complexity of the problem enormously. In synchronous circuit design this zero-delay viewpoint has been exploited for a long time, allowing the logic designer to view the circuit as representing equations on Boolean values.

Therefore, the most important new development in the **design** and programming of automatic control systems, embedded systems, system drivers, and signal-processing units is probably the introduction of the zero-delay hypothesis into software, the "synchronous programming" paradigm, whose most prominent exponent is ESTEREL [21, 22]. Other synchronous languages include Lustre [105], Signal [95], Argos [148], SyncCharts [10] (the last two being pure synchronous versions of Statecharts [108]). The perfectly synchronous model and languages appear independently in the beginning of the 80's in different places. With the synchrony abstraction the programmer may write 'declarative' code for each controller module, much like in logical or constrained programming. The specification of each module may be understood as a set of equations (or constraints) and the implementation (compilation) of the system as solving efficiently the conjunction of the sets of equations over all the modules). In control applications (as well as in hardware) synchrony is only a good abstraction because the reaction time of the system is bounded from above and a worst-case bound may be calculated efficiently off-line. This implies that those systems don't allow the use of recursive definition of functions (over unbounded or too large domains).

6.4 Workpiece Frame

Development of accounting systems, project management systems, CAE/CAD systems, and desk-top publication systems are examples of workpiece frame problems.

We have the basis for a workpiece problem frame if the **domain** represents itself in terms of typically a few categories of (construction) documents. For example (1) *templates* ("blank forms", or design standards, rules and conventions), (2) ("filled-in") *forms* (texts, or construction drawing), and (3) *aggregates*. Administrative system templates are ("empty", unused) forms with predetermined fields with explanatory texts, given types and blank space for "filling-in". Examples are: requisition forms, order forms, invoices, job applications, salary slips, etc. For technical constructions (CAE/CAD) templates are like logical rules for wiring power electricity, or water pipes, or telecommunication lines. Partially or fully filled-in administrative templates are the forms, and their summary into budgets and accounts are aggregates. Actual technical drawing following prescribed template constraints are then the forms, while their combination into (building) diagrams (where, for example, power lines are connected to water pumps and telecommunications gear) are aggregates. We have a workpiece **requirements** if the computer is to support the combination of all aspects of templates, forms and aggregates. The **requirements** may further turn the problem into also being an information systems frame problem. The workpiece frame problem may, in its **design**, depend on some constraint logic (inexact equational) solver.

The class of workpiece frame problems is very large and consists of rather different categories, viz.: the forms administration system vs. the CAE/CAD system vs. the desk-top publication system.

6.5 Other Frames

The above only suggests a number of frames. Jackson identifies further problem frames.

Among frames not mentioned above in detail are:

- *Resource Monitoring & Control — Estimation, Scheduling, Allocation:* This problem frame is typical in connection with both computer operating systems development as well as the development of resource management systems for such infrastructure systems as transport systems (railways, airlines, shipping, etc.).
- *Feature Interaction:*

  This problem frame is typical in connection with the development of telephony systems, computer operating systems and computer integrated manufacturing systems.

6.6 Remarks

Most real problems contain several frame aspects. The transaction frame evolves into one also containing parts of a workpiece frame when requirements mandate compile and/or runtime diagnostics & editing facilities. And so on.

The importance of the frame concept is that its proper handling makes the development process easier to manage: separate frame parts leads to separate concerns, and each frame part usually comes with an "own" development method [34].

7 Towards a Software Engineering Curriculum

7.1 Topics

We list around 45 course topics that seem to emerge from the analysis and proposals of this paper. Given a five year (10 semester) M.Sc.SE study a large subset of the course topics plus some elective courses could be absolved over eight semesters. This leaves time for a mid-study project semester and a final M.Sc. Thesis semester. Each semester could feature a 15 week course part with five courses and a two week examination part. Time for project work should also be assigned. The sequential listing of these course topics in no way indicates any ordering in an actual curriculum. The numbers in parentheses only indicate whether the course might conveniently be contained in a single, a double or a tri-semester course.

Each and every of the approximately 45 courses contain theory as well as practice parts.

You may thus view the below as a list of courses or as a list of topics. These can be ordered temporally (according to pre-requisites) and then be mapped onto a course structure with the weights given in parentheses. See section 7.2 page 42.

Basic Education: Mathematics and Natural Sciences

- Sets, Relations and Functions (1)
- Mathematical Logic & Meta-mathematics (1)
- Graph Theory & Combinatorics (1)
- Concrete, Abstract and Universal Algebras (1)
- Calculus: Differential Equations (2)

- ◦ Probability & Statistics (1)
- ◦ Numerical Analysis (1)
- ◦ Operations Research: Scheduling & Allocation (1)
- ◦ Optimization & Control Theory (1)
- ◦ Physics [71] (3)
- ◦ Chemistry (1)

Computer Engineering:
- ◦ Switching Theory & Circuits (1)
- ◦ Digital Electronics (1)
- ◦ Computer Architecture & Machine Organization [181, 200] (1)
- ◦ Data/Tele Communication (1)

Theoretical Computer Science:
- ◦ Computability and Complexity Theory [127] (1)
- ◦ Syntax: Automata Theory and Formal Languages [116] (1)
- • Semantics: Denotational, Axiomatic, Operational [191, 176, 110, 169, 213] (1)
- ◦ Type Theory [88, 2] (1)

Programming Languages:
- ◦ Functional Programming & Recursive Function Theory [160, 195, 212] (1)
- ◦ Imperative Programming & Hoare/Dijkstra Proof Systems [112, 13] (1)
- ◦ Logic Programming & Mathematical Logic [139, 125, 67] (1)
- ◦ Parallel Programming & Process Algebras [119, 114, 159, 110] (1)

Domain and Requirements Engineering:
- ◦ Computational Linguistics [45, 163] (1)
- • Abstraction & Formal Specification [75, 196, 197, 92, 93, 100, 101, 6, 135, 217] (1)
- • Domain and Requirements Engineering (1)

Software Design:
- • Software Architecture and Program Organisation (1)
- ◦ Algorithms & Data Structures [156] (2)
- • Design Calculi — Refinement [164, 49, 146, 147] (1)
- • Theorem Proving, Model Checking and Game Theories [182, 180, 54](1)

Frame Specific Software:
- ◦ Operating Systems (1)
- ◦ Translation: Grammars, Compiler Development (1)
- ◦ Information: Database Theory, Database Management Systems (1)
- ◦ Reactive Systems: Temporal Facets, Sampling & Feedback, Synchronous Approach (1)
- ◦ Communication: Connectors, Protocols (1)
- ◦ Transaction Processing: Distribution, Concurrency, 2-Phase Commit, Logging (1)

Software Technology:
- ◦ Pragmatics: Testing, Config. Management, Documentation, Quality Control (1)

◦ Information Technology Management: Project Management, Product Management (1)
◦ Platforms: Unix, C++, Java, Widgets (1)

7.2 An Example Software Engineering Cluster

If we take the bulleted (•) topics, these can be merged into for example a triple (1–2–3) or quadruple (1–2–3–4) of courses:

- Abstraction, Modeling & Specification
- Semantics
- Refinement Calculi, Domains, Requirements & Design
- Project Courses: Student Developments

Similar "merges" can be done for other set of topics across the eight main categories of topics.

8 Conclusion

8.1 General

We agree with G. Lelann [134] that poor system engineering practice is responsible for the fact that a large percentage of projects involving computing technology are significantly delayed, cancelled, entail much higher costs than anticipated or result in operational failures. We also agree with him that proof-based software engineering is the privileged vehicle to meet the ABC-challenge (Asap, Better, Cheaper) successfully. But not only industry is responsible for this state of affairs. Indeed, we would like to suggest that industry does not obtain properly educated engineers from university for that purpose. Formal methods (specifications, semantics, calculi, etc.) are not treated in the current curricula as the basic principles to get systems working correctly. It is our strong belief that a firm education such as suggested, with emphasis on abstraction and stepwise refinement, knowledge of formal semantics and methods for proving properties paired with industrial relevant project work would be a huge step towards a software which is reliable and maintainable. Who of us would not like to work in such environment?

8.2 Specifics

Paraphrasing Tony Hoare and He JiFeng [115], section 1.7: The ideal and the reality of engineering:

> The first idealisation (in our proposed graduate curriculum in software engineering) is that true domain knowledge, requirements and qualities of a product can be accurately captured in precise descriptions. Domain and requirements modelling is in fact the most impossible of all the engineers' tasks, because there is no way of checking that they describe what the customer actually is going to want when the product is actually delivered. Even the best requirements specifications are peppered with

> qualifications like "reasonable" and "normally" and "approximately" and "preferably", which cannot be made more precise until much later in the investigation of the design, or even after delivery. ...
>
> Another bold idealisation is that the specifications (also of domains), once formalized, will remain constant. They surely will change.[7] ...
>
> Finally, all other problems of engineering design must be subordinate to the over-riding imperative to deliver the promised product at the due time, and at a cost within the allocated budget. All the ideals of formal specification and design calculi: philosophy and logic, are of no avail if the engineer fails in this. ...
>
> ... In engineering some will ignore theoretical ideals, and rely exclusively on experience of their craft; but others will on occasion find guidance from their understanding and pursuit of an ideal, which is shared by other members of a recognized profession. The ideal suggests an integrated approach to the overall task, and enables deviations to be isolated and controlled separately.
>
> ... The privilege of the purest alliance to an ideal is that of the researcher, seeking to build a scientific foundation which will contribute simultaneously to the advancement of knowledge and education, as well as the continuous improvement of professional practice of the accredited engineer. One final appeal to an analogy with the physical sciences: it is in the pursuit of an ideal of truth that in the long run has led to the development of modern technology and engineering methods; and these have been of outstanding success in solving problems which continue to face the modern world.

We quote from Robert S. Boyer [41]:

Although Boyer is referring to undergraduate education in computer science, where we are dealing with graduate education in software engineering, we find that we share most views.

> The Rigor Resolution on Undergraduate Education — proposed for consideration by the Department of Computer Sciences University of Texas at Austin by Robert S. Boyer, Professor, September, 1995
>
> Computer science is a mathematical rather than a physical science. Following Church's thesis, we believe that the class of computations based on any currently imagined digital technology is completely characterized by the mathematical objects known as the partial recursive functions. That is, computer scientists need not make observations and experiments to determine the laws of the physical world relevant to their discipline; rather, computer scientists already know the fundamental law of computing, namely that we can compute exactly what can be computed by a universal Turing machine.
>
> Ideally, an undergraduate computer science curriculum should take as its principal goal that the students become skilled in reasoning rigorously about computing. And just as mathematics majors are taught rigorous mathematical thinking entirely by

[7] although it may surprise some that domain descriptions, rather than changing will converge (also a form of change). The more careful the domain work has been done, the less havoc requirements changes will cause. But this has to be verified through experience — and remains an idealistic claim.

the method of rigorously proving theorems about specific objects such as groups and continuous functions, computer science undergraduates should be taught rigorous mathematical thinking exclusively by the method of rigorously proving theorems about specific computational objects, such as specific partial recursive functions, i.e., algorithms.

...

As an antidote to what I perceive as a great overemphasis upon un-rigorous teaching in computing, I propose, by the following four resolutions, a major change of direction in our undergraduate curriculum.

The Rigor Resolution

RR-1. Resolved, that the Computer Sciences Department takes it as an objective, over the next ten years, to revise completely the undergraduate curriculum so that the following result is obtained, to wit, that every course in computing shall be taught upon a strict, mathematical basis. In every case that a computing system, language, architecture, algorithm, or technique is discussed, it will be presented to (or developed by) the students in a strictly rigorous fashion. Any program or system developed in such a class shall be developed in such a way that "correct" has a strictly mathematical, proven meaning. For example, the program or system may be proved to satisfy precisely given functional or performance requirements. This rigor requirement shall be extended to any prerequisite course that we require a student to take outside of computer science.

RR-2. Resolved, that those faculty members most skilled and experienced in reasoning rigorously about programs, and who also have some experience with undergraduate teaching, shall be mainly responsible for designing and teaching the introductory courses. A skill is best learned from the best. A skill badly learned is almost impossible to unlearn.

RR-3. Resolved, that in systematically reconsidering each and every course in the undergraduate curriculum, we shall insist upon identifying and publishing an answer to the question "What precise, incontrovertible scientific propositions are stated and proved in this course?"

RR-4. Resolved, that the primary societal objective we shall pursue in educating our undergraduate majors will be to prepare students for admission to and success in first rate graduate computer science programs, with the hope that these students will go on to advance the science of computing. (We take it that an analogous objective is currently followed in the departments of the two paradigm sciences, mathematics and physics.)

8.3 Acknowledgements

The authors acknowledge inspiration from IFIP Working Group 2.3 *Programming Methodology,* and from colleagues at UNU/IIST (Macau) and at their current respective places of work.

9 Bibliographical Notes

References

[1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, USA, 1996.

[3] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. *SIGSOFT Software Engineering Notes*, 18(5):9–20, December 1993. .

[4] G.D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, Oct 1995. .

[5] J.-R. Abrial, E. Boerger, and H. Langmaack. The Steam Boiler Case Study: Competition of Formal Program Specification and Development Methods. *Lecture Notes in Computer Science*, 1165:1–12, 1996.

[6] Jean-Raymond Abrial. *The B Book: Assigning Programs to meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.

[7] R. Allen and D. Garlan. A formal approach to software architectures. In *IFIP Transactions A (Computer Science and Technology); IFIP Wordl Congress; Madrid, Spain*, volume vol.A-12, pages 134–141, Amsterdam, Netherlands, 1992. IFIP, North Holland. .

[8] R. Allen and D. Garlan. Formalizing architectural connection. In *16th International Conference on Software Engineering (Cat. No.94CH3409-0); Sorrento, Italy*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press. .

[9] R. Allen and D. Garlan. A case study in architectural modeling: the AEGIS system. In *8th International Workshop on Software Specification and Design; Schloss Velen, Germany*, pages 6–15, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press. .

[10] C. André. Representation and Analysis of Reactive Behaviors: A Synchronous Aproach. In *Proceedings of CESA'96*, Lille, France, July 1996.

[11] Anon. *C.C.I.T.T. High Level Language (CHILL), Recommendation Z.200, Red Book Fascicle VI.12*. See [104]. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1980 – 1985.

[12] Anon. *C.C.I.T.T. High Level Language (CHILL), Recommendation Z.200, Red Book Fascicle VI.12*. See [104]. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1980 – 1985.

[13] K.R. Apt. Ten Years of Hoare's Logic: A Survey — Part I. *ACM Trans. on Prog. Lang. and Systems*, 3:431–483, 1981.

[14] K. Arnold and J. Gosily. *The Java Programming Language*. Addison Wesley, US, 1996.

[15] K.J. Åström and B. Wittenmark. *Adaptive Control*. Addison-Wesley Publishing Company, 1989.

[16] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. To appear, 1998.

[17] H.P. Barendregt. *The Lambda Caculus — Its Syntax and Semantics*. North-Holland Publ.Co., Amsterdam, 1981.

[18] M. Barr and G. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[19] Len Bass, Paul Clements, and Rick Kazman. *Software Architectur in Practice*. SEI Series. Addison-Wesley, 1997.

[20] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*. Addison-Wesley, ACM Press, 1989.

[21] G. Berry. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, chapter The Foundations of Esterel. MIT Press, 1998.

[22] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[23] D. Bjørner. Project Graphs and Meta-Programs: Towards a Theory of Software Development. In N. Habermann and U. Montanari, editors, *Proc. Capri '86 Conf. on Innovative Software Factories and Ada, Lecture Notes on Computer Science*. Springer-Verlag, May 1986.

[24] D. Bjørner. Software Development Graphs – A Unifying Concept for Software Development? In K.V. Nori, editor, *Vol. 241 of Lecture Notes in Computer Science: Foundations of Software Technology and Theoretical Computer Science*, pages 1–9. Springer-Verlag, Dec. 1986.

[25] D. Bjørner. The Stepwise Development of Software Development Graphs: Meta-Programming VDM Developments. In *See [33]*, volume 252 of *LNCS*, pages 77–96. Springer-Verlag, Heidelberg, Germany, March 1987.

[26] D. Bjørner. A Software Engineering Paradigm: From Domains via Requirements to Software. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, July 1997. .

[27] D. Bjørner. Towards a Domain Theory of The Financial Sevice Industry. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, July 1997. .

[28] D. Bjørner, C.W. George, B.Stig Hansen, H. Laustrup, and S. Prehn. A Railway System, Coordination'97, Case Study Workshop Example. Research Report 93, UNU/IIST, P.O.Box 3058, Macau, January 1997. .

[29] D. Bjørner, C.W. George, and S. Prehn. *Scheduling and rescheduling of trains*, page 24 pages. Prentice Hall (?), 1997.

[30] D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors. *VDM and Z – Formal Methods in Software Development*. Third International Symposium of VDM Europe, Kiel, FRG, April 17-21, 1990, Springer-Verlag, Lecture Notes in Computer Science, Vol. 428, April 1990.

[31] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer-Verlag, 1978.

[32] D. Bjørner and C.B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[33] D. Bjørner, C.B. Jones, M. Mac an Airchinnigh, and E.J. Neuhold, editors. *VDM – A Formal Method at Work*. Proc. VDM-Europe Symposium 1987, Brussels, Belgium, Springer-Verlag, Lecture Notes in Computer Science, Vol. 252, March 1987.

[34] D. Bjørner, Souleymane Koussoube, Roger Noussi, and Gueorgui Satchok. Jackson's Problem Frames: Syntax, Semantics and Pragmatics; Domains, Requirements and Design. Research Report 102, UNU/IIST, P.O.Box 3058, Macau, April + July 1997. Published as invited paper for ICFEM'97, Hiroshima, Nov.1997. IEEE Computer Society Press, Los Alamitos, Calif., USA.

[35] D. Bjørner and M. Nielsen. Meta Programs and Project Graphs. In *ETW: Esprit Technical Week*, pages 479–491. Elsevier, May 1985.

[36] D. Bjørner and O. Oest. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer-Verlag, 1980.

[37] Dines Bjørner. Requirements as an Arbiter between Domains and Software. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, 1997.

[38] R. Bloomfield, L. Marshall, and R. Jones, editors. *VDM – The Way Ahead*. Proc. 2nd VDM-Europe Symposium 1988, Dublin, Ireland, Springer-Verlag, Lecture Notes in Computer Science, Vol.328, September 1988.

[39] Egon Börger. *Annotated Bibliography on Evolving Algebras*, chapter Specification and Validation Methods, pages 37–51. Oxford University Press, 1995.

[40] Egon Börger. Why Use Evolving Algebras for Hardware and Software Engineering? *Lecture Notes in Computer Science*, 1012:236–271, 1995.

[41] Robert S. Boyer. The Rigor Resolution. Technical report, University of Teas at Austin, USA, 1995.
See `http://www.cs.utexas.edu/users/boyer/index.html`.

[42] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, second edition, 1997.

[43] L. Cardelli. Basic Polymorphic Type-checking. *Science of Computer Programming*, 8(2):147–172, 1987.

[44] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 19.

[45] R. Carnap. *The Logical Syntax of Language*. Harcourt Brace and Co., N.Y., 1937.

[46] R. Carnap. *Meaning and Necessity*. University of Chicago Press, 1956.

[47] Zhou Chaochen. Duration Calculi: An Overview. Research Report 10, UNU/IIST, P.O.Box 3058, Macau, June 1993. Published in: *Formal Methods in Programming and Their Applications*, Conference Proceedings, June 28 – July 2, 1993, Novosibirsk, Russia; (Eds.: D. Bjørner, M. Broy and I. Pottosin) LNCS 736, Springer-Verlag, 1993, pp 36–59.

[48] Zhou Chaochen and Michael R. Hansen. Lecture Notes on Logical Foundations for the Duration Calculus. Lecture Notes, 13, UNU/IIST, P.O.Box 3058, Macau, August 1993.

[49] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991.

[50] Zhou Chaochen, Dang Van Hung, and Li Xiaoshan. A Duration Calculus with Infinite Intervals. Research Report 40, UNU/IIST, P.O.Box 3058, Macau, February 1995. ublished in: *Fundamentals of Computation Theory*, Horst Reichel (ed.), pp 16-41, LNCS 965, Springer-Verlag, 1995.

[51] Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An Extended Duration Calculus for Real-time Systems. Research Report 9, UNU/IIST, P.O.Box 3058, Macau, January 1993. Published in: *Hybrid Systems*, LNCS 736, 1993.

[52] Zhou Chaochen and Li Xiaoshan. A Mean Value Duration Calculus. Research Report 5, UNU/IIST, P.O.Box 3058, Macau, March 1993. Published as Chapter 25 in *A Classical Mind*, Festschrift for C.A.R. Hoare, Prentice-Hall International, 1994, pp 432–451.

[53] W.J. Clancey. The Knowledge Level Reinterpreted: Modelling Socio–Technical Systems. *International Jounral of Intellegent Systems*, 8:33–49, 1993. .

[54] E.M. Clarke and J.M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Survey*, 28(4):626–643, December 1996.

[55] C. Courcoubetis, S. Graf, and J. Sifakis. An Algebra of Boolean Processes. In *Proc. of CAV'91*, pages 454–465, 1991.

[56] J. Cuellar and I. Wildgruber. The Steam Boiler Problem — A TLT Solution. *Lecture Notes in Computer Science*, 1165:165–183, 1996.

[57] Jorge Cuéllar, Dieter Barnard, and Martin Huber. A Solution relying on the Model Checking of Boolean Transition Systems. In *The RPC-Memory Specification Problem*, lncs, pages 213–??, 1996.

[58] C.J. Date. *An Introduction to Database Systems*. The Systems Programming Series. Addison Wesley, 1981.

[59] C.J. Date. *An Introduction to Database Systems*, volume II of *The Systems Programming Series*. Addison Wesley, 1983.

[60] J.W. de Bakker and ??? *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA, 1995.

[61] Harvey M. Deitel and Paul J. Deitel. *C++: How to Program*. Engineering, Science & Math. Prentice Hall, 2nd edition, December 1997.

[62] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite Systems. In van Leeuwen [209], pages 243–320.

[63] E.W. Dijkstra. Guarded Commands, Non-Determinacy and Formal Program Derivation. *Communications of the ACM*, 18(8):453–457, 1975.

[64] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[65] E.W. Dijkstra and W.H.L. Feijen. *A Method of Programming*. Addison-Wesley, 1988.

[66] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer–Verlag: Texts and Monographs in Computer Science, 1990.

[67] M. Dincbas and et al. The Contraint Logic Programming Language CHIP. In *FGCS'88: Intl. Conf. on Fifth Generation Computer Systems*. Japan, 1988.

[68] R.C. Dorf. *Modern Control Systems*. Addison-Wesley Publishing Company, 1967 (fifth ed. 1989).

[69] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check. *International Journal on Concurrent Engineering: Research & Applications*, 2(3), 1994.

[70] J.S. Poulin et al. A reuse-based software architecture for management information systems. In *Fourth International Conference on Software Reuse (Cat. No.96TB100015); Orlando, FL, USA*, pages 94–103, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press. .

[71] Richard Feynmann, Robert Leighton, and Matthew Sands. *The Feynmann Lectures on Physics*, volume Volumes I–II–II. Addison-Wesley, California Institute of Technology, 1963.

[72] A. Finkelstein, D. Gabbay, A.Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling In Multi-Perspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[73] A. Finkelstein and I. Sommerville. The Viewpoints FAQ. *Software Engineering Journal: Special Issue on Viewpoints for Software Engineering*, 1996.

[74] John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors. *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume LNCS 1313 of *Lecture Notes in Computer Science*, Heidelberg - Berlin, Germany, September 1997. Formal Methods Europe Symposium, Graz, Austria, Springer–Verlag.

[75] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques*. Cambridge University Press, 1997–1998.

[76] R.W. Floyd. Assigning Meanings to Programs. In *[192]*, pages 19–32, 1967.

[77] G.F. Franklin, J.D. Powell, and M.L. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley Publishing Company, 1980 (second ed. 1990).

[78] D. Gabbay, C. J. Hogger, and J. A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming: Logical Foundations*. Oxford Science Publications, Oxford University Press, Clarendon Press, 1993.

[79] D. Gabbay, C. J. Hogger, and J. A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming: Deduction Methodologies*. Oxford Science Publications, Oxford University Press, Clarendon Press, 1994.

[80] D. Gabbay, C. J. Hogger, and J. A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming: Nonmonotonic Reasoning and Uncertain Reasoning*. Oxford Science Publications, Oxford University Press, Clarendon Press, 1994.

[81] D. Gabbay, C. J. Hogger, and J. A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming: Epistemic and Temporal Reasoning*. Oxford Science Publications, Oxford University Press, Clarendon Press, 1995.

[82] D. Garlan. Research directions in software architecture. *ACM Computing Surveys*, 27(2):257–261, June 1995. .

[83] D. Garlan. Formal approaches to software architecture. In *Studies of Software Design. ICSE '93 Workshop. Selected Papers*, pages 64–76, Berlin, Germany, 1996. Springer-Verlag. .

[84] D. Garlan and M. Shaw. Experience with a course on architectures for software systems. In *Software Engineering Education. SEI Conference 1992; San Diego, CA, USA*, pages 23–43, Berlin, Germany, 199. Springer-Verlag. .

[85] D. Garlan and M. Shaw. *An introduction to software architecture*, pages 1–39. World Scientific, Singapore, 1993. .

[86] Marie-Claude Gaudel and Jim Woodcock, editors. *FME'96: Industrial Benefit and Advances in Formal Methods*, volume LNCS ??? of *Formal Methods Europe Symposium, Oxford, England*, Heidelberg - Berlin, Germany, March 1996. Formal Methods Europe, Springer–Verlag.

[87] M. Gelfond, V. Lifscitz, and A. Rabinov. What are the limitations of the situation calculus. In Robert Boyer, editor, *Automated Reasoning: Essays in Honour of Woody Bledsoe*, pages 167–179. 1991.

[88] Jean-Yves Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7. Cambridge Univ. Press, Cambridge, UK, cambridge tracts in theoretical computer science edition, 1989.

[89] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli. A prototype domain modeling environment for reusable software architectures. In W.B. Frakes, editor, *Third International Conference on Software Reuse: Advances in Software Reusability (Cat. No.94TH06940); Rio de Janeiro, Brazil*, pages 74–83, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press. .

[90] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[91] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer–Verlag, 1993.

[92] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[93] The RAISE Method Group. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[94] Nicola Guarino. Formal Ontology, Conceptual Analysis and Knowledge Representation. *International Journal of Human and Computer Studies*, 1996. Special issue on: (see title) Formal Ontology, Conceptual Analysis and Knowledge Representation; edited by N. Guarino and R. Poli.

[95] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming Real Time Applications with Signal. In *Another Look at Real Time Programming*, volume Special Issue of *Proceedings of the IEEE*, September 1991.

[96] C.A. Gunther. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1992.

[97] Yuri Gurevich. Evolving Algebras: An Attempt to Discover Semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[98] Yuri Gurevich. Evolving Algebras. *IFIP 13th World Computer Congress*, I: Technology and Foundations:423–427, 1994.

[99] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Brger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[100] J. Guttag, J.J. Horning, and J.M. Wing. Larch in Five Easy Pieces. Technical Report 5, DEC SRC, Dig. Equipm. Corp. Syst. Res. Ctr., Palo Alto, California, USA, 1985.

[101] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, , and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, Springer-Verlag New York, Inc., Attn: J. Jeng, 175 Fifth Avenue, New York, NY 10010-7858, USA, 1993.

[102] P. Haff and A.V. Olsen. Use of VDM within CCITT. In *[33]*, pages 324–330. Springer-Verlag, 1987.

[103] P.L. Haff, editor. *The Formal Definition of CHILL.* See [12]. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1981.

[104] P.L. Haff, editor. *The Formal Definition of CHILL.* See [12]. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1981.

[105] N. Halbwachs, P. Caspi, and Pilaud. The Synchronous Dataflow Programming Language Lustre. In *Another Look at Real Time Programming*, volume Special Issue of *Proceedings at the IEEE*, September 1991.

[106] M.R. Hansen and H. Rischel. Functional Programming in Standard ML. Lecture Notes, August 1997.

[107] Samuel Harbinson. *Modula 3.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1992.

[108] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[109] E.C.R. Hehner. *The Logic of Programming.* Prentice-Hall, 1984.

[110] M. Hennessy. *Algebraic Theory of Processes.* The MIT Press, Cambridge, Mass., USA, 1988.

[111] H. Herrlich and G.E. Strecker. *Category Theory.* Allyn and Bacon, Boston, 1973.

[112] C.A.R. Hoare. The Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12(10):567–583, Oct. 1969.

[113] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), Aug. 1978.

[114] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[115] C.A.R. Hoare and He Ji Feng. Unifying Theories of Programming. Publ.: Prentice Hall, 1997.

[116] J. Hopcroft and J. Ullman. *Introduction to Automa Theory, Languages and Computation.* Reading. Addison-Wesley, 1979.

[117] Dang Van Hung and Zhou Chaochen. Probabilistic Duration Calculus for Continuous Time. Research Report 25, UNU/IIST, P.O.Box 3058, Macau, May 1994. resented at *NSL'94 (Workshop on Non-standard Logics and Logical Aspects of Computer Science, Kanazawa, Japan, December 5–8, 1994)*, submitted to *Formal Aspects of Computing.*

[118] Dang Van Hung and Phan Hong Giang. A Sampling Semantics of Duration Calculus. Research Report 50, UNU/IIST, P.O.Box 3058, Macau, November 1995. ublished in: *Formal Techniques for Real-Time and Fault Tolerant Systems*, Bengt Jonsson and Joachim Parrow (Eds), LNCS 1135, Spriger-Verlag, pp. 188–207, 1996.

[119] Inmos Ltd. Specification of instruction set & Specification of floating point unit instructions. In *Transputer Instruction Set – A compiler writer's guide*, pages 127–161. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1988.

[120] Daniel Jackson. Structuring Z Specifications with Views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, October 1995.

[121] M. Jackson. Problems and requirements (software development). In *Second IEEE International Symposium on Requirements Engineering (Cat. No.95TH8040)*, pages 2–8. IEEE Comput. Soc. Press, 1995. .

[122] Michael Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.add-wes.co.uk, 1995. ISBN 0-201-87712-0; xiv + 228 pages.

[123] Michael Jackson. The meaning of requirements. *Annals of Software Engineering*, 3:5–21, 1997.

[124] Michael Jackson. *Software Hakubutsushi: Sekai to Kikai no Kijutsu (Software Requirements & Specifications: a lexicon of practice, principles and prejudices)*. Toppan Company, Ltd., 2-2-7 Yaesu, Chuo-ku, Tokyo 104, Japan, 1997. In Japanese. Translated by Tetsuo Tamai (Univ. of Tokyo, tamai@graco.c.u-tokyo.ac.jp) and Hiroshi Sako; ISBN 4-8101-8098-0; xxv + 267 pages.

[125] J.Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. Technical report, IBM Research, Yorktown, 1987.

[126] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[127] Neil D. Jones. *Computability and Complexity — From a Programming Point of View*. The MIT Press, Cambridge, Mass., USA, 1996.

[128] John J. Kenney. *Executable Formal Models of Distributed Systems based on Event Processing*. PhD thesis, Stanford University, Computer Systems Laboratory, 1996.

[129] Brian Kernighan and Dennis Ritchie. *C Programming Language*. Prentice Hall, 2nd edition, 1989.

[130] S.C. Kleene. *Introduction to Meta-Mathematics*. Van Nostrand, New York and Toronto, 1952.

[131] Orna Kupferman and Moshe Y. Vardi. Synthesis with Incomplete Information. In *Proceedings of the ICTL'97*, 1997.

[132] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[133] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[134] Gérard Le Lann. Proof-Based System Engineering for Computing Systems. In *ESA-INCOSE Conference on Systems Engineering*, Noordwijk, NL, November 1997.

[135] K. Lano. *The B Language and Method, A Guide to Pratical Formal Development*. Springer-Verlag, Formal Approaches to Computing and Information Technology (FACIT). Ed.: S.A. Schuman, 1996.

[136] Peter Gorm Larsen, editor. *Formal Methods*, volume LNCS ??? of *Formal Methods Europe Symposium, Odense, Denmark*, Heidelberg - Berlin, Germany, April 1993. Springer–Verlag.

[137] V. Lifschitz. (1) Pointwise Circumscription, (2) On the Semantics of Strips. In Ginsberg, editor, *Readings in Nonmonotonic Reasoning*.

[138] V. Lifschitz. An Introduction to Common Sense Reasoning. Lecture Notes, Stanford University and University of Texas at Austin.

[139] J.W. Lloyd. *Foundation of Logic Programming*. Springer-Verlag, 1984.

[140] D.C. Luckham. Rapide: A Language and a Toolset for Simulation of Distributed Systems by Partial Orderings. In *DIMACS Partial Order Methods Workshop*. Princeton University, July 1996, 25 pages.

[141] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture using Rapide. *IEEE Transaction on Software Engineering*, 21(4):336–355, April 1995.

[142] D.C. Luckham and J. Vera. An Event–based Architecture Definitions Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept. 1995.

[143] D.C. Luckham, J. Vera, and S. Meldal. Three Concepts of Software Architecture. Technical report, Stanford University, Computer Systems Laboratory.

[144] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[145] J. Magee and J. Kramer. Dynamic structure in software architectures. In D. Garlan, editor, *SIGSOFT '96. Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering; San Francisco, CA, USA*, pages 3–14, New York, NY, USA, 1996. ACM. .

[146] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.

[147] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.

[148] F. Maraninchi. The Argos Language: Graphical Representation of Automata and Descrption of Reactive Systems. In *International Conference on Visual Languages*, Kobe, Japan, 1991.

[149] J. McCarthy and et al. *LISP 1.5, Programmer's Manual*. The MIT Press, Cambridge, Mass., USA, 1962.

[150] John McCarthy. Circumscription—A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13:27–39, 1980. Reprinted in [153].

[151] John McCarthy. Applications of Circumscription to Formalizing Common Sense Knowledge. *Artificial Intelligence*, 28:89–116, 1986. Reprinted in [153].

[152] John McCarthy. *Formalization of Common Sense, papers by John McCarthy.* Ablex, San Diego, Calif., USA, 1990.

[153] John McCarthy. *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz.* Ablex, 1990.

[154] John McCarthy. Concepts of Logical AI. Note, Stanford Univ., Comp. Sci. Dept., Oct. 15 1997.

[155] John McCarthy. CS323: Nonmonotonic Reasoning. Stanford University Department of Computer Science Lecture Notes, Course 323, 14 lectures, 103 pages, 1997.

[156] K. Melhorn. *Data Structures and Algorithms: 3 vols.: 1: Multi-Dimensional Searching and Computational Geometry, 2: Graph Algorithms and NP-Completeness, 3: Sorting and Searching.* Springer-Verlag, EATCS Monographs, Heidelberg, 1984.

[157] Bertrand Meyer. Applying Design by Contract. *Computer (IEEE)*, 25(10):40–51, 1992.

[158] R. Milner. *Calculus of Communication Systems*, volume 94 of *LNCS.* Springer-Verlag, 1980.

[159] R. Milner. *Communication and Concurrency.* C.A.R. Hoare Series in Computing Science. Prentice Hall, 1989.

[160] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Mass., USA and London, England, 1990.

[161] Robin Milner. *Pi-Nets: A Graphical Form of pi-Calculus*, pages 26–42. Lecture Notes in Computer Science, Vol. 788, Springer 1994, ISBN 3-540-57880-3, 1994.

[162] Robin Milner, Joachim Parrow, and David Walker. Modal Logics for Mobile Processes. *Journal of Thepretical Computer Science*, 114(1):149–171, 1993.

[163] R. Montague. *Formal Philosophy: Selected Papers of Richard Montague. Eds.: Thomason and Richmond.* Yale Univ. Press, 1974.

[164] C. Carroll Morgan. *Programming from Specifications.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1990.

[165] M. Moriconi and R.A. Rimenschneide. Introduction to SADL 1,0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI Intl., CSL: Compter Science Lab., March 1997. 33 pages.

[166] M. Moriconi and Qian XiaoLei. Correctness and Composition of Software Architectures. In *ACM SIGSOFT'94 Symposium on Foundations of Software Engineering*, pages 164–174, New Orleans, Louisiana, December 1994. ACM SigSoft.

[167] M. Moriconi, Qian XiaoLei, and R.A. Rimenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[168] M. Moriconi, Qian XiaoLei, R.A. Rimenschneider, and Gong Li. Secure Software Architectures. In *Security and Privacy*. IEEE Computer Society Press, May 4–7, 1997.

[169] Peter D. Mosses. *Action Semantics*. Cambridge University Press: Tracts in Theoretical Computer Science, 1992. .

[170] T. Mowbray. The seven deadly sins of OO architecture. *Object Magazine; SIGS Publications*, 7(1):21, 24, March 1997. .

[171] Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors. *FME'94: Industrial Benefit of Formal Methods*, Formal Methods Europe Symposium, Barcelona, Spain, Heidelberg - Berlin, Germany, October 1994. Springer–Verlag.

[172] P. Naur. Proof of Algorithms by General Snapshots. *BIT, Nordisk Tidsskrift for Informations Behandling*, 6:310–316, 1966.

[173] P. Naur and B. Randall, editors. *Software engineering: The Garmisch Conference*. NATO Science Committee, Brussels, 1969.

[174] Peter Naur. *Computing: A Human Activity*. Dordrecht, 1995. Although I find this book problematic in many ways, I do strongly support many of Peter Naur's views on the human aspect of Computing.

[175] Greg Nelson, editor. *Systems Programming in Modula 3*. Innovative Technologies. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

[176] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons Ltd., Baffins Lane, Chishester West Sussec PO19 1UD, England.

[177] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Clarendon Press, Oxford University Press, Oxford, England, June 1990. .

[178] B. Nuseibeh, J. Kramer, and Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

[179] O. Oest. VDM From Research to Practice. In H.-J. Kugler, editor, *Information Processing '86*, pages 527–533. IFIP World Congress Proceedings, North-Holland Publ.Co., Amsterdam, 1986.

[180] S. Owre, J. Rushby, and N. Shankar. PVS: Prototype Verification System. In *11th Intl. Conf. on Automated Deduction (CADE-11)*, LNCS 607; Lecture Notes in Computer Science, pages 748–752, Saratoga, NY., USA, 1995. Springer-Verlag.

[181] Patterson and Hennesey. *Guess: Machine Organisation*. ???, ???

[182] L.C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Oddifreddi, editor, *Logic in Computer Science*, pages 361–386. Academic Press, 1990.

[183] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Comp. Sci. Dept., Aarhus Univ., Denmark; DAIMI-FN-19, 1981.

[184] G. Polya. *How to Solve It.* Princeton Univ. Press, 1957.

[185] S. Prehn and W.J. Toetenel, editors. *VDM ???* Fourth International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October, 1991, Springer-Verlag, Lecture Notes in Computer Science, Vol. 551, October 1991.

[186] P.J. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. *Proc. of the IEEE*, 77(1):81–98, January 1989.

[187] M. Reiser. *The* `OBERON` *System, User Guide and Programmer's Manual.* ACM Press. Addison-Wesley Publishing Company, 1991.

[188] R. Reiter. A Logic for Default Reasoning. In Ginsberg, editor, *Readings in Nonmonotonic Reasoning.*

[189] J.C. Reynolds. *The Craft of Programming.* Prentice-Hall, 1981.

[190] David Rydeheard and Rod M. Burstall. *Computational Category Theory.* Prentice-Hall Intl., 1991.

[191] D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon, 1986.

[192] J.T. Schwartz. *Mathematical Aspects of Computer Science, Proc. of Symp. in Appl. Math.* AMS, 1967.

[193] Robin Sharp. *Principles of Protocol Design.* International Series in Computer Science. Prentice Hall, 1994. ISBN 0-13-182155-5.

[194] C. Shekaran, D. Garlan, and et al. The role of software architecture in requirements engineering. In *First International Conference on Requirements Engineering (Cat. No.94TH0613-0); Colorado Springs, CO, USA*, pages 239–245, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press. .

[195] S. Sokołowski. *Applicative Higher-Order Programming: the Standard ML Perspective.* Chapman and Hall, 1991.

[196] J. Michael Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, UK, January 1988.

[197] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.

[198] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Mass., USA, 1977.

[199] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Engineering, Science & Math, 1992.

[200] Tenenbaum. *Guess: Architectures*. ???, ???

[201] J.G. Thistle and W.M. Wonham. Control of Infinite Behaviour of Finite Automata. *SIAM J. Control Optim.*, 32(4): , July 1994.

[202] J.G. Thistle and W.M. Wonham. Supervision of Infinite Behaviour of Discrete-Event Systems. *SIAM J. Control Optim.*, 32(4): , July 1994.

[203] W. Thomas. Automata on Infinite Objects. In van Leeuwen [209], pages –.

[204] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1996.

[205] D.A. Turner. `Miranda`: A Non-strict Functional Language with Polymorphic Types. In J.P. Jouannaud, editor, *Functional Programming Languages and Computer Architectures*, number 201 in LNCS. Springer-Verlag, Heidelberg, Germany, 1985.

[206] Jeffry D. Ullman. *Principles of database Systems*. Pitman, 1980.

[207] Jeffry D. Ullman. *Principles of Data and Knowledgebased Systems*, volume I. Computer Sciences Press, 1988.

[208] Jeffry D. Ullman. *Principles of Data and Knowledgebased Systems*, volume II. Computer Sciences Press, 1989.

[209] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier, 1990.

[210] Gerald M. Weinberg. *The Psychology of Computer Pogramming*. Van Nostrand Reinhold, 1971.

[211] Geo Wiederhold. *Database design*. McGraw-Hill, New York,N.Y., 2nd ed. edition, 1983.

[212] Å. Wikström. *Functional Programming using Standard ML*. Prentice-Hall, 1984.

[213] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA, 1993.

[214] M. Wirsing and M. Nivat (Eds.). *Algebraic Methodology and Software Technology*. Springer-Verlag, Lecture Notes in Computer Science, Vol. 1101, 1996. 5th International Conference, AMAST '96 Munich, Germany.

[215] N. Wirth. From Modula to Oberon. *Software — Practice and Experience*, 18:661–670, 1988.

[216] N. Wirth. The Programming Language Oberon. *Software — Practice and Experience*, 18:671–690, 1988.

[217] John Wordsworth. *Software Engineering with B.* Addison-Wesley Longman, 1996.

[218] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997. .