Journal of Universal Computer Science, vol. 7, no. 8 (2001), 641-667
submitted: 15/5/01, accepted: 21/8/01, appeared: 28/8/01

On Teaching Software Engineering based on Formal Techniques — Thoughts about and Plans for — A Different Software Engineering Text Book

Dines Bjørner
(Informatics and Mathematical Modelling
Technical University of Denmark
Building 322, Richard Petersens Plads DK–2800 Lyngby, Denmark
db@imm.dtu.dk)

Abstract: We present the didactic bases for a different kind of text book on Software Engineering — one that is based on semiotics, proper description principles, informal narrations and formal specifications, on phase, stage and stepwise development from developing understandings of the domain, via requirements to software design. Each of the concepts: Semiotics, description, documents, abstraction & modelling, domains, requirements and software design, are covered systematically while enunciating a number of method principles for selecting and applying techniques and tools for the efficient construction of efficient software. The proposed textbook presents many, what are believed to be novel development concepts: Domain engineering with its emphasis on domain attributes, stake–holder perspectives and domain facets (intrinsics, support technologies, management & organization, rules & regulation, human behaviour, *&c.*); requirements engineering with its decomposition into domain requirements (featuring such techniques as projection, instantiation, extension and initialization), interface requirements and machine requirements; etc.

Keywords: Software Engineering, Formal Methods
Category: D2, D3.1, F4.3

1 Introduction

1.1 State–of–Affairs

The present lot of text books in whose title the composite term 'Software Engineering' feature predominantly, to me, fails on four significant accounts:

1. They miss, almost universally, the *design issue:* They do not teach programming, design, nor specification — they rather take it for granted. That is: They do not show a development from the very beginning to the very end, or at least to such a state that "coding follows !" Here we use the term 'development'. Up above we used the term 'design'. The two relate as follows: Development spans everything, from — as we shall see — domains (domain models) via requirements (models) to software (design models). Design, although predominantly here, in this paper, used mostly in the context of

'software design', also includes the design, or rather — as we shall do it — the abstraction & modelling (as one concept) of domain models and of requirements models.

2. They basically all miss 30 years of *formal techniques:* There may be, tucked as a separate chapter, seemingly accidental, somewhere, one entitled: "Formal Methods", but it is always, without exception, abundantly clear that the authors are not conversant in formal techniques — rather they unscientifically venture such statements as "Formal Methods seem applicable only to the development to real–time, safety critical systems".
3. They basically all take a very syntactic and bureaucratic view of documentation — missing the whole issue, in particular, of description: That all we do is writing documents. That some of these documents are informative, others are descriptive and yet others are analytic. And that descriptions, whether informal or formal are crucial: That to find out what to describe, and how to express a description. Really is at the core of software engineering.
4. And, thus, for lack of a scientific foundation,[1] they "chase" the current fashions of "software engineering" — some are relevant, but how do we, the reader/student really know (?), and some, several, are irrelevant in an academic syllabus. They typically lack a logical, didactic foundations structure: Topics are treated in some not always didactically discernible order; and chapters are more independently readable essays.

1.2 Towards Another Kind of Text Book

In other words, a software engineering text book will be proposed,[2] one that is based on the last 30 years of programming methodology emphasizing the following viewpoints: description principles, and hence semiotics: pragmatics, semantics and syntax, documents: informative, descriptive and analytic; methods and methodology, models and modelling formal specification, abstraction & modelling, and a whole suite of related issues — while presenting the more mundane, but utterly necessary, topics of for example: quality, assurance and control; project and product management; legacy systems; business process–[re–]engineering; etc. All of this in a natural context of those formal techniques — and not as an "add—on".

Such a text book seems desired. I will offer you my thoughts and plans for this, based on more than 30 years of experience, of teaching more than 25 full semester

[1] We shall, just as unscientifically, but blissfully, refrain from referring to which such 100.000 copies books we are specifically referring to.

[2] The proposed book basically already exists in rather complete forms of chapters of lecture notes.

courses, of teaching more than a similar number of 2–3 week intensive course around the world to — perhaps — around 1000 students, of graduating more than 75 MSc and 15 PhD students in the topic, of initiating perhaps around 15 large scale international R&D software methodology and development projects, and initiating and supporting some six–seven Danish software houses, with more than seemingly around 500 staff, using such formal techniques.

1.3 Prerequisites

We do not understand curricula that "features" a special course in "Formal Methods". To us, the use of formal techniques is part of well–nigh any university course in programming, software engineering, and the like ! The pre–requisite courses are shown in [Figure 1].

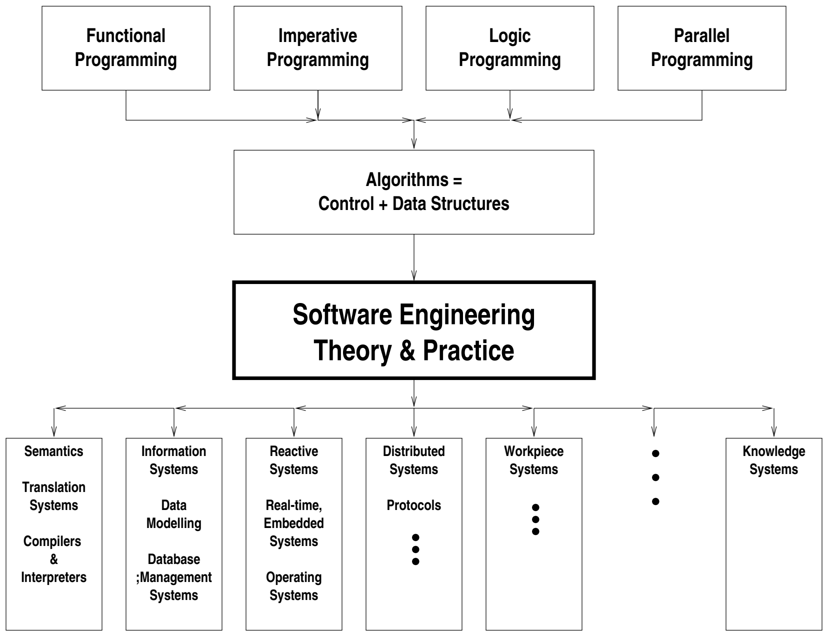

Figure 1: A Software Engineering Course Context

That figure also indicates courses that could follow a proper software engineering course, such as outlined in this paper. The figure, however, omits, for sake of brevity, many other necessary computer and computing science courses.

The proposed text book, although venturing into many of the (problem frame) areas indicated by the last row boxes of [Figure 1] — rather than covering these in–depth — will cover their basic principles, and will show how the principles covered in detail in the book apply to the more specialized problem frames. It is in this way that the book will "contain" itself, will avoid covering everything !

1.4 Summary of Issues

A brief list of main issues: **Method, semiotics, descriptions, documents, formal specification, abstraction and modelling, the software engineering triptych, domain engineering, requirements engineering, and software design.** "Orthogonal" to the above: **Problem frames.** In addition we treat some miscellanea: **Business process [re–]engineering, legacy systems,** and **management.**

The paper is cursory: Postulates, more than argues. Surveys, more than goes in—depth. A *Bibliographical Notes* section brings references that many offset this imbalance.

2 The Didactic Bases

2.1 The Basic "Theories"

The basic cornerstones of our approach to software engineering are: (α) That software development follows the triptych: From domains via requirements to software design. (β) That software development be conducted both informally and formally. (γ) That mastery of semiotics, of description principles and of proper documentation is part of professional software engineering. (δ) That mastery of abstraction and modelling principles and techniques is a prerequisite for subsequent fluency in domain, requirements and software design techniques. And (ω) that the detailed, highly structured principles and techniques of the latter[3] be *methodically* followed.

2.2 Science $\leftrightarrow$ Engineering $\leftrightarrow$ Technology

To us the engineer *"walks the bridge between science and technology":* Builds technological artifacts based on scientific insight, and examines technology so as to ascertain its possible scientific content. To us: Computer science is *the study and knowledge of the "things" that can exist inside computers.* And *Computing*

[3] Domain attributes, domain stake–holder perspectives and domain facets; domain requirements projection, instantiation, extension and initialization; as well as similarly large varieties of interface requirements, machine requirements, and software design principles and techniques.

science is *the study and knowledge of how to construct those "things".* These views pervade the proposed textbook, as well as the importance of the views of *Semiotics: Pragmatics, semantics* and *syntax; description principles;* and proper *documentation,* informal as well as formal.

2.3 A First View

Repeating the software engineering issues list given earlier, but now with a few more comments, we have:

Method: A set of **principles** for *selecting* and *applying* a number of *analysis* and *synthesis* **techniques** and **tools** in order *efficiently* to develop *efficient* artifacts (Software).

Semiotics: The confluence of *pragmatics semantics* and *syntax.*

Descriptions: *principles* and *techniques*: *designations, definitions,* and *refutable assertions.*

Documents: Be they *informative, descriptive,* or *analytic.*

Formal Specification[4]**:** Whether *property oriented,* or *model–oriented.*

Abstraction and Modelling: On one hand, *property oriented* specifications; and, on the other hand, *model–oriented* specifications.

The Software Engineering Triptych: Consists of *domain engineering, requirements engineering,* and *software design* .

Domain Engineering: That is: *Domain attributes, domain stake–holder perspectives,* and *domain facets.*

Requirements Engineering: That is: *Domain requirements, interface requirements,* and *machine requirements.*

Software Design: With design including: *Software architecture* design, *program organisation* design, and *implementation:* "Orthogonal" to the above we treat Michael Jackson's concept of:

Problem Frames: Whether: *Translation Frames, Reactive Systems Frames, Information Systems Frames, Workpiece Systems Frames, Connection Frames,* other frames, or combinations thereof. In addition we treat some miscellaenea:

Business Process [Re–]Engineering: Discovering *laws of the enterprise* from *domain analysis:* formulating "new" *enterprise structures, enterprise rules & regulations* and new *enterprise procedures:* and planning and effecting *change management.*

Legacy Systems: *Migrating* old systems into new systems based on *domain analysis* and *legacy systems models.*

Management: *Project Management,* primarily strategic, tactical and opera-

[4] — where formal specifications are just one part of the full specification "picture": The other parts include informal narrative and terminological descriptions.

tional resource management, and *Product Management,* primarily market assessment and planning.

3 A Second View

Finally we comment more extensively on the software engineering issues list:
Method: A set of **principles** for *selecting* and *applying* a number of *analysis* and *synthesis* **techniques** and **tools** in order *efficiently* to develop *efficient* artifacts (Software). Comments: The planned text book significantly features an adherence to and an exploration of this view of *"What a Method is !"*. Every other software engineering issue covered in the long lists now explored for the third time will be "equipped" with its *principles, techniques,* and *tools.* A predominant *tool* is that of *language,* since our "main businesses" are those of *semiotics, descriptions,* and *documents.* We seriously believe that this emphasis on a specific, proposed view of *"What a Method is "* is a main feature of the lecture notes.
Semiotics: The confluence of *pragmatics* — the reasons why we inform, describe and analyse — using specific textual forms; *semantics* — the meaning of these textual forms; and *syntax* — the structure of the textual forms. Comments: We consider ease of — fluency in — deploying the three concepts of semiotics important to good software engineering. Pragmatics is what makes us "tick", but pragmatics is elusive, cannot be formalised. Semantics and syntax can be formalised. Uninformed software engineers write pages of pragmatics, and fail to cover semantics properly.
Descriptions: *principles* and *techniques*: *designations,* manifest "pointers" to actual world phenomena ("things") in the form of descriptive texts; *definitions,* abstractions in the form of conceptual definitions; and *refutable assertions,* which if not potentially expressible renders our descriptions uninteresting. Comments: We consider ease of — fluency in — expression: succinctness, conciseness, and precision, as an indispensable prerequisite to good software engineering. The above can only be achieved when proper concepts have been duly identified. Following the description principles of designations, definitions and refutable assertions, goes hand–in–hand with *"ontologisation"* and *terminologisation.*
Documents: Be they *informative:* "loose" texts — like synopses — not conforming to any conventions; *descriptive:* rough texts, narratives, terminologies, or formal texts, which indeed describe something; or *analytic:* texts which report on concept formation from rough descriptions, or validation wrt. stake-holders of narratives and terminologies, or (formal) verification (proofs or model checkings) of formal descriptions. Comments : *Informative synopses* are like briefs: Starts ("roots") development. *Rough sketch descriptions* are like the proverbial "back—of—an—envelope" scribbles based on the analysis of iterations of which proper

concepts are discovered, identified and decided upon. *Narratives* and *terminologies* serve to address documentation needs of most stake–holders. *Formalisations* serve to secure trustworthy and efficient developments. *Analytic validation* and *verification* serve to secure believable and correct software.

Formal Specification: Whether *property oriented*, specifying properties of domains, of the required software, and of the software; or *model–oriented* — specifying domains, required software, and the software in terms of mathematical, in cases even computable values. Comments: Many *method principles* and *techniques* will be identified. Formal specification — and hence the choice of a proper, adequate variety of formal specification languages — form an indispensable set of tools for the practising software engineer. Ability to select an appropriate tool, here formal specification language, is thus crucial to professional software engineering.

Abstraction and Modelling: On one hand, *property oriented* specifications based on *sorts, generator, observer* and — sometimes, a few — *auxiliary function signatures*, and *axioms* over these (sorts and functions); and, on the other hand, *model–oriented* specifications based on *discrete mathematical values*, including *functions* (*sets, Cartesians, lists, maps*). Comments: Many *method principles* and *techniques* will be identified: *Representation* and *Operation Abstraction* either in terms of *sorts, function signatures* and *axioms* or in terms of *discrete mathematical, ie. abstract values* such as *sets, Cartesians, lists,* and *maps*, form a foundation for abstract modelling. Other abstract modelling principles and techniques are relevant: *Denotations* and *computations* — viewing syntactic "things" as denoting functions or describing computations; *hierarchies* and *compositions* — developing and/or presenting specifications "top—down" (most composite concepts first), or "bottom—up" (atomic concepts first), or in a judicious choice of both; *configurations:* spectra of *Contexts* and *states* — a "fine—grained state" concept: From static to dynamic, from time independent to temporal; *time, space* and *space/time* — introducing ontologies of time and space as appropriate abstractions and in appropriate parts of development; *discreteness, continuity* and *chaos* — when to consider the world, or the computing system as behaving *discretely, continuously* or *chaotically. &c.* Fluency in all of the above implied abstraction and modelling principles and techniques is necessary in order to be counted as a professional software engineer.

The Software Engineering Triptych: Consists of *domain engineering* — we cannot formulate requirements unless we have a reasonable understanding of the actual world in which potential, desired (ie. required) software is to be inserted; *requirements engineering* — and we cannot develop software unless we have formulated its requirements; and *software design* — which thus, ideally, but not necessarily, is developed after domain and requirements specifications have been established. Comments: Many *method principles* and *techniques* will be identified. An overall paradigm is that of *phase, stage* and *stepwise development* — of separation of concerns. One is not necessarily to strictly follow the above phase ordering in development. As long as the final product is documented in that order, with necessary and sufficient domain and requirements specifications. This development triptych is rather novel. It is not to be confused with *Knowledge Engineering.*

Domain Engineering: That is: *Domain attributes:* (i) *static* vs. *dynamic,* (ii) *tangible* vs. *intangible,* (iii) *one* or *multi–dimensional,* etc., attributes; *domain stake–holder perspectives:* identification of as relevant a spectrum of *stake–holders* as relevant (owners, management, workers, customers, regulatory agencies, politicians, etc.) and their *perspectives;* and *domain facets: Intrinsics* (the very essentials), *supporting technologies, management & organisation, rules & regulations, human behaviour,* etc. Comments: Many *method principles* and *techniques* will be identified. We stress our belief that our inclusion and rather extensive emphasis on *domain engineering* is not only novel, but possibly, when properly conducted, a main means for avoiding future *software development disasters.* Michael Jackson has covered *domain attributes* well in [Jackson (95), Jackson (97)]. All we do is to show relations to formalisations. The recent [McIver and Morgan (01)] studies the concepts of stake–holder perspectives and domain facets. We find the concept of *domain facets* novel and fascinating.

Requirements Engineering: That is: *Domain requirements: Projection, instantiation, extension* and *initialisation requirements* in direct support of *domain phenomena; interface requirements:* Identification of phenomena *shared* between the *domain* and the *machine,*[5] and the input/output facilitation of such phenomena, incl. *CHI: Computer Human Interface, bulk data input/output,* etc.; and *machine requirements: Performance, dependability, maintainability, portability, documentation (standards),* etc. Comments: *Performance* deals with both *time* and *space: response* and (in general) *execution times, storage space,* etc. *Dependability* issues include such as *reliability, availability, accessability, fault tolerance,* and *security. Maintainability* comes in three "flavours": *adaptive, perfective* and *corrective maintenance. Portability* deals with not only *execution code platforms,* but also *development* and *maintenance platforms.* But whereas almost all aspects of *domain requirements,* and many aspects of *interface requirements,* can both be informally narrated and *formally defined,* we find this to not yet be the case for *machine requirements !* Many *method principles* and *techniques* will be identified. Our treatment of *requirements engineering* is novel in that it takes it main departure point in *domain models;* and in showing "transformations" from *domains* to *domain requirements* and from *domain*

[5] By machine, with Jackson, we understand the hardware and software to be developed in response to requirements.

requirements to *software architecture* design, etc.
Software Design: With design including: *Software architecture* design, which addresses all *domain* and some *interface requirements; program organisation* design, which addresses remaining *interface* and all *machine requirements* — with the *program organisation* specification being an extension of the *software architecture* specification; and *implementation.* , which "transcribes" formally specified *architecture* and *program organisation* designs, via module (cum object–oriented) design to executable code. Comments: Our treatment of *software design* is novel in that it emphasises "transformations" from *formal requirements specification,* and in separating the concerns of *domain* vs. *machine requirements* — into *software architecture design,* respectively *program organisation design.* In these lecture notes we shall only cursorily cover program verification — leaving it to [RaiseMethod (95)] and other text books to cover this important area. Also we shall not express very many principles and techniques applicable within the software design phase: We leave that to specialised courses, some assumed prerequisites to a course based in these lecture notes. There simply is no way in which we can improve upon such text books as [Knuth (68), Dijkstra (76), Jones (80), Gries (81), Reynolds (81), Hehner (84), Jones (86a), Jones (90a)]. "Orthogonal" to the above we treat Michael Jackson's concept of:
Problem Frames: Whether: *Translation Frames,* as applicable to programming language interpreter and compiler development as well as to such software tools that in general perform abstract interpretation on structured texts; *Reactive Systems Frames,* typically real–time, embedded, safety critical systems; *Information Systems Frames,* typically ending up in conventional relational database, or in geographic or demographic information systems; *Workpiece Systems Frames,* typically forms handling systems (such as for accounting, bookkeeping, and resource management in general); *Connection Frames,* typically ending up in software that "connect—interfaces" with "other" machines "at either side" (such as web servers, instrument adaptors, etc.); other frames, typically actors and brokers in electronic commerce or in logistics; or combinations thereof. Comments: Many *method principles* and *techniques* will be identified. We refer to [Jackson (2001), Bjørner et al. (97b)]. The miscellanea includes:
Business Process [Re–]Engineering: Discovering *laws of the enterprise* from *domain analysis* — void of any reference to requirements to software, let alone such software; formulating "new" *enterprise structures, enterprise rules & regulations* and new *enterprise procedures* — with such "new" *enterprise behaviours* being supported, in cases, by computing *&c.;* and planning and effecting *change management* — a rather novel discipline, yet to be better understood. Comments: *Business process [re–]engineering* can now be done professionally: Using formal, abstract domain modelling principles and techniques.

A relevant need can thus be addressed in a scientific manner. The use of formal, *abstract domain modelling principles* and *techniques* for *Business process [re–]engineering* is novel. Much study is still needed.

Legacy Systems: *Migrating* old systems into new systems based on *domain analysis* and *legacy systems models* — requires "full–blown" domain analysis of the software systems that form the *"legacy"*, ie. a form of *"industrial archaeology"*. Comments: It seems, to me, that today's handling of *legacy systems:* old, convolute computing systems, including data bases, leaves much to be desired. *Legacy system migration* seems to be treated only syntactically. It seems that proper treatment implies the "full force" of *domain engineering.*

Management: *Project Management,* primarily strategic, tactical and operational resource management — based on software development graphs: From domains via requirements to software design; and *Product Management,* primarily market assessment and planning — based on wide—area domain analyses. Comments: Our treatment of management hinges crucially on the fact that we pursue software development according to all the principles and (also formal) techniques listed earlier. This changes management radically from what we normally see explained in conventional software engineering text books.

3.1 Some Comments

The ordering of the topics: *Semiotics, descriptions, documents, formal specification, abstraction* and *modelling,* the triptych; *Domains, requirements* and *software design, &c.* is no "accident". As we shall soon see, in the proposed text book, this ordering is preceded by a cursory treatment of *mathematics: Types, functions, algebras* and *logic.* The term 'model' has been used a couple of times, hence the lecture notes devotes a special chapter to the concepts of models and modelling.

4 The Structure of the Proposed Text Book

4.1 Overall Structure

The overall structure of the proposed text book follows the 'issues' listings given above. Most chapters have extensive examples, problem exercise formulations and solutions, and bibliographical notes. A special "feature" of the proposed text book is a number of individually rather large appendices which contain extensive domain, and in cases some requirements, models of selected infrastructure components. Instead of large examples, in–line with the principles and techniques they most aptly reflect, we have relegated such larger examples to those appendices. We strongly believe these example appendices and their cross–referencing from method issues, to be a most beneficial as well as a novel aspect.

In order to compose large specifications one must have read large such ! The proposed text book basically builds on the `RAISE` Specification Language, `RSL` [RSL (92), RaiseMethod (95)]. But here and there it makes excursions into various *modal logics* of *time, belief, knowledge, intention, commitment,* etc., process languages: `Petri Nets, StateCharts,` etc. The prerequisites for studying the proposed text book are basically: Programming courses in: functional programming á la `Standard ML,` logic programming á la `Prolog,` imperative programming á la `Java` (or `C`), and parallel programming á la `occam.` Also: courses in discrete mathematics and in mathematical logic, and one or two in algorithms & data structures. We leave it to the reader to study the contents listing !

4.2 Table of Contents

Aims & Objectives:

- To introduce the concepts of method and methodology, and to discuss their constituent principles, techniques and tools,
- to propose some meta–principles and techniques and
- to review principles and techniques related to Chapters 2–3.

Aims & Objectives:

- To introduce the notion of representational and operational abstraction,
- to introduce the notion of model oriented abstraction, and to contrast this notion with that of property oriented abstraction,
- to introduce the model oriented techniques of abstraction using discrete mathematical structures such as sets, Cartesian, lists, maps, and functions, including function lifting and
- to present more material on the concept of types.

Aims & Objectives:

- To explain the concepts of (i) models and of (ii) modelling,
- to enumerate and explain the concepts of (iii) iconic, (iv) analogic and (v) analytic models, of (vi) prescriptive and (vii) descriptive models, and of (viii) extensional and (ix) intensional models,
- to overview reasons for establishing models: To (x) gain understanding, to (xi) predict and (xii) assert, to (xiii) present, (xiv) educate and (xv) train, and for (xvi) implementation (whether for (a) business process [re–]engineering or for (b) computing systems development) and
- to establish principles of modelling.

Aims & Objectives:

- To introduce, more systematically, the concept of specification programming,
- to — in particular — cover aspects of applicative (ie. functional), imperative and concurrent (ie. parallel) programming,
- to thus introduce both imperative programming: Assignable variables and statements, and CSP: Communicating Sequential Processes and
- to introduce concepts of loose and under-specified specifications.

Aims & Objectives:
- To introduce the concept of modularity and means for expressing modularity in terms of the class, scheme and object concepts of RSL,
- to introduce the concept of parameterized and renameable specifications and
- to discuss the current "schools" of object orientedness in the light of our presentation of Subsections 8.2 –8.4.

Aims & Objectives:
- To introduce the concepts of hierarchies and compositions of development as well as of description document presentation,
- to introduce the concepts of denotational and operational (ie. computational) semantics,
- to introduce the concept of configurations in terms of the likewise introduced concepts of contexts and states and
- to introduce the ontological concepts of time, space and time/space.

Aims & Objectives:
- To introduce the concept of semiotics as consisting of the likewise introduced concepts of pragmatics, semantics and syntax and
- to emphasize the utter importance of considering and of modelling the world semiotically: (i) Adhering to pragmatics, (ii) focusing on achieving pleasing semantic types and functions, (iii) based on pleasing abstract syntaxes.

Aims & Objectives:
- To summarize essential linguistic features of common abstract [software] specification languages — such as RSL, VDM--SL, Z and others,
- to illustrate standard techniques for specifying the (mostly denotational) semantics of classical programming language concepts: Function (ie. applicative, SAL), imperative (SIL), modular (SMIL), logic (SLL) and parallel programming (SPIL) languages,
- to illustrate the phase-wise development of run-time computational semantics and compiling algorithms for selected such languages (SAL, SIL, SMIL, SLL) from denotational semantics specifications, and
- to illustrate the use of RSL in specifying a state transition (structural operational) semantics for a parallel programming language (SPIL).

12 A DEVELOPMENT PARADIGM

From Domains via Requirements to Software Design

Aims & Objectives:

- To become aware of the spectrum of 'software development' from 'domain engineering' via 'requirements' to 'software design',
- to also appreciate the pragmatic distinction between 'software architecture' and 'program organization' design and
- to finally appreciate that there are a number of techniques that relate requirements to domain descriptions, software architecture designs to domain and interface requirements, and program organization to machine requirements.

13 TOWARDS DESCRIPTIONS: THEORY & PRACTICE

Aims & Objectives:

- To introduce and further develop *Michael Jackson's* concepts of designations, definitions and refutable assertions,
- to introduce our notion of proper software development documentations, including their composition from informative, descriptional and analytical document parts and
- to introduce, especially *Per Galle's* concept of [design] acquisition and validation.

14 PHILOSOPHY OF DESCRIPTIONS

Aims & Objectives:

- To relate the software engineering quest for descriptions to philosophical issues of 'existence', of 'being', of what can be 'known', 'perceived' and 'described'.

15 DOMAIN ENGINEERING

Aims & Objectives:

- To introduce the concepts of (i) domain attributes, (ii) stakeholders & their perspectives, and of (iii) domain facets
  - intrinsics, support technologies, management & organization, rules & regulations, human behaviour, ...
- to achieve an understanding, by the reader, of the necessity of domain modelling and
- to make the reader acutely aware of the "looseness" of the domain: *That all is possible in the domain.*

Aims & Objectives:

- To introduce the concepts of domain, interface and machine requirements,
- to make the reader acutely aware of the novel domain requirements techniques of projection, instantiation, extension and initialization, and make the reader acutely aware of the possibilities of formally abstracting and modelling the interface requirements facets of computer human interfaces and dialogues,
- to prepare the reader for the many program organization design techniques for machine requirements such as performance, dependability, and maintainability and
- to cover some important issues of elicitation and validation of requirements as well as requirements development tools and management.

Aims & Objectives:

- To argue that software development cannot be seen in isolation from the more general computing (ie. hardware + software) systems development,
- to introduce the software design decomposition into software architecture, program organization, etc., that is: Argue that software architecture follows from domain requirements, that program organization follows from machine requirements, and that interface requirements influence both,
- to introduce the concepts of software system demos (and simulators) and of prototyping and
- to lead up the next chapter's treatment of software correctness.

Aims & Objectives:

- To argue the necessicity for correctness of software implementation wrt. requirements and in the context of domains,

- to introduce the notions of (i) implementation (cum refinement) relations, (ii) specification language proof systems, (iii) model checking and (iv) software testing,
- to cover these latter notions as they are found in the RAISE and VDM 'methods' (i–ii) — while otherwise referring to special textbooks and monographs on software development verification, and, briefly, explaining the ideas of model checking (iii), and software testing (iv), and
- to illustrate a number of actual design proofs — while, as said, only "telling the stories" on model checking and testing.

Aims & Objectives:
- To argue, with *Michael Jackson,* that no one 'method', ie. no one comprehensive set of techniques suffice for all software development,
- to introduce, "instead", the concept of a set of problem frames, each frame with its diversity of domain, requirements and design techniques and
- to illustrate, in particular such frames as the translation, the information system, the reactive systems, the workpiece systems, the connection, and other frames.

Aims & Objectives:
- To argue that major elements of software design can benefit significantly from the use of existing program packages and tools,
- to illustrate that many such platform uses relate more to interface and machine requirements than to domain requirements,
- to discuss and illustrate program package re–use and
- to relate a few of the current 'fashions' (Java, UML, OMG) to software design.

Aims & Objectives:
- To introduce the "standard" meanings of the terms: software quality, quality assurance and quality control,
- to discuss these meanings in the light of the software development approach taken in these lecture notes and
- to relate these issues to the ISO 9003 Standard.

Aims & Objectives:
- To investigate the various issues of software patents, intellectual property rights and copyrights,
- to investigate issues of software (development and product purchase) contracts, warranties and liabilities,
- to investigate issues of 'open', ie. 'free' software and
- to investigate the various issues of legalities of software education curricula, software engineers, software houses, software products — ie. of accreditation and certification.

Aims & Objectives:
- To introduce the notions of software development resource management: Strategic, tactical and operations management,
- to introduce the notion of project management: Planning, budgeting, accounting, monitoring & control, and project reviews and
- to introduce the notion of product management: Planning, marketing, pricing, sales and service.

Aims & Objectives:
- To discuss myths and commandments of "formal methods",
- to review what has been achieved with respect to a systematic, comprehensive enumeration of method principles, techniques and tools and
- to relate the message of these lecture notes to those of [other] textbooks on software engineering.

4.3 Some Comments

Yes, indeed: The proposed text book is rather comprehensive. And it requires mature students. We are aiming, not at college or undergraduate students, but at graduate students aiming for a serious, professional Masters degree in software engineering.

5 Conclusion

5.1 Software Engineering: Theory & Practice

The proposed text book in software engineering reflects a certain attitude, not common among todays software engineering text books. It promulgates a rather firm insistence on using suitable pairings of both informal and formal approaches, and it emphasizes respect for method principles, techniques and tools, for semiotics, for describing — as an occupation — ie. for composing elegant, enjoyably readable descriptions, and for the meticulous carrying out of phases, stages and steps of development (refinement, implementation).

The proposed textbook may seem far too encyclopedic, far too wide–spanning. That is, You may think that too many issues are "mixed up" (a) Semiotics, (b) documentation and (c) description principles. (d–g) Specification programming: Functional, imperative, logic and parallel. (h) Abstraction & modelling — with, if not "zillions", then at least quite a substantial variety of subsidiary principles and techniques. All the aforementioned, and then (i) the development triptych, and then (j) the problem frame orientation, and then, as if that was not enough, the more (k) mundane issues: Management, legacy, business process re–engineering, etc.

The success of the book depends, obviously, on: Showing that the software engineer of the next 10–15 years need be a hybrid of all of the above; and that all these seemingly diverse disciplines (itemized [a–k, etc.] above) form a consistent whole.

The proposed text book offers its loyal student another way of looking at software and at software engineering, one that is intellectually demanding as well as stimulating, one that emphasizes software engineering and its attendant documents as a universe of intellectual quality rather than the universes of material quantity hitherto offered by the engineering based on the natural sciences. As the careful reader will have noticed, from the table of contents for the proposed text book: Philosophy lies right in front of the software engineering: *What can be described ?* is, of course, always a burning question.

5.2 Experience

I have taught some 25 semester and some 25 two–three week courses in software engineering: The formal techniques programming methodology for large scale software systems, since 1977 — in Denmark and abroad — and to more than 1000 Danish students. Some of these former students, ie. MSc candidates, around some 200 or so, have meanwhile gone on to apply the principles, techniques and tools that are now being gathered in the text book discussed here, in actual, industrial and commercial projects since 1980. Some of them have founded a number of software houses (six–seven), and/or staff these companies, with a total of more than 500 staff. The companies still exists !

The UNU/IIST[6] significantly "features" training in and research into such software engineering, now to more than 100 candidates from more than 30 developing countries — who stay at Macau for 10–12 months — while applying the

[6] UNU/IIST: The UN University's International Institute for Software Technology in the Macau SAR, China

method to large scale infrastructure support software development: Railways, manufacturing, ministry of finance, telecommunications, airlines, port management, university management, electronic commerce, and much more.

5.3 Projection

The present lecture note writing project is the third such since 1977. For the period 1977–1991 the lecture notes were called *Software Architectures and Programming Systems Design.* For the period 1997–1999 the lecture notes were called *Software Engineering — A New Approach.* Now I call them *Software Engineering — Theory and Practice.* The first version focused only on software design. The second version — which was never fully completed — was a total rewrite, much in the style of this, the third version, but many topics had not then found their, what I now consider, proper place, and there were simply too many in–line examples, now consolidated in the extensive project documentation oriented appendices (A–G). I should ("easily"[7]) be able to complete the third version this fall. I will use it for my Spring 2002 course ! I hope for a publisher — a rather vain hope these days !

6 Acknowledgements

The work that lies behind the textbook here being discussed took its first beginning, for me, on a visit to the IBM Vienna Laboratory, March 1973. *Peter Lucas* fetched me and my wife at the airport. They wanted me to change from IBM Research to IBM Vienna. There was never any doubt. Before, during or since. It continued during my years at the IBM Vienna Laboratory working there with *Peter Lucas,* the late *Hans Bekić, Cliff Jones* and several others. And it continued thereafter and forever ! ?

Zillions thanks are due to the above, others, but in particular to *Peter Lucas:* A European Era goes to an end these days. *Peter:* Thanks from the depth of my heart and from the top of my brain, left and right, for a life–long inspiration. You above anyone else must be credited foremost for the successes then and later.

Thanks to *Cliff Jones* for getting me to Vienna, for sharing office at Vienna, for co–writing books, for co–organizing VDM and other events, for sharing in the establishment of VDM–Europe, now FME (Formal Methods Europe), and for all things in particular and no things in general ! To *Kurt Walk, Heinz Zemanek* and many other IBM Vienna Laboratory people. Those were my most formative years.

[7] Cf. the extensive material represented by previous versions and the referenced, more recent publications of mine !

Thanks also goes to *Ole N. Oest* (DDC Intl.) of later years, and to *Søren Prehn,*[8] *Chris W. George, Zhou Chaochen, Richard Moore, Tomasz Janowski, Bo Stig Hansen,* and other UNU/IIST[9] people — in Macau: A "New Vienna", though still going strong !

Special thanks are due to Hidetaka Kondoh (Hitachi, Tokyo) for valuable discussions. I wish to have many more such.

To countless students and colleagues, notably: to *Hans Bruun* — for making the CHILL and Ada projects possible, and hence for making Dansk Datamatik Center feasible, and hence, basically, for also making it possible for me to stand here today; to *Micheal Mac an Airchinnigh* (Trinity College, Dublin) — for raising VDM to a Categorical term, for enlightening us all, and for steadfastly showing us all how lucky we are in being able to enjoy the finest of all occupations of the last 30 and the next many years: Programming Methodology; and to my former student, now my peer (and much more), *Peter Gorm Larsen* (IFAD, Odense, Denmark) — for his enthusiasm, his agility and his carrying VDM–SL to the US of A, Japan, ..., "around the world !".

Thanks are also due to the IFIP WG 2.3 Members and Observers, notably *Sir Tony Hoare* and *Michael Jackson,* for invaluable prodding, inspiration and stimulation.

7 Bibliographical Notes

Since this paper is about a book I am trying to write I have taken the liberty of primarily referencing my own publications: They are then intended to serve as examples of the kind of style and treatment the book will feature.

First co–authored and co–edited VDM "books" were [Bjørner and Jones (78), Bjørner and Jones (82)]. Cliff Jones independently [co–]published [Jones (80), Jones (86a), Jones (90a), Jones and Shaw (90)] — with many more references expected in his paper for this Colloquium. The most recent VDM–SL book is [Fitzgerald and Larsen (97)] — with, I believe, one more imminent. Thoughts on software engineering education based on formal techniques, a basis for the present paper are published in [Bjørner (93a), Bjørner and Cuéllar (98)].

Methodology issues of VDM and related methods were first published in [Bjørner (89a)] and most recently in [McIver and Morgan (01)]. Our approach to domain engineering is covered in [Bjørner and Nilsson (92), Bjørner (98a)] and [Bjørner (00b)], to requirements engineering in [Bjørner (97)], and to software architectures and program organisation in [Bjørner (98b)]. In [Bjørner (80d)] additional thoughts around methodology and the software engineering triptych are

[8] Dansk Datamatik Center, CRI Intl., UNU/IIST, Terma Inc., now Frontbase — a long and still unfolding collaboration.

[9] UNU/IIST: The UN University's International Institute for Software Technology, Macau SAR, China. I was first and founding UN Director of UNU/IIST, 1991–1997.

found. The software engineering of the programming language domain based on the Vienna approach is found in [Beki et al. (74), Bjørner (77a), Bjørner (77b), Bjørner (78a), Bjørner and Oest (80)]. The software engineering of the information systems with database domain based on the Vienna approach is found in [Bjørner (80b), Bjørner (80c), Bjørner (82), Bjørner and Løvengrenn (82a), Bjørner and Løvengrenn (82b)].
General development approaches were published in [Bjørner (78b), Bjørner (81), Bjørner and Prehn (83), Bjørner (87a), Bjørner (89b), Bjørner (91a),] — as well as notably in Cliff Jones' books referenced above.

Software engineering planning in terms of software development (project) graphs are covered in [Bjørner and Nielsen (85), Bjørner (86a), Bjørner (86b), Bjørner (87b)].

Example domain models are presented in Robotics [Bjørner (93c)], Railways [Bjørner et al. (94), Bjørner et al. (95), Bjørner et al. (97a), Bjørner et al. (99a), Bjørner et al. (99b), Bjørner et al. (99c), Bjørner (00a)], Air traffic [Bjørner(95)], Strategic, tactical and operational resource management [Bjørner (99a)], Sustainable development [Bjørner (99b)], Projects and production [Bjørner (99c)], and E–Commerce [Bjørner (01)] (technical reports are imminent). Software engineering, using formal techniques, across Problem Frames, [Jackson (2001)], is studied, briefly, in [Bjørner et al. (97b)]. [Bjørner (99d)] provides a recent survey of 25 years of formal techniques.

References

[Beki et al. (74)] H. Bekić, Dines Bjørner, W. Henhapl, C.B. Jones, and P. Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, IBM Laboratory, Vienna, 20 September 1974.

[Bjørner (77a)] Dines Bjørner. Programming Languages: Linguistics and Semantics. In *International Computing Symposium 77*, pages 511–536. European ACM, North-Holland Publ.Co., Amsterdam, 1977.

[Bjørner (77b)] Dines Bjørner. Programming Languages: Formal Development of Interpreters and Compilers. In *International Computing Symposium 77*, pages 1–21. European ACM, North-Holland Publ.Co., Amsterdam, 1977.

[Bjørner (78a)] Dines Bjørner. The Systematic Development of Compiling Algorithm. In Amirchahy and Neel, editors, *Le Point sur la Compilation*, pages 45–88. INRIA Publ. Paris, 1979.

[Bjørner (78b)] Dines Bjørner. *The Vienna Development Method: Software Abstraction and Program Synthesis*, volume 75: Math. Studies of Information Processing, RIMS, Kyoto, August 1978 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Bjørner (80a)] Dines Bjørner, editor. *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Bjørner (80b)] Dines Bjørner. Formalization of Data Base Models. *[Bjørner (80a)]*, pages 144–215, 1980.

[Bjørner (80c)] Dines Bjørner. Application of Formal Models. In *Data Bases*. INFOTECH Proceedings, October 1980.

[Bjørner (80d)] Dines Bjørner. Formal Description of Programming Concepts: a Software Engineering Viewpoint. In *MFCS'80, Lecture Notes Vol. 88*, pages 1–21. Springer-Verlag, 1980.

[Bjørner (81)] Dines Bjørner. The VDM Principles of Software Specification and Program Design. In *TC2 Work.Conf. on Formalisation of Programming Concepts, Peniscola, Spain*, pages 44–74, LNCS Vol. 107, 1981. IFIP, Springer-Verlag.

[Bjørner (82)] Dines Bjørner. Realization of Database Management Systems. In *[Bjørner and Jones (82)]*, chapter 13, pages 443–456. Prentice-Hall, 1982.

[Bjørner (86a)] Dines 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.

[Bjørner (86b)] Dines 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.

[Bjørner (87a)] Dines Bjørner. On The Use of Formal Methods in Software Development. In *Proc. of 9th International Conf. on Software Engineering, Monterey, California*, pages 17–29. IEEE, April 1987.

[Bjørner (87b)] Dines Bjørner. The Stepwise Development of Software Development Graphs: Meta-Programming VDM Developments. In *See [Bjørner et al. (87)]*, volume 252 of *Lecture Notes in Computer Science*, pages 77–96. Springer-Verlag, Heidelberg, Germany, March 1987.

[Bjørner (89a)] Dines Bjørner. Towards a Meaning of 'M' in VDM. In E.J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State–of–the–Art Reports, pages 137–258. Springer-Verlag, Heidelberg, Germany, 1991. An IFIP TC2 Seminar, Persepolis, Brasil.

[Bjørner (89b)] Dines Bjørner. Specification and Transformation: Methodology Aspects of the Vienna Development Method. In *TAPSOFT'89*, volume 352 of *Lab. Note*, pages 1–35. Springer-Verlag, Heidelberg, Germany, 1989.

[Bjørner (91a)] Dines Bjørner. Formal Software Development: Requirements for a CASE. In *European Symposium on Software Development Environment and CASE Technology, Königswinter, FRG, June 17–21*. Springer-Verlag, Heidelberg, Germany, 1991.

[Bjørner (91b)] Dines Bjørner. Formal Specification is an Experimental Science (in Russian). *Programmirovanie*, 6:24–43, 1991.

[Bjørner (92)] Dines Bjørner. Trustworthy Computing Systems: The ProCoS Experience. In *14'th ICSE: Intl. Conf. on Software Eng., Melbourne, Australia*, pages 15–34. ACM Press, May 11–15 1992.

[Bjørner (93a)] Dines Bjørner. University Curricula in Software Technology. Technical Report 7, UNU/IIST, P.O.Box 3058, Macau, March 15 1993. Keynote address: IFIP TC3 WG3.4/SRIG-ET (SEARCC) International Working Conference 1993: *Software Engineering Education*, Hong Kong, September 28 – October 2, 1993.

[Bjørner (93b)] Dines Bjørner. Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. Technical Report 12, UNU/IIST, P.O.Box 3058, Macau, 7 November 1993. Appendix — on a railway domain model — by Søren Prehn and Dong Yulin, Published in *Proceedings from first ACM Japan Chapter Conference*, March 7–9, 1994: World Scientific Publ., Singapore, 1994.

[Bjørner (93c)] Dines Bjørner. *Formal Models of Robots: Geometry & Kinematics*, chapter 3, pages 37–58. Prentice-Hall International, January 1994. Eds.: W.Roscoe and J.Woodcock, *A Classical Mind*, Festschrift for C.A.R. Hoare.

[Bjørner (95)] Dines Bjørner. Software Systems Engineering — From Domain Analysis to Requirements Capture [— an Air Traffic Control Example]. Technical Report 48, UNU/IIST, P.O.Box 3058, Macau, November 1995. Keynote paper for the *Asia Pacific Software Engineering Conference*, APSEC'95, Brisbane, Australia, 6–9 December 1995.

[Bjørner (97)] 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.

[Bjørner (98a)] Dines Bjørner. Challenges in Domain Modelling — Algebraic or Otherwise. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, March 1998.

[Bjørner (98b)] 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.

[Bjørner (99a)] Dines Bjørner. Domain Modelling: Resource Management Strategics, Tactics & Operations, Decision Support and Algorithmic Software. In J.C.P. Woodcock, editor, *Festschrift to Tony Hoare*. Oxford University and Microsoft, September 13–14 1999.

[Bjørner (99b)] Dines Bjørner. A Triptych Software Development Paradigm: Domain, Requirements and Software. Towards a Model Development of A Decision Support System for Sustainable Development. In ErnstRüdiger Olderog, editor, *Festschrift to Hans Langmaack*. University of Kiel, Germany, October 1999.

[Bjørner (99c)] Dines Bjørner. Project Information, Monitoring and Control Systems — A Domain Analysis. Technical report, Dept. of Informatics and Mathematical Modelling, Technical University of Denmark, Bldg. 322, DK–2800 Lyngby, Denmark, 1999.

[Bjørner (99d)] Dines Bjørner. Pinnacles of Software Engineering: 25 Years of Formal Methods. *Annals of Software Engineering*, 2000. Eds. Dilip Patel and Wang Yi.

[Bjørner (00a)] 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.

[Bjørner (00b)] Dines Bjørner. Domain Engineering, A Software Engineering Discipline in Need of Research. In *SOFSEM'2000*, Lecture Notes in Computer Science. Springer Verlag, 18–24 November 2000.

[Bjørner (01)] 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.

[Bjørner and Cuéllar (98)] Dines Bjørner and Jorge R. Cuéllar. Software Engineering Education: Rôles of formal specification and design calculi. *Annals of Software Engineering*, 6:365–410, 1998. Published April 1999.

[Bjørner and Jones (78)] Dines Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978. This was the first monograph on *Meta-IV*.

[Bjørner and Jones (82)] Dines Bjørner and C.B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[Bjørner and Løvengrenn (82a)] Dines Bjørner and H.H. Løvengreen. Formal Semantics of Data Bases. In *8th Int'l. Very Large Data Base Conf.*, Mexico City, Sept. 8-10 1982.

[Bjørner and Løvengrenn (82b)] Dines Bjørner and H.H. Løvengreen. Formalization of Data Models. In *[Bjørner and Jones (82)]*, chapter 12, pages 379–442. Prentice-Hall, 1982.

[Bjørner and Nielsen (85)] Dines Bjørner and M. Nielsen. Meta Programs and Project Graphs. In *ETW: Esprit Technical Week*, pages 479–491. Elsevier, May 1985.

[Bjørner and Nilsson (92)] Dines Bjørner and J.F. Nilsson. Algorithmic & Knowledge Based Methods — Do they "Unify" ? — with some Programme Remarks for UNU/IIST. In *International Conference on Fifth Generation Computer Systems: FGCS'92*, pages (Separate folder, "191–198"). ICOT, June 1–5 1992.

[Bjørner and Oest (80)] Dines Bjørner and O. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Bjørner and Prehn (83)] Dines Bjørner and S. Prehn. Software Engineering Aspects of VDM. In D. Ferrari, editor, *Theory and Practice of Software Technology*. North-Holland Publ.Co., Amsterdam, 1983.

[Bjørner et al. (87)] Dines 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.

[Bjørner et al. (92)] Dines Bjørner, A.E. Haxthausen, and K. Havelund. Formal, Model-oriented Software Development Methods: From VDM to ProCoS, and from RAISE to LaCoS. *Future Generation Computer Systems*, 1992.

[Bjørner et al. (94)] Dines Bjørner, Dong Yu Lin, and S. Prehn. Domain Analyses: A Case Study of Station Management. Research Report 23, UNU/IIST, P.O.Box 3058, Macau, 9 November 1994. Presented at the *1994 Kunming International CASE Symposium*: KICS'94, Yunnan Province, P.R.of China, 16–20 November 1994.

[Bjørner et al. (95)] Dines Bjørner, C.W. George, and S. Prehn. Domain Analysis — a Prerequisite for Requirements Capture. Technical Report 37, UNU/IIST, P.O.Box 3058, Macau, February 1995.

[Bjørner et al. (97a)] Dines 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.

[Bjørner et al. (97b)] Dines Bjørner, Souleimane Koussobe, Roger Noussi, and Georgui Satchok. Michael Jackson's Problem Frames: . In Li ShaoQi and Michael Hinchley, editors, *ICFEM'97: Intl. Conf. on Formal Engineering Methods*, Los Alamitos, CA, USA, 12–14 November 1997. IEEE Computer Society Press.

[Bjørner et al. (99a)] Dines Bjørner, C.W. George, and S. Prehn. *Scheduling and Rescheduling of Trains*, *Industrial Strength Formal Methods,* Eds.: M. Hinchey and J.P. Bowen. FACIT, Springer–Verlag, London, England, 1999.

[Bjørner et al. (99b)] Dines Bjørner et al. Formal Models of Railway Systems: Domains. Technical report, Dept. of IT, Technical University of Denmark, Bldg. 344, DK–2800 Lyngby, Denmark, September 23 1999. Presented at the FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, Toulouse, France. Avaliable on CD ROM.

[Bjørner et al. (99c)] Dines Bjørner et al. Formal Models of Railway Systems: Requirements. Technical report, Dept. of IT, Technical University of Denmark, Bldg. 344, DK–2800 Lyngby, Denmark, September 23 1999. Presented at the FME Rail Workshop on Formal Methods in Railway Systems, FM'99 World Congress on Formal Methods, Toulouse, France. Avaliable on CD ROM.

[Dijkstra (76)] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Fitzgerald and Larsen (97)] John Fitzgerald and Peter Gorm Larsen. *Developing Software using VDM–SL*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.

[Gries (81)] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Hehner (84)] E.C.R. Hehner. *The Logic of Programming*. Prentice-Hall, 1984.

[Jackson (95)] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley Publishing Company, Wokingham, nr. Reading,1995.

[Jackson (97)] Michael A. 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.

[Jackson (2001)] Michael Jackson. *Problem Frames — Analysing and structuring software development problems.* ACM Press, Pearson Education. Addison–Wesley, Edinburgh Gate, Harlow CM20 2JE, England, 2001.

[Jones (80)] C.B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall, 1980.

[Jones (86a)] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986. Superceded by [Jones (90a)].

[Jones (90a)] C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990.

[Jones and Shaw (90)] C.B. Jones and R.C.F. Shaw. *Case Studies in Systematic Software Development.* Prentice Hall International, 1990.

[Knuth (68)] D.E. Knuth. *The Art of Computer Programming, 3 vols: 1: Fundamental Algorithms, 2: Seminumerical Algorithms, 3: Searching & Sorting.* Addison-Wesley, Reading, Mass., USA, 1968, 1969, 1973; newly revised 2000.

[McIver and Morgan (01)] Annabelle McIver and Carrol Morgan, editors. *Programming Methodology: Recent Work by Members of IFIP Working Group 2.3*, chapter Dines Bjørner: "What is a Method ?" — A Study of Some Aspects of Software Engineering. IFIP WG2.3. MacMillan, Oxford, UK, 2001. To be published.

[RaiseMethod (95)] The RAISE Method Group. *The RAISE Method.* The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[Reynolds (81)] J.C. Reynolds. *The Craft of Programming.* Prentice-Hall, 1981.

[RSL (92)] The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.