Some Thoughts on Teaching Software Engineering*
Central Rôles of Semantics

Dines Bjørner
Computer Science and Engineering
Informatics and Mathematical Modelling
Building 322, Richard Petersens Plads†
Technical University of Denmark
DK–2800 Lyngby, Denmark
E–Mail: db@imm.dtu.dk

19th of May 2002

Abstract

Somehow fitting the occasion for which this essay is written, I reminisce on a quarter century's teaching of software engineering. I delineate the sciences of computers and computing, and motivate a triptych of software development: From domain descriptions via requirements prescriptions to software architecture and component design. Via characterisations of domain attributes, stake–holder perspectives and facets; domain, interface and machine requirements and attendant domain projections, domain instantiations, domain extensions and domain initialisations, I venture into a series of characterisations of various forms of principles and techniques of abstraction and modelling: Basic ("assembler language–like") and conceptual ("modular, procedural") abstractions and models. From here I go back into issues of semiotics: Pragmatics, semantics and syntax, and to the art, craft and discipline of descriptions.

Throughout — framing in boxes — I risk my reputation by uttering dogmas and prejudices.

As my teaching is based mostly on own research, I will primarily only refer to own recent reports, lecture notes and publications — as well as to those of Jaco de Bakker's which had a deep influence on my work in the 1960s and 1970s.

Contents

*This essay is to appear in a Liber Amicorum in honour of Professor, Dr Jaco W. de Bakker on the occassion of his retirement, 31st of August, 2002.

†Prof. Richard Petersen was the instigator of the development and use of the first Danish stored programme electronic computer

1 Some Software Engineering Dogmas

1.1 From Science via Engineering to Technology

The engineer "walks the bridge" between science and technology: Creates technology based on scientific insight; and, vice–versa, analyses technological artifacts with a view towards understanding their possible scientific contents. Both science and technology; both synthesis and analysis.

In teaching software engineering we must teach both programming methodological, ie. computing science skills, as well as more mundane computer science skills — in order to walk both ways, forwards and "backwards", to, respectively from technology.

Work in the early 1970s, at the IBM Vienna Laboratory [1] on establishing a semantics for the IBM programming language PL/I, as well as that of my groups in the late 1970s and early to mid 1980s in establishing semantics for the CHILL and Ada programming languages [2, 3], clearly, besides a scientific content, had the semantics definition engineers walk the bridge from the technologies of PL/I, CHILL and Ada "back" to science, trying to discover whatever scientific values those languages might have had.

Current work on trying to establish a semantics for UML is not of the same nature. Whereas PL/I, CHILL and Ada could indeed be said, or claimed, to be soundly based on previous good scientific insight into programming language design, it seems that UML missed the boat: 20 years of painstaking programming methodological insight appears not to have been embodied in UML.

In teaching software engineering we are confronted with the recurrent dilemma of being asked to train in current, fashionable technologies, for which there is little scientific merit. My answer is one of almost Dutch Reformed Church pietism and strictness: Don't. Instead I focus on the wonderful, practical theories that ought to have been in those technologies.

1.2 CS ⊕ CS ⊕ SE

Computer science, to me, is the study and knowledge of the artifacts that can "exist" inside computers: Their mathematical properties: Models of computation, and the underlying mathematics itself. Computing science, to me, is the study and knowledge of how to construct those artifacts: (i) Programming languages, their pragmatics, their semantics, including proof systems, their syntax, and the principles and techniques of use; (ii) computing systems such as compilers, operating systems, database management systems, data communication systems, etc.; and (iii) applications — mostly.

The difference, between the computer and the computing sciences, is, somehow, dramatic: One is more contemplative, analytic, appeals to clever school boys. The other more adventurous, daring — in my prejudiced mind.

Software engineering is the art, discipline, craft, science and logic of conceiving, constructing, and maintaining software.

The sciences are those of applied mathematics and computing. I consider myself both a computing scientist and a software engineer.

Many so–called Computer Science departments, for lack of understanding, or because their lecturing cum researcher staff can't agree, or other, "waver" a course of teaching software engineering that sometimes contain too much theoretical computer science courses in relation to too few real programming methodological courses, or, vice–versa: contains too little theoretical computer science courses in relation to too many rather ordinary programming and software technology courses.

My, 'ideal' software engineering candidates have been taught one semester computer science courses in at least (i) automata theory, formal languages and computability, (ii) algorithms and complexity theory, and (iii) the theories underlying algebraic, axiomatic and denotational semantics. They have also been taught one or two semester computing science cum programming methodology courses each in (i) functional, (ii) logic, (iii) imperative, and (iv) parallel programming, in (v) algorithms and data structures (basic, intermediate and advanced), in (vi) real–time, embedded and concurrent systems design, and a (vii–ix) two–three semester course in the kind of software engineering outlined in this essay. After all this come courses in (x) compiler design, (xi) operating systems design, (xii) database management and database system design, (xiii) distributed systems & protocol design, etc. All of these computing science courses are based on the use of formal techniques: formal specification, analysis, verification, etc.

1.3 Informatics

1.3.1 The Quadrant of Informatics

Informatics, such as I see it, is a combination of: Mathematics, computer & computing science, software engineering, and applications. Some "sobering" observation: Informatics relates to information technology (IT) as biology does to bio—technology; $\mathcal{E}$tcetera ! I am somewhat "saddened" at the confusion of our field, of informatics, with that of information technology. The former is based on mathematics: Logic and algebra. The latter on the natural sciences. The former is a universe of intellectual quality: Elegance, beauty, correctness, fit. The latter is a universe of material quantity: Faster, smaller size, lower cost, larger capacity.

Many departments, who call themselves 'Informatics' departments, do not have a precise understanding of what they mean by 'informatics', and many computer science, or computing science (etc.) departments do not make sufficiently clear the relationships between their sciences and those primarily behind information technology.

1.3.2 Informatics of Infrastructures

What makes informatics interesting, to me at least, is its relations with the concept of infrastructures.

The World Bank Concept of Infrastructure: One may speak of a country's or a region's infrastructure.[1] But what does one mean by that ?

According to the World Bank,[2] 'infrastructure' is an umbrella term for many activities referred to as 'social overhead capital' by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spill-overs from users to non-users).

Our interpretation of the 'infrastructure' concept, see below, albeit different, is, however, commensurate.

[1] Winston Churchill is quoted to have said, during a debate in the House of Commons, in 1946: ... *The young Labourite speaker that we have just listened to, clearly wishes to impress upon his constituency the fact that he has gone to Eton and Oxford since he now uses such fashionable terms a 'infra–structure'* ...

[2] Dr. Jan Goossenarts, an early UNU/IIST Fellow, is to be credited with having found this characterisation.

Concretisations: Examples of infrastructure components are typically: The transportation infrastructure sub–components (road, rail, air and water [shipping]); the financial services industry (banks, insurance companies, securities trading, etc.); health–care; utilities (electricity, natural gas, telecommunications, water supply, sewage disposal, etc.); and perhaps also education, etc. ?

1.3.3 A Software Concept of Infrastructure

At UNU/IIST we took, in the mid 1990s[3], a more technical, and, perhaps more general, view, and saw infrastructures as concerned with supporting other systems or activities.

Software for infrastructures is likely to be distributed and concerned in particular with supporting communication of information, people and/or materials. Hence issues of (for example) openness, timeliness, security, lack of corruption, and resilience are often important.[4]

1.3.4 A Unification Attempt

We shall accept the two characterisations in the following spirit: For a socio–economically well–functioning infrastructure (component) to be so, the characterisations of the intrinsics, the support technologies, the management & organisation, the rules & regulations, and the human behaviour, must, already in the domain, meet certain "good functionality" conditions.

That is: We bring the two characterisations together, letting the latter "feed" the former. Doing so expresses a conjecture: One answer, to the question" *"What is an infrastructure"*, is, seen from the viewpoint of systems engineering, that it is a system that can be characterised using the technical terms typical of computing systems.

The Question and its Background: The question and its first, partial answer, only makes sense, from the point of view of the computer & computing sciences if we pose that question on the background of some of the achievements of those sciences. We select a few analysis approaches. These are aspects of denotational, concurrency, type/value, and knowledge engineering approaches, as well as a computer science approach.

An important aspect of my answer, in addition to be flavoured by the above, derives from the *semiotics* distinctions between: *pragmatics*, *semantics*, and *syntax*.

1.3.5 Discussion

The major challenge in software engineering seems to lie in the successful, and in the believably so, development of trustworthy, very large scale computing systems for world–wide infrastructure components. Such systems embody well–nigh all facets of computing abstractions: denotational as well as computational, concurrent and distributed (hence spatial), real-time (temporal), and, as we shall comment on later, autonomous multi–agencies probably embodying mechanised speech acts.

This is what makes informatics fascinating. And this is why *Unifying Theories of Programming* [4] becomes an overriding theoretical concern.

In understanding infrastructures we shall seek the semantics road.

[3] I write "mid 1990's" since that is what I can vouch for.

[4] The above wording is due, I believe, to Chris George, UNU/IIST.

1.4 A *Triptych* Software Engineering Dogma

Before software can be designed, we must understand the requirements. Before requirements can be expressed we must understand the domain.

Software engineering consists of the engineering of domains, engineering of requirements, and the design of software. In summary, and ideally speaking: We first *describe* the domain: $\mathcal{D}$, from which we *prescribe* the domain requirements; from these and interface and machine requirements, ie. from $\mathcal{R}$, we *specify* the software design: $\mathcal{S}$. In a suitable reality we secure that all these are properly documented and related: $\mathcal{D}, \mathcal{S} \models \mathcal{R}$, when all is done !

In proofs of correctness of software ($\mathcal{S}$) wrt. requirements ($\mathcal{R}$) assumptions are often stated about the domain ($\mathcal{D}$). But, by domain descriptions $\mathcal{D}$ we mean "much more" than just expressing such assumptions.

Domain engineering resembles, but is not the same as knowledge engineering. In the former we seek only understanding of the domain — through establishing precise, mathematical models. In the latter, it seems, knowledge engineers seek immediately computable models.

I find, but this may be just my personal "hang up", that many programming methodology cum software engineering curricula do not sufficiently enunciate the difference between domain and requirements engineering.

2 Formal Techniques vs. "Formal Methods"

A significant characteristics in our approach is that of the use of formal techniques: formal specification, verification & model checking. The area as such is usually — colloquially — referred to as "formal methods". By a method we understand a set of *principles* of analysis and for selecting *techniques* and *tools* in order efficiently to achieve the construction of an efficient artifact.

As such methods cannot be formal: Being carried out by humans whose ingeniousness or lack of same cannot be "straight–jacketed" — cannot be formalised.

3 Some Issues of Domain Engineering

Since, as we claim, domain engineering is rather a novel idea, we shall spend some space enunciating ideas of domain modelling.

3.1 Domain Facets

To understand the application domain we must describe it. We must, I believe, describe it, informally (ie. narrate), and formally, as it is, the very *basics,* ie. the *intrinsics;* the *technologies* that *support* the domain; the *management & organisation* structures of the domain; the *rules & regulations* that should guide human behaviour in the domain; those *human behaviours:* the correct, diligent, loyal and competent works; the absent–minded, "casual", sloppy routines; and the near, or outright criminal, neglect. *&c.*

We have not found these domain facet concepts in the software engineering literature — so perhaps we should advertise, here, their usefulness in directing the software engineering in being systematic about certain aspects of domains.
For the software engineer, to make effective use of the domain facet concepts, principles and techniques for their abstraction and modelling must be provided.

In [5] we present some such principles and techniques.

3.2 Domain Attributes

Michael Jackson, in [6], has convincingly, enunciated a number of attributes of phenomena of domains: Static and dynamic attributes; tangible and intangible attributes — which we further classify into humanly or otherwise physically perceivable tangible attributes as well as only conceptually, hence — in a sense — intangible, conceptual attributes; and zero, one or multi–dimensionality attributes. To these we add: discrete, continuous and chaotic attributes (not necessarily along a time axis); and temporal, spatial and combined time/space attributes.

For the software engineer, to make effective use of the domain attribute concept, principles and techniques for their abstraction and modelling must be provided.

In [11] we present some such principles and techniques

3.3 Domain Perspectives

Software serves many masters: Some are themselves machines, others are humans. Some humans, ie. managers, order software for their enterprise. Others use it daily. yet others "suffer" from such uses. In any software development project it is of interest to try delineate the spectrum of stake–holders: From enterprise owner, via strategic, tactical and operational enterprise management, to "floor" ("blue collar") workers, enterprise clients, providers of such software, IT, etc., regulatory agencies, citizens "at large", and the ever present, talkative politicians who interfere in almost anything: All have a stake in the computing systems, one way or another. The art is now to sufficiently identify this spectrum, to identify their perspective upon the domain, and, hence, to model these perspectives, coherently and consistently.

Too much "softness", too much politically correct talk is, to my taste, connected with the topic of securing proper attention to stake–holder perspectives. It need not be so. There simply is an exciting theory and likewise worthwhile engineering techniques for modelling the stake–holder perspective notion.

In [5] we present some such principles and techniques. In general, Chapter 15 of our lecture notes, [7], cover many principles, techniques and tools of domain abstraction and modelling.

3.4 The Evidence

How are we describing the domain ? We are *rough sketching* it, and *analysing* these sketches to arrive at *concepts.* We establish a *terminology* for the domain. We *narrate* the domain: A concise professional language description of the domain using only (otherwise precisely defined) terms of the domain. And we *formalise* the *narrative.* We then *analyse* the *narrative* and the *formalisation* with the aims of: *validating,* "against" domain stake–holders, and *verifying* properties of, the *domain description.*

Software engineering text books perfunctorily cover the subject of documentation, see next, but rarely enunciate the distinction between rough sketching, terminologisation, narration, and formalisation. We find it of utmost importance that the software engineer be trained in principles and techniques of description: Of expressing oneself in natural, albeit the professional language of the application domain at hand.

3.5 On Documentation in General

In general there will be many documents for each phase[5], stage[6] and step[7] of development: Informative documents: Needs and concepts, development briefs, contracts, *&c.* Descriptive/prescriptive documents: Informal (rough sketches, terminologies, and narratives) and (formal models) analytic documents: Concept formation, validation, and verification. These sets of documents are related, and occur and re–occur for all phases.

3.6 Discussion

We find that current, popular software engineering text books let the students down on making the above distinctions — etcetera !

[8, 9, 10] provide various examples of the application of domain modelling principles and techniques. [11] provides a summary overview.

In describing domains we shall seek the semantics road.

4 Some Issues of Requirements Engineering

We see requirements prescriptions as composed from three viewpoints: Domain, interface and machine requirements. We survey these.

Requirements are about the machine: The hardware and software to be designed.

4.1 Domain Requirements

Requirements that can be expressed solely with reference to, ie. using terms of, the domain, are called *domain requirements*. They are, in a sense, "derived" from the *domain understanding*. Thus whatever vagueness, non–determinism and undesired behaviour in the domain, as expressed by the respective parts of the domain *intrinsics*, *support technologies*, *management & organisation*, *rules & regulations*, and *human behaviour*, can now be constrained, if need be, by becoming requirements to a desirably performing computing system.

We do not find, in the software engineering literature, this distinction between, on one hand doing a sufficiently proper at understanding, including, notably, formalising, models of the domain, and, on the other hand, "deriving", as it were, domain requirements from domain models. So perhaps we should advertise, here, their usefulness in directing the software engineering in being systematic about certain aspects of requirements.

[5] Domain, requirements and software design are three main phases of software development.

[6] Phases may be composed of stages, such as for example the domain requirements, the interface requirements and the machine requirements stages of the requirements phase, or, as another example, the software architecture and the software component stages of the software design phase.

[7] Stages may then consist of one or more steps of development, typically data type reification and operation transformation — also known as refinements.

The development of domain requirements can be supported by principles and techniques of *projection:* Not all of the domain need be supported by computing — hence we project only part of the domain description onto potential requirements; *determination:* Usually the domain description is described abstractly, loosely as well as non–deterministically — and we may wish to remove some of these properties; *extension:* Entities, operations over these, events possible in connection with these, and behaviours on some kinds of such entities may now be feasibly "realisable" — where before they were not, hence some forms of domain requirements extend the domain; and *initialisation:* Phenomena in the world need be represented inside the computer — and initialising computers is often a main computing task in itself, as is the ongoing monitoring of the "state" of the 'outside' world for the purpose of possible internal state (ie. database) updates. There are other specialised principles and techniques that support the development of requirements.

We do not find these domain projection, determination, extension and initialisation concepts in the software engineering literature — so perhaps we should advertise, here, their usefulness in directing the software engineering in being systematic about certain aspects of requirements.

In describing domain requirements we shall seek the semantics road.

4.2 Interface Requirements

Requirements that deal with the phenomena shared between external users (human or other machines) and the machine (hardware and software) to be designed, such requirements are called *interface requirements.* Examples of areas of concern for interface requirements are: Human computer interfaces (HCI, CHI), including graphical user interfaces (GUIs), dialogues, etc., and general input and output (examples are: Process control data sampling (input sensors) and controller activation (output actuator)). Some interface requirements can be formalised, others not so easily, and yet others are such for which we today do not know how to formalise them.

The old "adage": 'User friendliness' has become 'pat'. What we increasingly need, and what we, in fact, increasingly, can also express formally, is what is meant by 'user friendliness', namely that the interface reflects only, and exactly those concepts that are indigenous to the domain — albeit in some, usually diagrammatically rendered form.

4.3 Machine Requirements

Requirements that deal with the phenomena which reside in the machine are referred to as *machine requirements.* Examples of concerns of machine requirements are: performance (resource [storage, time, etc.] utilisation), maintainability (adaptive, perfective, preventive, corrective and legacy–oriented), platform constraints (hardware and base software system platform: development, operational and maintenance), business process re–engineering, training and use manuals, and documentation (development, installation, and maintenance manuals, etc.).

Whereas domain requirements seem formalisable, that is, whereas it seems possible to express domain requirements precisely, such seems not the case, currently with machine requirements: All right, we can mathematically express for example performance and dependability issues, but we seem not to know how to "refine" such expressions into provably related implementations — such as we increasingly know how to do it for domain requirements.

[12] covers notions of domain to requirements "derivation". So does a later [13]. [14] provides a summary overview.

5 Some Issues of Software Design

Once the requirements are reasonably well established software design can start. We see software design as a potentially multiple stage, and, within stages, multiple step process. Concerning stages one can identify two "abstract" stages.

5.1 Software Architectures

The software architecture design stage in which the domain requirements find an computable form, albeit still abstract. Some interface requirements are normally also, abstract design–wise "absolved".

5.2 Software Components

and the software component design stage in which the machine requirements find a computable form. Since machine requirements are usually rather operational in nature, the software component design is less abstract than the software architecture design. Any remaining interface requirements are also, abstract design–wise "absolved".

The views that software architectures emerge from domain requirements, and hence have their "root" in domain models, whereas software component designs emerge from machine requirements, and hence have their root in the possibilities of the machine — those views seem different from that of the prevailing literature. We have had some difficulty, in the past, in getting enthusiastically excited about the seemingly, individually isolated concepts of "software architectures", respectively "software components". We now know why. It is all very simple.

[15] covers notions of domain to requirements to software architecture and component design.

6 Abstraction and Models

Whether we describe domain phenomena, or prescribe requirements, or specify software, we express models. They are just models. They are not the real thing. In expressing models we abstract. Judicious use of abstraction seems more important to software engineers than to, for example, automotive engineers, or to chemical engineers.

[16] provides a condensed overview of what I believe to be pertinent abstraction and modelling techniques.

6.1 Abstraction

To conceive of pleasing and adequate abstractions seems to be an art. But much can be learned from reading beautiful abstractions. In the following we will mention a few ideas.

It seems, to me, that we, in the computer and computing sciences as well as in the software engineering education, still have a lot to communicate to our students: The art, the discipline, the craft, the logic, and the science of abstraction.

In capturing abstractions we shall seek the semantics road.

6.2 Models and Modelling

When describing (as for domains), prescribing (as for requirements) and specifying (as for software designs), we create models. It is therefore important, for the software engineer, to decide which aspects these models are to portray and how: Whether they are analogic, iconic, or analytic models; whether they are prescriptive or descriptive; whether they do so in extension or in intension; and for what purposes the models are established: to gain understanding; and/or to get inspiration and to inspire; and/or to present, educate and train; and/or to assert and predict;and/or to implement. Finally, the principle of modelling, manifested by the problem domain itself, the mathematical structure of the model, and the identification between the two, clearly spells out the importance of the software engineer being conscious about the rôle, *&c,* of models.

It seems that engineers, for example taught control theory or operations research, are made better aware of the above notions of models and modelling — judging, simply, from the software engineering versus the respective literatures of control theory and operations research.

In building models we shall seek the semantics road.

6.3 Basic Modelling Principles & Techniques

Distinctions are made between property and model oriented abstract modelling. Property oriented models are usually algebraically cum axiomatically expressed. Model oriented models are usually expressed in terms of such mathematical entities as Booleans, numbers, sets, Cartesians, lists, maps and functions. The type concept, in the early days of Scott's (and de Bakkers) contributions to mathematical models for the λ–Calculus known as domain theory, is perhaps the finest contribution computer science has made to mathematics. Specification languages such as `OBJ, Act One, CafeOBJ, Maude,` and `CASL` facilitate property oriented specifications. Specification languages such as `VDM-SL, Z, RAISE's RSL, B,` and `ASM` facilitate model oriented specifications. All chronologically listed.

In a proper software engineering education we must make sure, I think, that our candidates know at least one property oriented specification approach, and at least two, reasonably diverse, model oriented approaches.

Both property oriented and model oriented specification work benefit from focusing first on semantics.

6.4 Additional Modelling Principles & Techniques

Over and above the basic modelling principles we find a number of additional modelling paradigms: (i) Non–determinism, internally or externally "chosen", and looseness; (ii) specification programming: Applicative (functional), imperative, logic, parallel, and algebraic programming — composing functions, dealing with references, propositions, composing processes or composing algebras; (iii) hierarchical ("top–down") versus compositional ("bottom–up") development and/or presentation of models; (iv) denotational versus computational semantic models; (v) configurations in terms of a spectrum from the more static contexts (environments) to the more dynamic states (stores); (vi) temporal, spatial and time/space models; *&c.*

The reader, by now, starts to see a picture emerging: One of a sizable variety of abstraction and modelling principles, techniques and tools. From the basics of property and model oriented abstractions, via the one just listed above (i-vi, *&c.*), and the domain attributes, stake–holder perspectives, domain facets, to the domain requirements projects, determinations, extensions and initialisations, and so and and so forth. Software engineering is indeed a universe of intellectual conceptualisations. And we need study and teach it all !

All of them reflecting, at their core, semantics, in one way or another.

7 Complementary Issues

7.1 On the Importance of Semiotics

"It is not for nothing" that Jaco de Bakker has devoted considerable time to the semantics of programming languages. It is, without question, the singlemost important issue of software engineering. Not just as a user of programming languages. But much more because whatever artifact the software engineer is designing, core issues of pragmatics, semantics and syntax enter into the design: The input to any software system, however end–user oriented, constitutes a language. Proper attention to what it is, ie. values of which semantic types, that one wishes to express, is hence of utmost importance. After semantics, in importance, come syntax.

Pragmatics is being neglected as a scientific and engineering topic although it, without question, is the most important topic of the three components of semiotics. Probably because it, by its very nature, cannot be formalised.

The software engineer must be well–versed in semiotics — in particular semantics and syntax — modelling techniques: denotational, operational, axiomatics, etc. And the software engineer must well–trained in making appropriate distinctions as to when a problem is a pragmatic, or is a semantic, or is ("just") a syntactic problem.

7.2 Logics, Agents and Language–based Knowledge Engineering

There is the *knowledge engineering* view. In one, of several, variants of this view — and we shall only cover that variant, albeit ever so briefly — one focuses on *logics, agent behaviours* and *speech acts*.

The *logics* area has two facets to it: The classical logics which are part also of the denotational, concurrency, type system, and formal techniques facets described and assumed earlier, and the less classical logics of modal logics. Thus, by *logics* we here mean those of the *epistemic logics* of *knowledge & belief,* the *deontic logics* of *permission & obligation,* the *modal logics* of *possibility & necessity, &c.* Other logics are relevant — also when describing domains: *dynamic logics* of action, *defeasible, uncertainty* and *possibilistic logics, logics* of *belief revision, &c.* These are not just logics of AI and logic programming but also logics of general domain engineering.

We present what may be termed the AI approach to *"agency"*, but intend to "lift" the AI *"agency"* notions to apply, inter alia, to domain engineering as well as to software (requirements and design). Agents *interact* through *communication.* Agents come in groups: *Multi agent "systems".* Agents perform both *competitive* and *co–operative* tasks. *Open multi agent "systems"* have agents serve different interests, *autonomously* and *heterogeneously.* Just like humans ! Agent *interaction* (alphabetically listed)[8] involves *arguments:* Formation of reasons, drawing of conclusions, and applying these actively; *commitments, conversations, co–ordination, dialogue, negotiation, obligation, planning, &c.* In doing so agents deploy various modal logics, and, as we shall next see: Speech acts.

Speech acts are characterised by: *Locutions* — The physical utterances of speakers; *illocutions* — The intended meaning of speaker utterances; and *perlocutions* — The actions that result from locutions. Wrt. illocutions, speech acts are often classified in the following five *performatives: Assertive,* ie. statements of fact; *directive,* ie. commands, requests or advice; *commissive,* eg. promises; *expressive,* eg. feelings and attitudes; and *declarative* which entail the occurrence of an action in themselves. Obviously speech acts and agents relate strongly.

We see an increasing fusion of software engineering, as it is classically known, with knowledge engineering, as indicated above. We cannot, at all, accept current, separational distinctions between software engineering and AI.

It is surprising to see how very little work has been actually done, based on 30 years of semantics of programming language, to formulate clear and concise semantics descriptions of multi–agencies and speech acts. My student, Mr Hans Madsen Petersen has recently completed a nice MSc Thesis on this topic (20 June, 2002).

7.3 On Description Principles and Techniques

Michael Jackson, in his delightful [6], espouses a "theory" of descriptions based on designations, definitions and refutable assertions — with designations centering around notions of recognition rules and designation sets. Although we do modify Jackson's description theory a bit, we wish here to gratefully acknowledge our debt. Jackson refrains, perhaps wisely so, from providing other than simple propositional logic examples. We advocate, in Chapter 13 of [7], a number of description principles, techniques and tools.

We can not overemphasise the importance of our software engineering candidates becoming far more capable of succinctly mastering their own, national language. One almost wishes 'Rhetoric', as a university discipline back !

[8] The listing is extracted from my MSc student, Hans Madsen Petersen's MSc pre–project report: *Agent Communication Languages and Speech Acts — and their Semantics,* October 2001.

7.4 Towards a Philosophy of Informatics

An issue central to description of domains is *"What can be described ?"* This question borders to, or is an outright philosophical problem. Such philosophy of logic, of language and of mathematics topics as mereology [17], epistemology [18], and ontology, are seen as increasingly important concepts of an emerging philosophy of informatics discipline.

We can only advocate that every software engineer be given serious courses in Theories of Science, Philosophy of Mathematics, and of Logic, and of Language.

We advice our students to have such books as [19, 20, 21, 22] at their desk for ready consultation.

8 Conclusion

We have surveyed a set of notions of software engineering. Our departure pont, in this discussion, has been that the underlying science of informatics has a strong base in mathematics — as well as in the philosophies of logic, mathematics and language: Epistemology, ontology and mereology, to name a few ingredients. But that it, basically, does not have a basis in the natural sciences. There is still a long way to go for may computer & computing scientists before their university colleagues from the natural sciences and mathematics understand that informatics is a whole new discipline, like theirs are old disciplines.

We have provided the subtitle: *A Rôle for Semantics* to the main title of this essay. Maybe it has not been sufficiently emphasised above, so let it be emphasised here:

The rôle of semantics, next to that more elusive one of pragmatics, far overshadows that of syntax. The software engineer must be fluent in semantics modelling, covering as wide a spectrum of techniques as possible. Here Jaco de Bakkers work over the last almost 40 years, including his seminal books [23, 24] have played, play, and will continue to play an important research and teaching function.

In a previous paper [25] we outlined more, including technical, details on the issue of teaching software engineering. In [26] I outlined the contents of a "massive" set of lecture notes, perhaps a publishable book [7], for such a set of main courses in formal techniques based software engineering. These lecture notes represent some 25 years of thinking and practice. They are nearing an n^{th} iteration ($n \approx 4$) of completion ! Together they could easily cover 3–4 semesters of teaching !

9 Acknowledgements

In my earlier years, at the IBM Vienna Laboratory, I came across [27], foreboding exciting things to come. And they came: Over the next almost 20 years I studied and enjoyed the contents and the precise style of numerous papers: [28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41]. Beyond 1980 the references are too numerous to list: But work on `POOL` figures among them. I am sure this Liber Amicorum will bring a comprehensive list of all of Jaco's splendrous works. So: Thanks, Jaco, for a lifetime of steadfast insistence on what we now consider a crowning achievement of yours — as well as of our field: The myriad of principles

and techniques of describing and analysing the semantics of a great variety of programming languages.

The reader will be excused: I primarily refer only to my own current reports, lecture notes and recent publications — and to those of Jaco de Bakker's !

References

[1] H. Bekič, Dines Bjørner, W. Henhapl, C.B. Jones, and P. Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, Vienna, Austria, 20 September 1974.

[2] P.L. Haff, editor. *The Formal Definition of CHILL.* See [42]. ITU (Intl. Telecomm. Union), Geneva, Switzerland, 1981.

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

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

[5] Dines Bjørner. *"What is a Method ?" — A Study of Some Aspects of Software Engineering.* IFIP WG2.3. MacMillan, Oxford, UK, 2002. Programming Methodology: Recent Work by Members of IFIP Working Group 2.3. Eds.: Annabelle McIver and Carrol Morgan. To be published.

[6] Michael A. 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.

[7] Dines Bjørner. *Software Engineering: Theory & Practice.* "In publisher's hand !", expected out 2003. Presently these lecture notes of around 1,000 pages are in a fourth phase of rewriting. Earlier phases took place in the mid 1980s, the late 1980s and the mid to late 1990s. A published book version is to be a "cut" version of the lecture notes.

[8] Dines Bjørner. Towards the E–Market: To understand the E–Market we must first understand "The Market". In *Government E–Commerce Development.* Ningbo Science & Technology Commission, Ningbo, Zhejian Province, China, 23–24 April 2001.

[9] Dines Bjørner. Formal Software Techniques in Railway Systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, Technical University, Braunschweig, Germany, 13–15 June 2000. VDI/VDE-Gesellschaft Mess– und Automatisieringstechnik, VDI-Gesellschaft für Fahrzeug– und Verkehrstechnik. Invited plenum lecture.

[10] Dines Bjørner. What is an Infrastructure ? In *The UNU/IIST 10th Anniversary Symposium.* UNU/IIST, Springer, March 2002. Eds.: Armando Haeberer, Tom Maibaum and Carlo Ghezzi.

[11] Dines Bjørner. Domain Engineering — A Prerequisite for Requirements Engineering — Principles and Techniques. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 16, 14, 44, 45, 46, 47, 48, 49].

[12] Dines Bjørner. Domains as Prerequisites for Requirements and Software *&c.* In M. Broy and B. Rumpe, editors, *RTSE'97: Requirements Targeted Software and Systems Engineering*, volume 1526 of *Lecture Notes in Computer Science*, pages 1–41. Springer-Verlag, Berlin Heidelberg, 1998.

[13] Dines Bjørner. From Domains to Requirements — Some Protocol Challenges. In *FORTE: Formal Protocol Description and Verification Techniques*. IFIP WG6.1, Kluwer Press, 2001.

[14] Dines Bjørner. Requirements Engineering — Some Principles and Techniques — Bridging Domain Engineering and Software Design. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 16, 11, 44, 45, 46, 47, 48, 49].

[15] Dines Bjørner. Where do Software Architectures come from ? Systematic Development from Domains and Requirements. A Re–assessment of Software Engneering ? *South African Journal of Computer Science*, 1999. Editor: Chris Brink.

[16] Dines Bjørner. Principles and Techniques of Abstract Modelling — Some Basic Classifications. — Towards a Methodology of Software Engineering. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 11, 14, 44, 45, 46, 47, 48, 49].

[17] Peter M. Simons. *Foundations of Logic and Linguistics: Problems and their Solutions*, chapter Łeśniewski's Logic and its Relation to Classical and Free Logics. New York, 1985. Georg Dorn and P. Weingartner (Eds.).

[18] Jonathan Dancy and Ernest Sosa, editors. *The Blackwell Companion to Epistemology.* Blackwell Companions to Philosophy. Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK, 1994.

[19] Rober Audi. *The Cambridge Dictionary of Philosophy.* Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.

[20] Ted Honderich. *The Oxford Companion to Philosophy.* Oxford University Press, Walton St., Oxford OX2 6DP, England, 1995.

[21] David Crystal. *The Cambridge Encyclopedia of Language.* Cambridge University Press, 1987, 1988.

[22] Nicholas Bunnin and E.P. Tsui-James, editors. *The Blackwell Companion to Philosophy.* Blackwell Companions to Philosophy. Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK, 1996.

[23] J.W. de Bakker. *Mathematical Theory of Programming Correctness*. Prentice-Hall, 1980.

[24] J.W. de Bakker and Erik de Wink. *Control Flow Semantics*. 608 pages The MIT Press, Cambridge, Mass., USA, April 1, 1996. ISBN: 0262041545

[25] Dines Bjørner and Jorge Cuellar. Software Engineering Education: The Rôle of Formal Specifications and Design Calculi. Annals of Software Engineering, 6 (1998) 365–409.

[26] Dines Bjørner. On Teaching Software Engineering based on Formal Techniques, Thoughts about and Plans for, A Different Software Engineering Text Book. *Journal of Universal Computer Science*, Vol.7, no.8, 2001: Colloquium "Formal Aspects of Software Engineering". Text of a talk given at the *Abschieds–Symposium* in honour of Professor Peter Lucas on the occassion of his retirement from The Technical University of Graz, Austria.

[27] Dana S. Scott and Jaco de Bakker. Approx. title: Notes on a Mathematical Model for the the λ–Calculus Informal, handwritten notes: IBM Vienna Laboratory, Vienna, Austria, 1979.

[28] J.W. de Bakker. Axiomatics of simple assignment statements. *MR94, Math. Centrum, Amsterdam*, pages 1–37, 1968.

[29] J.W. de Bakker. Semantics of programming languages. In *Advances in Information Systems Sciences, 2*, chapter 3, pages 173–227. Plenum Press, 1969.

[30] J.W. de Bakker. *Recursive Procedures*, volume 24. Math. Centre Tracts, Amsterdam, 1971.

[31] J.W. de Bakker. Axiom systems for simple assignment statements. In *[50]*, pages 1–22, 1971.

[32] J.W. de Bakker and W.P. de Roever. A calculus for recursive program schemes. In M. Nivat, editor, *International Colloquium on Automata, Languages and Programming, European Association for Theoretical Computer Science*, pages 167–196. North-Holland Publ.Co., Amsterdam, 1973.

[33] J.W. de Bakker and L.G.L.T. Meertens. On the completeness of the inductive assertion method. *International Journal of Computer and Information Sciences*, 11:323–357, 1975.

[34] J.W. de Bakker. The fixed point approach in semantics: Theory and applications. In J.W. de Bakker, editor, *Foundations of Computer Science*, pages 3–53. Math. Centre Tracts 63, Mathematisch Centrum, 1975.

[35] J.W. de Bakker. Semantics and termination of nondeterministic recursive programs. In S. Michaelson and R. Milner, editors, *International Colloquium on Automata, Languages and Programming, European Association for Theoretical Computer Science*, pages 435–477. Edinburgh Univ. Press, 1976.

[36] J.W. de Bakker. Least fixed points revisited. *Theoretical Computer Science*, 2:155–181, 1976.

[37] K.R. Apt and J.W. de Bakker. Exercises in denotational semantics. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science, Proceedings*, pages 1–11. Lecture Notes in Computer Science, Vol. 45, Springer-Verlag, 1976.

[38] K.R. Apt and J.W. de Bakker. Semantics and proof theory of Pascal procedures. In A. Salomaa and M.Steinby, editors, *International Colloquium on Automata, Languages and Programming, European Association for Theoretical Computer Science*, pages 30–44. Lecture Notes in Computer Science, Vol 52, Springer-Verlag, 1977.

[39] J.W. de Bakker. Semantics and the foundations of program proving. In B. Gilchrist, editor, *IFIP World Congress Proceedings*, pages 279–284. North-Holland Publ.Co., Amsterdam, 1977.

[40] J.W. de Bakker. Recursive programs as predicate transformers. In *[51]*, pages 165–181, 1978.

[41] J.W. de Bakker. A sound and complete proof system for partial program correctness. In J. Bećvář, editor, *Mathematical Foundations of Computer Science, Proceedings*, pages 1–12, Springer-Verlag, 1979. Lecture Notes in Computer Science, Vol. 74.

[42] Anon. *C.C.I.T.T. High Level Language (CHILL), Recommendation Z.200, Red Book Fascicle VI.12.* See [2]. ITU (Intl. Telecmm. Union), Geneva, Switzerland, 1980 – 1985.

[43] Dines Bjørner. Models, Semiotics, Documents and Descriptions — Towards Software Engineering Literacy. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [16, 11, 14, 44, 45, 46, 47, 48, 49].

[44] Dines Bjørner. Healthcare Systems. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 16, 11, 14, 45, 46, 47, 48, 49].

[45] Dines Bjørner. E–Business. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 16, 11, 14, 44, 46, 47, 48, 49].

[46] Dines Bjørner. Logistics. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 16, 11, 14, 44, 45, 47, 48, 49].

[47] Dines Bjørner. Projects & Production: Planning, Plans & Execution. Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 16, 11, 14, 44, 45, 46, 48, 49].

[48] Dines Bjørner. Railways Systems: Towards a Domain Theory. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 16, 11, 14, 44, 45, 46, 47, 49].

[49] Dines Bjørner. Financial Service Institutions: Banks, Securities Trading, Insurance, *&c.* Towards a Domain Theory for Work Flow Systems. Technical report, Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs.Lyngby, Denmark, 2001. This paper is one of a series of papers currently being submitted for publication: [43, 16, 11, 14, 44, 45, 46, 48, 47].

[50] E. Engeler. *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*. Springer-Verlag, 1971.

[51] E. Neuhold. *Formal Description of Programming Concepts (I)*. North-Holland Publ.Co., Amsterdam, Proc. of IFIP TC-2 Work.Conf., St. Andrews Canada, Aug. 1977, 1978.