Models, Semiotics, Documents and Descriptions Towards a Software Engineering Literacy

Dines Bjørner

Informatics and Mathematical Modelling, Technical University of Denmark

Abstract: The problem to be dealt with in this paper is that of providing an analysis of some base concepts of software engineering: models and modelling; semiotics: pragmatics, semantics, and syntax; documents: informative, descriptive and analytic; and descriptions. With respect to descriptions, following Michael Jackson [Jackson 1995b], we propose principles and techniques for the construction of descriptive documents. This part of the paper constitues an analysis of the concepts of [Jackson 1995b] and proposes how to respond to those ideas in the context of formal specification.

We believe the paper contributes to a software engineering methodology by emphasising the importance of, and providing development principles and techniques for a *literacy of software engineering;* by emphasising the interplay between being informal and being formal — including provision of principles for selecting one, the other or both.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering—*Programming Methodology*

General Terms: Model, Modelling, Literacy, Semiotics, Document, Description, Method, Principle, Technique

Additional Key Words and Phrases: Pragmatics, Semantics, Syntax, Domain Enginerering, Requirements Engineering, Software Design, Formal Specification, Iconic Model, Analogic Model, Analytic Model, Prescriptive Model, Descriptive Model, Extensional Model, Intensional Model, Synopsis, Rough Sketch, Narrative, Terminology, Ontology, Annotation, Designation, Definition, Refutable Assertion, Type, Value, Axiom

Contents

Special: This paper is an extract of Chapters 6 and 13 of the author's lecture notes [Bjørner 2002].
General: This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001a; Bjørner 2001j; Bjørner 2001i; Bjørner 2001b; Bjørner 2001e; Bjørner 2001h; Bjørner 2001d; Bjørner 2001c]
Address: Informatics and Mathematical Modelling, Building 322, Richard Petersens Plads, Technical University of Denmark, DK–2800 Kgs. Lyngby, Denmark; E–Mail: db@imm.dtu.dk; URL: www.imm.dtu.dk/~db; Phone: +45 4525 3720, Fax: +45 4593 0074

1. INTRODUCTION

The engineer *walks the bridge between science and technology: Constructs technology based on scientific insight, and studies and experiments with technology so as to ascertain its scientific value.* Computer science is *the study and knowledge of the "things" that can exist inside computers.* Computing science is *the (programming methodological) study and knowledge of how to construct those things.*

Although programming includes the discovery of algorithms and data structures, and hence is very much a creative, human ingenuity driven process, we believe that a large set of guidelines can be established for software engineering conduct. In a number of papers, the present and [Bjørner 2001g; Bjørner 2001a; Bjørner 2001j], we shall contribute, we believe, with a number of such guidelines.

1.1 The Problem to be Addressed

The present paper is, in a sense, the first in the series referenced above. References [Bjørner 2001g; Bjørner 2001a; Bjørner 2001j], are, in that order, the next.

In this paper we address issues that we have learned are crucial in all phases, stages and steps[1] of software development. These issues are listed in the title of the paper: *Models, semiotics, documents* and *descriptions.* That choice of issues may seem strange to some, obvious — to the point of being trivial — to others. Yet we have learned that most software engineers fail to properly appreciate their significance and to act accordingly.

Thus this paper may seem "strange": It has no theories, it proposes no mathematical models of software development itself, it relies on no specific programming methodological approach[2]. The paper, although it uses `RAISE` Specification Language (`RSL`), is believed to bring material that is common to all these and other techniques and specification languages — ie. applicable when using either of these.

Instead the paper is intended to provide an intellectual framework for software engineering conduct, a framework of 'literacy'. A framework that is predicated by the observation that all we do, almost all we do, as software engineers, is to describe. We also analyse; and in all cases we document our descriptions and our analyses. The concepts we, as software engineers, create, are intellectual. They are not bound by laws of natural sciences, but by laws of mathematics: Algebra and logic. And the mastery of language plays a dominant rôle in this framework.

1.2 The Background

The series of papers referred to above represent an essence of the author's work during the past 25 years: Studying and applying formal specification (techniques) to large scale software developments, and studying how software engineers adopt and adapt to such formal techniques. The unfortunate term 'formal method' has been

[1] 'Phases', 'stages' and 'steps' are briefly characterised in Section 4.1.

[2] Whether `VDM` [Bjørner and Jones 1978; Bjørner and Jones 1982; Jones 1990; Fitzgerald and Larsen 1997], `Z` [Spivey 1988; Ince 1988; Imperato 1991; Potter et al. 1991; Spivey 1992; McDermid and Whysall 1992; Hayes 1993; Wordsworth 1993; Diller 1994; Rann et al. 1994; Ratcliff 1994; Barden et al. 1994; Bottaci and Jones 1995; Bowen 1996; Potter et al. 1996; Woodcock and Davies 1996; Jacky 1997], `RAISE` [George et al. 1992; George et al. 1995], `B` [Abrial 1996], `CASL` [Mosses 1996; Mosses 1997; Mosses 1999; Bidoit et al. 1998], `CafeOBJ` [Diaconescu and Futatsugi 1996; Nakagawa 1997; Nakagawa et al. 1997; Futatsugi and Diaconescu 1998], or other.

associated with this endeavour. In those 25 or more years such "formal methods" have gained currency: Are "spreading". Yet "old ways" of doing software still dominate. Latest Software Engineering text books sometimes contain, disturbingly, a chapter on Formal Methods. That is not the way we think of formal specification and verification techniques: As an "add on" one can conjoin practical software engineering. To us, formal techniques permeate most software development. *But how ?* The present paper (and its sequels) attempts to summarise an answer to this question.

1.2.1 *Methodology: Search for Principles, Techniques and Tools.* We need single out some terms and characterise them.

> By a *method* we understand a set of *principles* for `selecting` and `applying` a set of *techniques*, using a set of *tools*, in `analysing` a problem in order `efficiently` to `construct` an `efficient artifact` (here software[3]).

In the above characterisation the three terms: *principle*, *technique* and *tool* have been emphasised. To us, they form the essence of *What a method is !* [Bjørner 2001k]. The software engineer, we believe, should be guided by one or another method, hence by one or another sets of *principles*, *technique* and *tools*.

In the present paper we shall enunciate a number of such principles and techniques.

> By *methodology* we understand the study and knowledge of one or more *methods*.

Since, as poignantly argued by Michael Jackson, since there can be no one method that suffices for all kinds of software development, our study is not a method, but a methodology study. The software engineer — be she a compiler engineer, be he a real-time embedded systems engineer, or be she an information & database systems engineer — still need master several methods, thus reasonably "fluent" in several principles, several techniques. Each principle and each technique selected "in competition" with other applicable principles and techniques.

In this paper we shall thus single out, where deemed opportune, general *principles* and general *techniques* — applicable across the span of several kinds of software engineers (as exemplified above). As with *examples*, these are terminated with, instead of a period (.) by a ∎

1.2.2 *A Triptych of Software Engineering.* Before software[3] can be designed, their requirements must be engineered. And before requirements can be expressed, we must understand the domain.

Whether we *describe domains*, or *requirements*, or *software*, we are, in all cases *describing models*, and we are expressing these *models linguistically*. We have certain "axes to grind" (the *pragmatics*), we wish to express certain concepts (the *semantics*), and we have to write it down (the *syntax*). We thus *document* our *models: Inform* about them, *describe* the "thing" being *modelled*, and, usually, *analyse* these *models*. Hence *semiotics (pragmatics, semantics, syntax)*, *documents* and *descriptions* are at the foreground of the *software engineer*'s concerns.

[3]In general: *computing & communication systems.*

—By domain we understand the *application area*, the *"actual world"* in which some desired (or actual) *software*[3] is to reside (resp. resides).

Examples of man–made application areas could be railway systems [Bjørner 2001i], logistics of a freight forwarding systems [Bjørner 2001e], consumer/retailer/wholesaler/producer markets [Bjørner 2001b], project and production planning and monitoring & control systems [Bjørner 2001h], financial service industry [Bjørner 2001c], or health–care systems [Bjørner 2001d].

—By domain engineering we understand a set of science–based (but also practical) engineering activities which results in a set of *models*, ie. a set of *documents informing* about, *describing*, and *analysing* "all" the *phenomena* of the "actual world" (that are) relevant to *requirements engineering* — and usually more than that !
Which these *phenomena* are, how to *identify* them, how to *inform* about them, and how to *describe* and *analyse* them, is the subject of a *domain engineering methodology*.
A *domain description* is *indicative:* Describes what there is. It makes no reference — whatsover — to any (subsequently) *required software*, let alone to any *design* of such *software*[3].
Domain engineering is both a science and an art ! Whether a *domain description* is adequate ('sufficient'), or not, can usually be somehow ascertained through a concensus of as many relevant domain *stake–holders* as are feasible.

Just like the physicist studies nature, that is: Have nature as the domain, the domain engineer has a usually man–made application area as the domain.

Models of actual application areas are therefore to be understood linguistically and documented, in particular: Described.

—By requirements engineering we understand a set of science–based (but also practical) engineering activities which results in a set of *models*, ie. a set of *documents informing* about, *describing*, and *analysing* what is expected from *software*[3] that is to support activities of the *domain*.
Which these *requirements* are, how to *identify* them, how to *inform* about them, and how to *describe* and *analyse* them, is the subject of a *requirements engineering methodology*.
A *requirements description* is *putative:* Describes what *software* there shall be. It makes no reference — whatsover — to any *design* of such *software*[3].
Requirements engineering is both a science and an art ! Crucial parts of a *formal requirements description*, the *domain requirements description*, must stand in a formal relation to a similarly *formal domain description*. Whether a *requirements description*, additionally, is adequate (is 'sufficient'), or not, can usually be somehow ascertained through a concensus of as many relevant domain *stake–holders* as are feasible.

Requirements models are similarly to be understood linguistically, documented and described.

—By software[3] design we understand a set of science–based (but also practical) engineering activities which results in a set of *models*, ie. a set of *documents*

informing about, *describing,* and *analysing* the *construction*, *structure* and *behaviour* of the *software*[3] that supports required activities of the *domain.*
A *software description* is *imperative:* Describes how the *computing & communication system,* when following the prescription of the software design, behaves.
The software design is expected to satisfy, ie. be correct wrt., the *requirements description* in the context of the *domain description.*

Models now become computational models.

—By software engineering we thus understand a confluence of the practical (management etc.), the scientific and the technical issues of domain engineering, requirements engineering and software design.
We have chosen the term 'software engineering' to cover all three phases of what we shall thus call software development. We do so for want of a better word. The present characterisation is in line with current uses of the term — extends and makes more precise that conventional usage. Thus software development is a broader term than software design.

1.3 Related Work

Our quest for method principles and techniques is far from unique. What may be unique is our quest for ever larger frameworks for these principles and techniques. Usually it is common, in programming methodology, to contribute with one or another beautiful principle or technique. We are not satisfied if that is all we could contribute. The software engineer needs many and diverse principles. No one method, ie. no one set of principles and techniques (nor tools) suffice for all kinds of software development. The "larger scale" general principles and techniques that we are searching for, studying, experimenting with and proposing, seem to include such which apply, however, to not just compiler development, database management system development, real–time embedded systems development, etc., but also — *"some principles here, some techniques there"* — to the development of software systems for financial services [Bjørner 2001c], logistics [Bjørner 2001e], health–care [Bjørner 2001d], railways [Bjørner 2001i], electronic commerce [Bjørner 2001b], project & production planning, monitoring & control systems [Bjørner 2001h], health–care [Bjørner 2001d], etc.

Our quest is one of software engineering principles (etc.), and we find that most so–called Software Engineering textbooks fail, so is our opinion, in properly enunciating the principles and the techniques — they certainly do not dwell long on such issues as are covered in the present paper: Models, semiotics (pragmatics, semantics and syntax), nor on documents and descriptions from the linguistic viewpoints such as we do.

There is one source of related work, however, that has brought us much inspiration. And that is the work of Michael Jackson: [Jackson 1991; Jackson 1994b; Jackson 1994a; Jackson 1995a; Jackson 1995b; Zave and Jackson 1997; Jackson 1997; Jackson 2001]. Whereas Jackson studiously avoids deploying specific formalism — other than a casual use of first order predicate calculus — we are interested in the relationship between informal and formal expressions. Jackson emphasises the principles, searches for them, proposes a great number of these, discusses them, delightfully so — to unravel their essence. We wish to press on: Find (delineate) and propose further principles, and to see the extent to which differently named method concepts (viz.: different principles, different techniques, different tools) result, if at all, in different formal expressions. In this paper some concepts of documents, Section 4, and the main three concepts of descriptions (designations, definitions, and refutable assertions), Section 6, relate directly to concepts introduced by Jackson [Jackson 1995b]. Our contribution has been to further analyse Jackson's ideas and provide techniques that we believe will help the software engineer in constructing (especially formal) descriptions. We also question a few of his observations: Showing that some of them cannot be distinguished in a semantic world.

1.4 Structure of Paper

The four main issues of the paper title are covered in respective sections, Models and Modelling in Section 2, Semiotics in Section 3, Documents in Section 4, and Descriptions in Section 6. Models (etc.) is a subject that can be treated orthogonal to the other three, and has been singled out for treatment first. Semiotics: The pragmatics of why we deal with certain issues in forming concepts in any of the phases, stages and steps of software development (the social context of our choice), the semantics of these concepts, and the syntax we deploy when expressing the concepts — all this underly documents and descriptions. Software, to us, is a set of documents, and includes — besides program code — those that document the entire software development, the installation manuals, the maintenance manuals, and the user manuals. There are other documents being developed in the course of software development, but the ones mentioned above typically constitute the documents delivered to software users. Documents, to us, contain informative, descriptive and analytic parts — and Section 4 thus discusses their composition: Those that are part of the development process but not part of customer deliverables, those that are rough, "back of an envelope" sketches which help us form and clarify concepts, those that narrate, informally and formally, and those that analyse. *How do find out what to describe, and having found out, how do we describe ?* — these are vexing questions bordering (was it not for our "end goal", computable models) philosophical questions. Section 6 deals with some fundamentals: Designations "start" a *description,* definitions "fill" and complement it, and refutable assertions "make it interesting". Although Section 6 brings examples of abstract models it is not the intention of this paper to cover principles and techniques of abstract modelling. See [Bjørner 2001g]. Section 6 takes its departure point in Michael Jackson's delightful [Jackson 1995b], but goes beyond that book. Each of the three concepts: Designations, definitions and refutable assertions are shown their possible counterparts in formal specifications — and we point out complexities not obvious from [Jackson 1995b].

2. MODELS AND MODELLING

A model is an abstraction of some *universe of discourse*. Modelling is the act (or process) of identifying appropriate concepts and choosing appropriate abstractions in order to construct a model which reflects appropriately on this *universe of discourse*. We shall use the term model interchangeably to stand for the model description and the model "itself".

We shall, in this section, not cover principles or techniques for concept identification or abstraction choice, but only, and then just briefly, analyse the concept of 'model' itself. Other current papers, for example [Bjørner 2001g; Bjørner 2001a; Bjørner 2001j], cover such principles and techniques, and yet other current papers, [Bjørner 2001b; Bjørner 2001i; Bjørner 2001e; Bjørner 2001d; Bjørner 2001h; Bjørner 2001c], show actual identification of concepts (from respective domains: (1) E-Market, (2) railways, (3) logistics, (4) health–care, (5) project & production planning, monitoring & control, and (6) financial services).

2.1 Models

One can look at models from at least three viewpoints. The next 5+4+4 subsections cover three such clusters of model attributes.

2.1.1 *Iconic Models.* An iconic model is an "image" of some universe of discourse.
Example: A road sign showing, from the rear, the silhouette of an automobile and, under it, two crossing curves, designates that the road ahead may be slippery∎

2.1.2 *Analogic Models.* An analogic model resembles some other universe of discourse than the universe of discourse purported to be modelled.
Example: We are, by now, used to icons: A computer operating system (visual display unit) pictograph of a trash can (together with related functions) designates a means for deletion of electronic documents∎

2.1.3 *Analytic Models.* An analytic model allows analysis which (i) can either be used for such experiments (incl. computations) that may help determine whether the model appropriately models some universe of discourse, and, if so, (ii) can help in the design of technology that support operations in that universe, and/or (iii) predict future behaviour of that universe.
Example: Typically a formally specified software[3] requirements model is analytic in that it can serve as a basis for systematic development of the required software; and, similarly, a domain model can serve as a basis for similar development of domain requirements∎

2.1.4 *Discussion: Iconic, Analogic and Analytic Models. Examples:* So, typically, the domain, requirements and abstract software design descriptions designate analytic models. Additionally, an automobile computing system itself may interact with users via for example graphic display units which show either iconic or analogue models: *turn left or right, down a street,* respectively a *dead mans skeletal cranium* (possibly designating `danger ahead`)∎

2.1.5 *Icon, Analogy and Analysis Principles and Techniques. Principles:* In computer human interface designs choose appropriate iconic or analogic models and display these appropriately. Let the analytic models of domains, requirements and/or abstract software designs serve as bases for subsequent stages and steps of development ■

Techniques: Typically iconic and analogic models, when relevant in a computing context, calls for graphics expressions, whereas analytic models are formally specified ■

2.1.6 *Prescriptive Model.* A *prescriptive model* models something as yet to be implemented. *Examples:* Software requirements (prescribe software functionalities etc.), software architecture designs (prescribe abstract computing system behaviour) and program code (prescribe execution sequences) ■

2.1.7 *Descriptive Model.* A *descriptive model* models something already existing. *Examples:* Domain descriptions ■

2.1.8 *Discussion: Prescriptive and Descriptive Models.* Thus one should use the term prescription if and when reference is made to for example software requirements (descriptions). We will, however, use the term description for both descriptions and prescriptions. Michael Jackson [Jackson 1995b] discusses the related issues of *indicative, putative,* and *imperative* modes of specification.

2.1.9 *Prescription and Description Principles and Techniques. Principle:* The *prescriptive modelling principle:* When informally *putatively* describing what there shall be, then use the term 'shall'; do not leave doubt as to whether something prescriptive, or other, is meant ■

Principle: The *descriptive modelling principle:* When informally *indicatively* or *imperatively* describing what there is, respectively how it is, then use the term 'is'; do not leave doubt as to whether something descriptive, or other, is meant ■

Techniques: Since formal, non-modal specification languages cannot distinguish between indicative, putative and imperative modes of specification, we cannot semantically "match" the informal 'shall' or 'is' with respective formal, ie. mathematical operators ■

2.1.10 *Extensional Model.* An *extensional model* ("black box") models something as if observed by someone external to the *universe of discourse* (viz.: external to the software).

The extension of the word 'railway net' is the set of all railway nets, whether existing or planned, or just conceivable, and with no further qualification than being proper railway nets. The extension of the sentence 'the author of this paper' is the person named Dines Bjørner. Extensions refer to (empty, singleton, finite or infinite) sets of objects.

2.1.11 *Intensional Model.* An *intensional model,* in addition to modelling external behaviour also ("white box") models the internal structure of the *universe of discourse* (viz.: the software design).

The intension (the sense) of the word 'railway net' includes the particulars of the topology, rail curves and slopes, running over bridges, through tunnels, along lakes, etc. The intension of the sentence 'the author of this paper' is the person that is 183 centimeters of height, weighs — well too much, has a hare lip, is almost bald, etc. Intensions stand for concepts.

2.1.12 *Discussion: Extensional and Intensional Models.* Domain, requirements and, to some extent also software architecture design models, are usually, or preferably extensional. Program code is necessarily intensional.[4]

2.1.13 *Extension and Intension Principles and Techniques. Principle:* The *extensional modelling principle:* When specifying properties aim, if possible and when most appropriate, at property–oriented models.

Principle: The *intensional modelling principle:* When considering efficiency of software choose model–oriented models, that is: Specifications expressed directly in terms of such mathematical entities and scalars, sets, Cartesians, lists, maps and functions.

Techniques: For either of the extensional or intensional modelling approaches, choose, when formalising the descriptions, such techniques as are presented in for example [Bjørner 2001g; Bjørner 2001a; Bjørner 2001j].

2.2 Modelling

The process of creating models is called modelling.

2.2.1 *Property Oriented Modelling.* In modelling we can focus, as for extensional models, on the properties.

Techniques: Usually that is done, in most specification languages, by using algebraic concepts such as sorts (abstract types), function signatures and axioms that relate type values and function values∎

Such property oriented model descriptions intend to define a class of mathematical models, where by this latter term (mathematical model) we now, here, mean a mathematical entity. We shall — without loss of generality — think of a model as a binding of the names of the property oriented model description to mathematical entities (such as scalar values, sets, Cartesians, lists, maps, functions, or [when schemes, classes or objects are defined, then their names may be bound] to models themselves).

Principle: We advocate using property oriented descriptions when describing domains and requirements. We also, when making use of auxiliary data types in software designs, advocate defining these algebraically, ie. in a property oriented manner∎

2.2.2 *Model Oriented Modelling.* Instead of providing a property oriented model description which intends to denote a set of one or more mathematical models, we can, in a model oriented description "directly" designate that set of models.

Technique: Model oriented descriptions are usually expressed by the explicit use of scalar values, sets, Cartesians, lists, maps and functions∎

Example: We exemplify models by an abstract example, expressed both property

[4]In a wider context than for example software engineering, more precisely, in the context of philosophy, the terms intension and extension, to some philosophers, have basically the following meanings: Intension, the connotation, or sense, of an expression, as opposed to extension: Those things signified by the expression. The intension of a predicate expression is typically a concept, whereas the extension of a predicate expression is the set of "things" (objects) that "fall under" the concept which is the intension of that predicate expression. See [Audi 1995; Honderich 1995] for appropriate entries.

oriented (the sorts A and B exemplify property orientedness, and the concrete type C and the function f, model orientedness:

type
  A, B, C = A × B
value
  a,a′,a″:A, b,b′,b″:B
  f: → C
  f() ≡ (a,b) ⌈⌉ (a′,b′) ⌈⌉ (a″,b″)

The above specification denotes a set of models. Each of these models, according to the semantics of the RAISE Specification Language (RSL [George et al. 1992; George et al. 1995]), map A into any type of mathematical entities, likewise B. C is mapped into Cartesians of A and B values as are determined by the specific model. a, a′, a″, b, b′, and b″ are mapped into (arbitrary) such A and B values as are determined by the specific model. f is mapped into a function which non–deterministically (internal choice, ⌈⌉) yields either of one of the tree pairs described. Since there can be infinitely many interpretations of A, B, C, a, a′, a″, b, b′, b″ and hence f, there are similarly very many models. Each model's A and B bindings constrain those of C, a, a′, a″, b, b′, b″ and f ∎

Principle: We advocate using model oriented descriptions when describing software designs. Sometimes describing some domain phenomena or requirements in a property oriented way becomes too cumbersome and we then advocate trying whether an abstract model oriented description, which normally is textually shorter, reasonably well expresses properties ∎

2.2.3 *Discussion.* It is more an art than just an engineering technique to construct pleasing model descriptions. There are no "cast iron" concrete rules for exactly when to model in the property oriented style, when in the model oriented manner. Judicious choice, often based on experiments, and on considerable experience in reading someone else's pleasing models are usual prerequisites for good modelling.

2.3 Other Model Attributes

We only touch upon the rôle attributes.

2.3.1 *Rôle of Models.* We pursue modelling for one or more reasons: (i) *To gain understanding:* In the process of modelling we are forced to come to grips with many issues of the *universe of discourse.* (ii) *To get inspiration and to inspire:* Abstraction often "invites" such generalisations that induces, in the writer, or in the reader, desires of change. (iii) *To present, educate and train:* A model can serve as the basis for presentations to others for the purposes of awareness, education or training. (iv) *To assert and predict:* A mathematical, including a formal model, usually allows abstract interpretation — in the "vernacular": calculations, computations — that simulate, estimate or otherwise express potential properties of the *universe of discourse.* (v) *To implement:* Two kinds of "implementations" can be suggested: (v.1) In *business process re–engineering* we propose the *re-engineering* of some *domain* on the basis of a *domain model;* and, (v.2) in *computing systems design* we base the *development* of *requirements* on a *domain model* and we base *software* *design* on the *requirements [model(s)].*

2.3.2 *Discussion. Principle:* When first starting out to create a model express precise rôles for that model — as part of its preamble informative document: Which *pragmatics* (one or more) of the following: *(i) gaining understanding, (ii) inspirational, (iii) presentational, (iv) educational, (v) training, (vi) assertional, (vii) predicative,* and/or *(viii) implementational.*

2.4 The Identification Principle

We emphasise the most crucial aspects (α, β, γ) of models and modelling:

Principle: Modelling hinges upon: (α) There is the *problem domain:* The apriori given *universe of discourse;* there is (β) the *mathematical structure* of the *model,* ie. "the model itself"; and there is (γ) the *identification* of the relationship between the former two: The *problem domain* and the *model*∎

The principle expresses that the developer must constantly keep the above "triplet" in mind.

The model is not the universe of discourse, only a representation of it. Whatever properties can be argued about the model are not, necessarily, properties of the universe of discourse.

2.5 Discussion

The term 'model' seems to have been used in two senses: To cover the syntactical description and to cover its semantical meaning. Thus a property oriented model (description) — as well as a model oriented model (description) — denotes a set of one or more mathematical models. The property oriented descriptions are intended to be more abstract in not prescribing exactly which mathematical entities (such as scalars, sets, Cartesians, lists, maps and functions) are intended — whereas the model oriented descriptions more explicitly hints at such models. We have more to say about this in Section 3.2.2.

3. SEMIOTICS

3.0.1 *Language as Instrument of Thought and Action.* Language, whether informal or formal, whether our mother tongue or a professional (ie. occupational) language, determines much of our intellectual thinking as well as our material action. Mastery of professional as well as formal language linguistic notions is therefore deemed indispensable. *Semiotics* is here seen as the confluence of *pragmatics* (the social use of language), *semantics* (the meaning of what we express), and *syntax* (the form of our expressions). In this section we shall briefly survey these notions in the context of domains, requirements and software designs.

3.0.2 *Linguistics: Semiotics, Documents and Descriptions.* Three concerns motivate this paper's present and next sections. First (1), along the *semiotic* line, we often find, in informal documentation, that software engineers, in their documentation, confuse what is *pragmatics* and what is *semantics.* In conjunction with this, but independently thereof, we find that software engineers may be rather careful about *syntax* but fail to convey properly what the meaning, the *semantics,* is of that *syntax.* Then (2), along the *documentation* line, we then find that the software engineers do not, in our mind, properly separate what is *informative,* what is *descriptive* and what is *analytic.* And (3), finally, along the *descriptive* line, we find that, although it may be easy, once technically grasped, to construct formal descriptions, many have great difficulties only developing and presenting informal descriptions (narratives). The problem seems to be that it is hard, just using an informal descriptive approach, to determine *"where to start ?," "what to describe ?,"* and *"how to informally describe ?".* The problem seems to lie in properly understanding what constitutes an informal language description: "What is a description ?".

3.0.3 *Systems as Language Implementations.* Software engineers more often develop software for *systems* than for *languages,* it is believed ! In this subsection we shall argue that *'systems',* such as we understand them, can be viewed as 'implementations' of *languages.*

By *languages* we shall understand structures of *pragmatics, semantics* and *syntax.* Usually one views languages first *syntactically,* then *semantically.* And usually one does not "formally" consider *pragmatics.* We take *pragmatics* to be the most important, *semantics* to be next in line, and *syntax* to be necessary, but, in some sense, accidental: Many different forms of *concrete syntax* can be demonstrated with seemingly different *syntactical structures* (entities, values) having identical *meanings.* What the *language structures* are, formally, we shall not go into in detail other than claiming, that for the languages that we are interested in, the *formal languages,* one can give them *formal syntax* and *formal semantics,* but one cannot formalise their *pragmatics !*

By a *system* we shall understand a set of *processes,* with processes having *states* and exhibiting observable *behaviours:* Sequences, *traces,* of *actions* and *events,* where *actions* manipulate process *states* (reads, updates), and where *events* are seen as *[synchronised] communications* between *processes* — where we include a notion of *environment: "That which is 'outside' the set of specific processes being considered".* Usually 'practical software engineers' draw pictures of *systems* as *diagrams* of *boxes* and *arrows.* The *boxes* then are the (ie. denote) *processes,* and the *arrows,* that either infix boxes or are incident upon or leave boxes (from, resp. to the environment) stand for *channels* that allow *[synchronised] communication.*

The connection to *languages* is the following: The *systems* are usually such "things" as *railway systems, financial service systems, logistics systems, health–care systems, project and production systems,* etc. These "systems" are inhabited by people, and they communicate through language and other sign structures with the system, ie. with other system stake–holders, with God given phenomena, or with man–made artifacts of the system — and the system likewise "communicates back". Another way of looking at this is that the various traces of events [as observed over the (synchronous, un–buffered, or asynchronous buffered) channels] constitute sentences, possibly infinite such ! Hence it is natural to speak of a compiler system, or a run–time system for our artificial, man–made programming languages, as natural as it is to speak of the professional language spoken by the staff of a railway system, or a securities exchange system, or a health–care system, etc.

Principle: In understanding a domain focus on the language of terms: Nouns and verbs spoken by the domain stake–holders in relation to that domain∎

The above view of "Systems as languages" is just a conjecture. It is certainly not meant to cover more general situations than those for which we seek domain descriptions, requirements and software designs. But for those, we have found that the 'conjecture' serves a constructive purpose: Namely that of identifying "the professional languages of systems".

3.0.4 *Purpose and Structure of Section.* This, therefore is the motivation for the present and the next two sections, and thereby for most of this paper. But first *semiotics,* a "sum" of *pragmatics, semantics* and *syntax.* After these three topics are surveyed we further review semantics — in the light of syntactic and semantics, static (contextual, constant) and dynamic (temporal, variable) values.

3.1 Pragmatics

We distinguish between the *pragmatics of (a specification) language,* and the *pragmatics of a software development document (part).*

By the *pragmatics of language* we shall here understand the study and knowledge about the use of words, sentences, and structures (aggregations, systems) of sentences, and how contexts — that is the historic, previously uttered words, sentences, etc. — affects the meaning of words, sentences, etc.

By the *pragmatics of a specific software*[3] *development document (part)* we mean such issues as why a, some or any special description was given: The rôle the specific document (part) plays wrt. the client, or wrt. the software, or in the software development project itself. That is, the social context in which that particular development takes place, why it was initiated, what political, socio–economic and/or other aims it may have.

We shall focus on the latter concept of pragmatics.

Example: In introducing, in starting out, the development of software for the support of the creation, manipulation and use of documents, usually civil and mechanical engineering drawings, for a, or any, large road, bridge and tunnel construction project, it may be appropriate to state such things as: *"A Fehmern–sound*[5] *road/-bridge and/or tunnel is needed in order to bring about further regional economic and social development, to provide smoother travel between Scandinavia and the European continent, to alleviate traffic congestion in Denmark and to improve road safety."* ■

No reference is made to the civil engineering documents, nor to operations on these, the main focus of the software development being actually aimed at in this context. Once statements are made about the civil engineering documents, those statements may be descriptive and contain information about their syntax and semantics, hence are no longer of pragmatic nature.

Example: "To ensure maximum acceptance of the new computing system, its user interfaces must be so–called 'user friendly'" ■

What is meant by 'user friendly' is not explained. Once that is detailed, that detailing is descriptive, not informative, and belongs, not to pragmatics, but to syntax and semantics. But the full example illustrates a pragmatic concern.

Example: "The software development project must adhere to 'best practices', and must ensure that development staff, through the project's use of advanced techniques and tool, stay abreast of the field !" ■

The first example illustrated socio–economic, traffic policy and safety concerns — something "totally" outside, we claim, the realm of the software itself and its developers. The second example "hinges" upon some aspects of software requirements, but really refers to psychological (and other) aspects of user behaviour, not with respect to the functionalities of the software, its relation to the domain, but with respect to the general, common human psyche — yet the acceptance concern seems more one of political correctness, one of "luring the market to buy the product", than of human user, individual concern (in which latter case it "borders" to a non–function interface requirements). The third example did not characterise the software in its social, economic or political context, but to those aspects of its developers — and is again an example of pragmatics.

It is not possible, given the richness of human life and the diversity of its many infra—structural components (to which software is being applied), to characterise, in "closed form", all the classes of pragmatic concerns. That is: There is no fixed "check list", other than the general characterisation of *pragmatics,* with respect to which we can test (analyse) whether a text is of *pragmatic* nature: We, as developers, must insert pragmatics (text parts) where needed, and avoid them where not needed.

3.2 Semantics, Models, Denotations and Values

One may distinguish, for later use, between the *semantics* of a *syntactic construct,* its *models, denotations* and its *values.*

3.2.1 *Semantics.* As for pragmatics, a distinction need be made between the *semantics of a (descriptive) language,* and the *semantics of a descriptive document (part).* Further we distinguish between informal and formal languages, respectively informal and formal descriptions. Notice that we constrain semantics, as a concept,

[5]Fehmern–sound separates South–eastern Denmark from North Germany

to apply only to descriptive languages and to descriptions. (Not to informative or analytic texts.)

By a language we, syntactically, mean a set of sentences (expressions, statements). A description can be considered a sentence of a language such that it has a well–defined semantics.

By the *semantics of a description* (in some (descriptive) language) we the meaning of that particular sentence (ie. description).

Example: The *semantic value* of the (syntactic, decimal) integer numeral '11' (or 'eleven', or the Roman numeral 'xi', or the binary integer numeral '01001l') is the mathematical entity, the integer number that is the immediate successor, respectively immediate predecessor, of the numbers designated by the numerals '10' and '12'.

If no wider context is spoken of, in connection with these integer numerals, than just these numerals themselves, then we say that the designated integer numbers are *models* of the numerals and that the integers are their *denotations*∎

By the *semantics of a (descriptive) language* we mean an explication of the meaning of each and every sentence of the language.

Example: The semantics of the language of decimal numerals can, for example, be explained as follows: If the numeral (n) is one of the digits '0', '1', '2', '3', '4', '5', '6', '7', '8', or '9', ie. d, then the semantics, $\mathcal{D}$, of that numeral (d) are the mathematical entities, the numbers otherwise named zero, one, two, three, four, five, six, seven, eigth, respectively nine. If the numeral can be written nd (ie., by the left–to–right juxtaposition of n and d, where d is a digit and n is a numeral, then the semantics can be mathematically expressed as $10 * \mathcal{N} + \mathcal{D}$, where $\mathcal{N}$ is the semantics of the numeral n.

If no wider context is spoken of, in connection with these integer numerals, than just these numerals themselves, then we say that the integer number type is a, in this case the, *model* of the syntax of numerals and that the integer type is their *denotation*∎

An expression containing free identifiers usually, in the denotational sense, is given a semantics which is expressed as a function from (the) contexts (in which the free identifiers are ascribed meaning in the form of models, denotations or values) to values.

Example: Let one context (an environment ρ) ascribe the numbers seven and nine to the identifiers a and b, let another environment ρ' ascribe the numbers three, respectively nine to those identifiers, etc. The expression $a + b * 3$ now is either a function, or a set of functions, in the function space $\mathtt{ENV} \rightarrow \mathbf{Num}$ — from an, or all environments, `ENV`, to specific numbers, **Num**, namely those that arise as the result of an evalution of the expression in a given (one of these) environment(s).

We say that the above illustrated set of functions, from environments to integers, are models, are denotations, of the syntactical arithmetic expressions with free varaibles — where the environments bind variable identifiers to integers. We must be prepared for the eventuality, subject to which wider linguistic context the arithmetic expressions occur in, that the environments binds other identifiers to values of other semantic types∎

The above focused on what we shall later call dynamic semantics. See Section 3.4.

3.2.2 *Models.* By the *model of a description language* we understand one of the models of its semantics (which we took as a set of models). By a *model of a description* we understand an arbitrarily selected model among the set of models ascribed to that description by the semantics of the description language. [Any one model may give rise to several possibly, ie. alternative bindings of free identifiers to models (or values).]

Example: For the most recent example above there could be just one model, the one function, given: From one specific environment to a number value; or there could be several models∎

3.2.3 *Semantic Values.* By the *semantic value of a description* we understand a value of the description when evaluated in one of its arbitrarily selected models. Evaluation takes the free identifiers of the description and substitute, in their place, values found from the identifier bindings in similarly selected environments of the chosen model.

Section 3.4.1 introduces a spectrum of values: From syntactic (static) to semantic (dynamic). Thus that section expands the meaning of the above concept of *value of a description* to include models that contain such (mixed) values.

3.2.4 *Denotations.* By the *denotation* of a language construct (a description, or a part of it) we may thus mean a set of models, a model, or a value (arising from evaluation within a model) — depending on the context in which we view the syntactic entities. Among values can be functions. Typically functions from [such] contexts (ie. environments) [which bind free identifiers of the description (part) to their models, denotations, semantic values — as the case may be]. If we deal with imperative programming languages (with dynamically allocated asignable variables), that with such dynamic systems whose explication is furthered by the introduction of a state concept, then the denoted (model) functions additionally map from states, to values — where states map dynamically allocated variable locations (or system component identifiers) to semantic values.

3.2.5 *The Semantics Modelling Principle.* Important principles can now be formulated.

Principle: The *semantics modelling principle:* In any domain, requirements or software design specification (ie. description), focus first on what it is one wishes to express, ie. the semantics (not the syntax !)∎

This principle goes hand–in–hand, first with:

Principle: The *semantics model principle:* When formalising the description, select a specification language whose semantics "fits" the concept of models that one wishes, that is: The paradigm that "fits" the domain, the requirements or the software[3] design∎

And then with:

Principle: The *semantic value principle:* Determine, probably first, the semantic value types∎

Examples: The semantic value of the digit '0' is the integer whose name is zero. The semantic value of an identifier 'a', in a context which binds 'a' to the integer i is that integer. And hence the value type is (let's name it) `integer`∎

3.2.6 *Discussion.* Thus the four concepts: *semantics, models, values* and *types* (including *denotations*), are closely related. And hence they are often, as terms, used interchangeably, and sometimes, unfortunately, confusingly mixed.

3.3 Syntax

We distinguish between the *syntax of a (specification) language,* and the *syntax of a software development document (part).*

By a, or the *syntax of a (specification) language* we understand general rules for, and forms of, structures, be they sentential or otherwise. By a *formal syntax,* or a *grammar,* we understand a set of rules based on syntactical categories (types) — such that these rules allow us to both *generate* proper sentences and to *analyse* all text strings (respectively structures) in order to determine whether they are proper syntactical constructs, and if so, determine their (possibly ambiguous) composition.

By the *syntax of a software development document (part)* we mean an analysis of that part (sentential structure) that informs us as to its precise syntactical composition, and hence its possible models, denotations, or values.

Examples: Informal descriptions of syntax.

`(0) Real Numerals:` A *real numeral* consist of three parts: An integer numeral part, a period ('.'), and a fraction numeral part — in that order, left–to–right. The integer part is a finite sequence of digits. The fraction numeral part is a finite sequence of digits. These sequences are thought of as consisting of a digit preceded by an integer, respectively followed by a fraction numeral part. The following are digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9∎

`(1) Finite Trees:` A *finite tree* consists of a labelled *root* and a finite quantity of (zero, one or more) uniquely labelled *subtrees.* A *subtree* is a *tree. Labels* are further unanalysed entities. By a 'labelled root' or a 'labelled subtree' we mean an association (a "pairing") of a label with the root, respectively the subtree. (*Subtree labels* are said to be *branch labels.* A *tree* with no *subtrees* is said to also be a *leaf.*)

`(2) Infinite Graphs:` A possibly *infinite graph* consists of (a pair of) a possibly infinite set of unique *vertices,* and a possibly infinite set of *edges.* An *edge* is a pair of *vertices* such that these *vertices* ' "are in the" (the set of *vertices* of the) *graph* '. *Vertices* are further unanalysed entities. For both (1) and (2) the above lists the inductive and the basis clauses — to which we add the extremal clause: Only such structures whose construction (or successful analysis) follows from a finite number of applications of the above rules are *trees,* respectively *graphs*∎

Examples: Formal description of syntaxes of `(0) Real Numerals, (1) Finite Trees,` and `(2) Infinite Graphs:`

(0) $\mathrm{R} = \mathrm{I} \times \mathrm{F}$, $\mathrm{I} = \mathrm{D}|(\mathrm{I}{\times}\mathrm{D})$, $\mathrm{F} = \mathrm{D}|(\mathrm{D}{\times}\mathrm{F})$, $\mathrm{D} = 0|1|2|3|4|5|6|7|8|9$
(1) L, $\mathrm{T} = \mathrm{L} \times (\mathrm{L} \underset{m}{\rightarrow} \mathrm{T})$
(2) V, $\mathrm{G}' = \mathrm{V}\text{-}\mathbf{infset} \times (\mathrm{V}{\times}\mathrm{V})\text{-}\mathbf{infset}$
(2) $\mathrm{G} = \{|\ (\mathrm{vs},\mathrm{es}){:}\mathrm{G}' \bullet \forall\ (\mathrm{v},\mathrm{v}'){:}(\mathrm{V}{\times}\mathrm{V}){\bullet}\ (\mathrm{v},\mathrm{v}') \in \mathrm{es} \Rightarrow \{\mathrm{v},\mathrm{v}'\} \subseteq \mathrm{vs}\ |\}$

The second (2)–rule above explains the ' "are in the" *graph* ' constraint mentioned above∎

We have, perhaps tediously, given the above, perhaps obvious examples in order to emphasise that not just sentences but also, for example, data structures, or diagrams, etc., has, and may thus be given, a formal syntax.

Hence we should look for their semantics !.

Often software engineers, in fact most programmers, tend to not consider the issue of *"what is the semantics of my hash trie data structure ?* or *"... of my bill–of–lading document ?"*.

Example: The semantics of bill–of–ladings could, for example, be the set of all the possible traces (ie. sequences) of actions and events that the freight item designated by the bill–of–lading, undergoes. Thus the semantics of the syntax type of bill–of––ladings has, seen in some context, a set of models from these bill–of–ladings to their sets of traces∎

Principle: Whenever the syntax of some enterprise documents (such as social well-fare application forms, logistics systems bill–of–lading, etc.) are being developed, develp first their semantics. Then their syntax∎

We find that most software engineers, and well–nigh all text books, having 'Software Engineering' as a significant part of their title, fails on understanding, let alone adhering to the above principle.

3.3.1 *Abstract Syntax.* By an *abstract syntax* we understand a syntax description which focuses on the compositional properties of syntactic entities.

Examples: The tree and graph syntax examples given above focus on their properties and give no hint as how to draw neither trees nor graphs∎

3.3.2 *Concrete Syntax.* By a *concrete syntax* we understand a syntax description which (in addition to an abstract syntax focus, also) focuses on the presentational (textual, graphical, or other), ie. layout properties of syntactic entities.

Examples: The earlier informal syntax of real numerals focused on textual layout. The formal syntax of reals abstracted ("away") the period. One cannot say that because a Cartesian I×F lists I before F then is "come before" fs. If the syntax for trees or graphs (above) also included components that stood for (x, y) coordinates of vertices and (eg.) triplets of these for *Bezier* curve specifications for edges, then the appropriate syntax could be termed a concrete syntax∎

3.3.3 *Discussion. Principle:* Aim, whenever a syntax has to be expressed, "in principle", at "as abstract syntax" as well as "as simple an abstract syntax" as possible∎

Doing so often leads to discoveries of general, and useful concepts.

Principle: In both property and model–oriented descriptions, in both domain, requirements and software architecture design descriptions, use abstract syntax when (formally) defining structures. It is only when we need be concrete, in expressing, for example interface (CHI) requirements and when software designing their implementation, that we need turn to concrete syntax∎

Techniques: Techniques for defining abstract syntaxes are covered in for example [Bjørner 2001g]. Techniques for defining concrete syntaxes are covered in usual textbooks dealing with the tool of regular and context free grammars (such as BNF)∎

3.4 Abstract Interpretations

The above explication took a rather "binary" approach to language: There being *syntax* and there being *semantics*, and hence that there was only one form of *semantics* for a given set of syntactic forms.

As is well–known there is a whole spectrum of *semantics* definition styles. In general we shall call a *semantics* an *abstract interpretation.* To understand, briefly, what is meant by *abstract interpretation* we introduce a spectrum of *semantic values* spanning from static to "fully" dynamic.

3.4.1 *Static and Dynamic Values.* By a *static value* we mean a *syntactic value,* and by that we mean a *syntactical,* that is a *textual structure.*

Examples: Examples of static values are: Finite linear character strings, texts, as well as finite (syntactic) sets, finite (syntactic) Cartesians, finite (syntactic) lists and finite (syntactic) maps[6] of static values∎

By a *syntactic set, Cartesian, lists of map* we mean a textual structure.

Examples: Let $a_1, a_2, \ldots, a_m$ and $b_1, b_2, \ldots, b_n$, stand for not necessarily distinct static values. Then: (syntactic set) $\{a_1, a_2, \ldots, a_m\}$, (syntactic Cartesian) $(a_1, a_2, \ldots, a_m)$, (syntactic list) $\langle a_1, a_2, \ldots, a_m \rangle$, and (syntactic map) $[a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots, a_m \mapsto b_n]$, are examples of static, structured values∎

In this paper we shall, without loss of generality, assume only such static values as were outlined above.

By a (mixed) *dynamic value*[7] we mean a *mathematical value* – such as a static value, a number, a truth value, some arbitrary token value for which no representation is sought, and mathematical sets, Cartesians, lists, and maps of these, as well as algebras and functions over such. By prefixing sets, Cartesians, lists and maps with the term 'mathematical' we mean to express that the entities now are mathematical. That, for example means that they are subject to the mathematical operations that are definable over sets, Cartesians, lists, maps and functions (union, decomposition, head, application, composition, etc.).

3.4.2 *Static Semantics.* By the *static semantics of a formal language* we shall here, perhaps somewhat narrowly, and roughly speaking, mean a "syntactic semantics", a *well–formedness* function, that is, a *predicate* which determines whether sentences of the language satisfy certain relations solely between parts of these sentences — where any auxiliary predicate arguments solely contain syntactic quantities, ie. static values.

Example: The second (2)–rule for the example formal syntax of graphs illustrates a static semantics∎

3.4.3 *Dynamic Semantics.* By the *dynamic semantics of a formal language* we shall here, roughly speaking, mean a *semantic function* which determines the mean-

[6]A map can be understood as a set of pairs of domain/range elements. If the set of pairs is such that no two distinct pairs have the same domain element then the map is said to be deterministic. We shall, without loss of generality, assume deterministic maps.

[7]We could, as well have called statis, resp. dynamic values for syntactical, resp. semantic values — but since we wish, for historic and other reasons, to speak of static, resp. dynamic semantics (rather than syntactic semantics, resp. semantic semantics), we have remained with the historical names.

ing of any well–formed sentence — where auxiliary arguments typically embody dynamic values.

Example: When we wish to express, albeit abstractly, the intended (set of) computation(s) prescribed by programs of a programming language we give it a dynamic semantics∎

3.4.4 *Static and Dynamic Semantics.* Since there is a spectrum of dynamic values, ie. since these can contain mixed values, there is a spectrum of dynamic semantics. What was said earlier about models apply likewise: There is a spectrum of models, "from" static "to" dynamic.

Examples: (1) A semantics of a programming language which determines, for every well–formed program which defined identifiers are used where in the program text, such a *binding analysis* is a *static semantics.* (2) A semantics which similarly determines which input data (ie. which produced data) are consumed (incl. "used", applied) where, ie. such a *data flow analysis* is a *static semantics.* (3) A semantics which similarly determines possibly executable program paths, ie. not necessarily "executed" paths, such a *control flow analysis* is a *static semantics.* (4) A semantics which "mimics" the executions specified by a program text, ie. which deals with dynamic (ie. semantic) values, is a dynamic semantics∎

Principle: For every language (and, thus, cf. Section 3.0.3, system) identify its nouns and verbs, and to each of these establish their meaning. From compositions of nouns and verbs, etc., establish their meaning homomorphically (see Section 3.4.5 below)∎

3.4.5 *Homomorphisms.* The denotational principle call for two properties to be satisfied. One is that denotations, "in principle", are functions. The other is that the meaning of atomic (syntactic) constructs are simply expressible functions, and that the meaning of a composite (syntactic) construct can be homomorphically expressed as some function of the meanings of the components:

$\mathcal{M}(a) = \lambda x \bullet \mathcal{E}(x, a)$

$\mathcal{M}(c_1, c_2, \ldots, c_n) = \mathcal{F}(\mathcal{M}(c_1), \mathcal{M}(c_2), \ldots, \mathcal{M}(c_n))$

This technique of expressing functions homomorphically appears to be a nice "divide & conquer" technique in many domain, requirements and software design situations.

Principle: As a principle we can advice the use of the technique of homomorphic definitions and of homomorphic programming, that is: structuring programs so as to "exhibit" applicable homomorphisms more readily ∎

3.4.6 *Denotational and Operational Semantics.* By a *denotational semantics* we understand a semantics which is defined using the homomorphism principle.

By an *operational semantics* we understand a semantics which ascribes sequences (or traces) of configurations (contexts and states) of length two or more to syntactic constructs (length two if atomic, more if composite).

Technique: Operational semantics definitions are usually in the form of a set of *'next configuration' transition definitions,* one or more such transitions being defined for every syntactic construct or intermediate (not necessarily syntactically well–formed) forms of such constructs∎

Technique: Denotational semantics definitions are structured according to the homomorphic "programming" principle∎

3.4.7 *Discussion.* This subsection has touched upon a number of semantics issues which we group under the concept of 'Abstract Interpretation'. We refer to primarily two "schools" of scientific and technical work that relates to and expands on the topics above. There is the " school" anchored in Patrick Cousot's work on 'Abstract Interpretation' [Cousot and Cousot 1977; Cousot 1996; Cousot 1997b; Cousot 1997a]; and there is the "school" labelled 'Partial Evaluation' anchored in the works of three scientists: (1) Yoshiko Futamura [Futamura 1971; Futamura et al. 1991], (2) Andrei P. Ershov [Ershov 1977b; Ershov 1977a; Ershov 1978b; Ershov 1978a], and (3) Neil D. Jones [Jones et al. 1993]. The more mundane software engineering technologies of software testing and program (data and control) flow analysis have their theoretical background in abstract interpretation.

3.5 Discussion

We refer to proper text books on 'Programming Language Semantics' for theories of semantics, to proper text books on 'Automata Theory and Formal Languages' for theories of syntax, and to proper text books on 'Compiler Construction' for theories and practices of concrete syntax (ie. grammars).

We have brought the material of this section since we often find that software engineers take semiotics far too lightly, if they are at all aware of the concepts. We find that most practicing software engineers, as well as most, if not all standard text books on 'Software Engineering', focus too much and too early on syntax, and fail to pay proper attention to semantics, confuse semantic and pragmatic issues, and, for lack of proper mastery of semantics specification languages (including attendant specification techniques) — when indeed aware of semantic issues — are frustrated in their ability to express adequate semantics.

Principles: We therefore take as a guiding principle that of constantly, throughout all phases, stages and steps[8] of software development, considering as many and as generally the pragmatic, semantic and syntactic issues, and separating these clearly∎

[8]'Phases', 'stages' and 'steps' are briefly characterised in Section 4.1.

4. DOCUMENTS AND DOCUMENT PARTS

'Beauty is our Business', so titles David Gries a 'Festschrift' in honour of Edsger W. Dijkstra [Gries 1990]. A reference is implied to the belief — held also by the current author — that the artifacts we (as software engineers cum programmers) create, must possess beauty, and that that beauty can first be experienced by those who create the artifacts, and can be discovered and enjoyed, also by others, when reading descriptions of software development: From domains, via requirements to software designs. If the concepts that we have conceived for the domain, the requirements and the software — additional concepts as we move from a development phase into the next — if those concepts possess beauty, the conjecture goes on, then the software is beautiful: Computations and human interfaces, as prescribed by the software documents, can then be "felt pleasing" by the users.

All we (as software engineers cum programmers) create are documents ! And some of those documents directly form the basis for computations. There are no middle–men, no technicians, no craftsmen, who, as for the work of other kinds of engineers, through laborious processes, convert mechanical or civil engineering drawings to automobiles or bridges. The documents are our end–product. All maintenance of our software takes place through changes to documents, not through physical mechanical or building processes. Ours is a world of intellectual artifacts of quality and is based on mathematics — in contrast to other engineering worlds which are based on the natural sciences and where material quantity measures play a primary rôle: Faster circuits, smaller, more storage capacity, cheaper. We are concerned, it appears, with beauty (cf. *"user friendly software"*,) (intangible) affinity to actual (usually man–made) application domain phenomena, and with correctness expressible in mathematical logic and with reference to domain and requirements models.

Hence it may serve the software engineer, clients and users well, when, as we shall advocate, serious and time consuming effort is spent on creating beautiful documents.

4.1 Phase, Stage and Step Relations of Development Documents

For each (domain engineering, requirements engineering and software design) *phase;* for *stages* within these (as when the software engineer, for example, refines software architecture designs to program organisation designs); and for each *step* within stages (as when the software engineer, for example, refines a tail recursive procedure into a while loop) — for each such phase, or stage, or step — we create a full complement of documents. Although we at present, ie. here, do not dictate that the order in which we have implied (ie. listed) the phases, stages and steps above, is also the order in which they are temporally developed, we do expect software to be delivered with a reasonably complete development span of such sets of documents, and readable (forward and backward) wrt. the order implied.

Pairwise such two sets of adjoining step documents are to enjoy a number of relations. We expect, as will be shown in [Bjørner 2001a], certain domain facet descriptions to be (*staged*) extensions of other domain facet descriptions — where 'extension' is a technical concept. We expect that (*phased*) software design be correct wrt. some requirements. We expect that domain requirements be (*staged*) projections, extensions, instantiations and initialisations of domain descriptions. And we do expect (*stepwise*) development of algorithms and data structures from sorts or abstract models to be likewise correct. Etcetera.

We expect all these things, because we are the one who impose all these *"separation of concerns" phases, stages* and *steps.*

So we must pave the way for establishing, validating and verifying the many kinds of document relations that are implied. That way is dealt with in the next sections: Sections 4.2–4.4.

The specific decomposition of documents into the document parts outlined in those sections is determined pragmatically, not semantically nor syntactically.

4.2 Informative Document Parts

Informative documents (parts), serve to open for the development and/or reading (incl. maintenance) of a phase, a stage or a step of development. They focus on the *pragmatics* of the (development) problem at hand. There appears to be at least the following kinds of informative documents.

4.2.1 *Development Briefs.* A *development brief* is usually a, or the, very first *informative* and *pragmatic* ("letter of intent") document of any software development project. It states, in very general terms, what is to be done, why, by whom, for whom, when, where, by what time, at what approximate cost, etcetera.

Example: "A software system, for the partial support of a number of logistics information keeping and processing needs, is to be developed. Deployment of the desired software system shall make client y more competitive by speeding up and shortening, significantly, its response, respectively service times vis–a–vis customers. The software system is to be developed by software house x for client y, starting on data d and to be finished by date d', at the b branch office of x, and to be delivered within a total cost of \$ dollars." ... ▪

Development briefs may just deal with the development of a domain description, or a requirements description, or a software design, or with so–called "executable" program code, or with any composition of two or more ("adjacent") of these. We shall not deal further with development briefs — other than advising that development briefs be aware of, and hence be formulated around, the triptych concept of software engineering (domains, requirements, designs) and the corresponding, relevant development principles, techniques and tools.

Principle: We continue the example above: ... *The software development is to be decomposed into domain specification development, requirements definition development, and software design. Each of these development phases, and stages within these — as well as steps within stages — are to be properly related according to paradigmatic principles and techniques of validation and verifiability* ▪

4.2.2 *Contracts.* A *contract* is usually a in *informative,* as well as a legally binding *pragmatic document,* which, when properly worded and signed, can play a significant rôle in a court of law should client and developer disagree, for example, on whether delivered software has been delivered on time, and/or at cost, and/or satisfying certain expectations, etc.

The contract may call for the development and delivery of (i) a domain description, or the development and delivery of (ii) a requirements based on an existing domain description, or for the development and delivery of (iii) software based on an existing requirements descriptions. In either case the contract should refer to "best practices" for the respective kind of development task, and, in case of (ii—iii), to respective existing documents.

Example: "Developer x *(address so–and–so, legal registration number so–and–so, etc.) shall develop for client* y *(address so–and–so, legal registration number so–and–so, etc.) a software system that adheres, as described below, to the attached requirements document* d*. Delivery date is to be* dmy*. Developer* x *may make use up to* my *man years of the following kind client* y *staff: ... and scheduled as follows: .... Development cost is to be* m *US Dollars. The following acceptance criteria will be applied: .... Failure to pass the acceptance test, or failure to deliver on time or at cost will be handled as follows: ...."* — with more text to follow.∎

But it is not just between software houses and their clients, but also between in–house developers, that contracts are needed. When one group of in–house developers deliver, say a domain description, then another such group is expected to further develop, say a requirements definition, from the domain model, based on an internal, in–house contract that relates the two groups.

We shall not deal further with contracts — other than advising that contracts, in general, whether external or internal, be formulated around the triptych concept of software engineering and the corresponding, relevant development phase, stage and step principles, techniques and tools.

4.2.3 *Synopses.* A *synopsis* is a short, *informative document* that refers to both development briefs and contracts, and which focuses on setting the *scope* and *span* for the desired *universe of discourse*. The *scope* statements provide the general setting for the development. The *span* statements provide the specific, more narrow, setting. If that *universe of discourse* is a *domain*, then the *synopsis* shall provide general and specific settings for all major *concepts* of that *domain*. Similarly for the requirements and the software design universes of discourse. The first universe is "synopses" solely in terms of domain terms, the last universe primarily in terms of software (and, in general, also hardware) technological terms with reference to the requirements documents, and the requirements universe is expressed both in domain and software[3] terms.

Example: A Logistics System Synopsis: *"The logistics computing system to be developed shall deal with all relevant customer relations (inquiry, dispatch and receipt of freight, freight tracing, schedules and fees, etc.), with all relevant 'logistics-firm'–to–'transport–firm' relations (truck, rail, ship and aircraft conveyor route nets, time tables (schedules), fees, reservation, cargo delivery, cargo receival, actual position of conveyors, etc.), and with all relevant 'logistics–firm'–to–'freight–transfer–hub' relations (actual conveyor arrival and departure times, cargo loading, unloading, temporary storage conditions, etc.)."*∎

We shall not deal further with synopses — other than pointing out that synopses are not descriptions. They are pragmatic documents that merely "list" a number of important universe of discourse concept terms.

4.2.4 *Discussion.* The point of bringing the subsections above, and especially of bringing the, all be they very tiny, examples, is indeed to illustrate what kind of *informative documents* software engineering is dealing with, their use of language, of terminology, etc. It is especially their relation to *descriptive documents* (or *document parts*) that is of interest to us.

4.3 Descriptive Document Parts

Descriptive document parts serve to tell "the real story": *What is it all about,* the *semantics* and the *syntax. Indicatively* of the domain ideas (*"What is it ?"*), *putatively* of the requirements concepts (*"Which are they, what is to be implemented ?"*), and *imperatively* of the software facilities (*"How does it work ?"*).

The next section (Section 6) will deal specifically with some of the main principles and techniques related to the above questions. The below subsections can be read now, or after having read Section 6.

4.3.1 *Rough Sketch Document Parts.* The purpose of *rough sketching* is to "force" forward the identification and formation of concepts. By a *rough sketch* we understand a tentative, possibly ill– and/or unstructured (primarily) *descriptive* text that primarily aims at describing pertinent phenomena, "things" of the relevant *universe of discourse* (*domain, requirements* or *software design*). *Rough sketching* is like the ubiquitous "back–of–an–envelope–sketching–rough–ideas". A *rough sketch* may not (ever) be(come) a consistent nor a complete, and certainly never a polished document (part). It serves only as a means for arriving at a *narrative* and a *terminology.*

Example: Patient Medical Journals: *"Patients visit physicians, pharmacies, hospitals, rehabilitation centres, clinical test laboratories, health insurance providers, etc. In any encounter with these health–care professionals and institutions the patient's medical journal is either created, inspected or augmented. Physicians interview, analyse, diagnose and treat the patient, and observe reactions (responses) to treatments. Interview questions and answers, analysis types and values (results), diagnoses, treatments and reactions to these are all (or so it is intended) noted down in the patient medical journal. For each separable case of illness a separate patient medical record is established, with all medical records of a patient constituting that patient's medical journal."* ▪

Rough sketches are normally not part of developer–to–client deliverables. After termination of the specific contract in connection with which they were first established, they are usually filed away and maybe inspected sometime in future.

4.3.2 *Terminology Document Parts.* A *rough sketch* serves, as was mentioned, to identify all the concepts, all the phenomena, the "things" (entities, functions, events, behaviours, etc.) that are characteristic of the universe of discourse.

We record such concepts, etc., in a *terminology.* By a *terminology* we understand a structured set of terminological entries, possibly sorted by **key-worded terms,** and with a text that may highlight such other terms as are also (to be) defined in the terminology.

Example: Patient Medical Record (PMR):

—*A* ***PMR*** *contains administrative information and of a set (a list) of uniquely time–and–'author' "stamped"* *entries.*

—*A* ***PMR entry*** *is either an* *interview*, *an* *analysis*, *a* *diagnostics*, *a* *treatment*, *or a* *response entry*.

—*An* ***administrative information PMR entry*** *consists of . . .*

—*An* ***interview PMR entry*** *consists of . . .*

—***Initialising*** *an interview entry means . . .*

—***Augmenting*** *an interview entry means . . .*

—***Evaluating*** *an interview entry means . . .*

Etcetera. That is: Further terminological entries[9] dealing with analysis, diagnostics, treatment (plans), and response PMR entries, as well as on operations on these∎

Example: Patient Medical Journal (PMJ):

—*A* ***PMJ*** *is a set of PMRs relating to one and the same patient.*

—*A* ***PMJ search*** *is a projection of all those PMRs which satisfy a given search criterion.*

—*A* ***PMJ search criterion*** *is . . .*

—*A* ***PMJ projection*** *is a set (ie. a structured list) of PMRs such that . . .*

Etcetera∎

Terminologies should normally be part of 'developer'–to–'client' deliverables.

4.3.3 *Narrative Document Parts.* By a *narrative* we understand an informal description of some aspects of some universe of discourse.

Example: A logistics system consists of a number of senders and receivers (of freight), a number of logistics firms, a number of transport firms, a set of related transport nets, and a set of freight currently either being entered into the logistics system, or leaving this system, or en route from a sender to a receiver via one or more transport firms as handled, for each freight item, by a single logistics firm. A sender (i) may inquire, with a logistics firm, as to the cost and time and other conditions for the conveyance of a further specified item of freight; may receive a quote for such a conveyance — with the quote offering, at specified cost, and with a guaranteed delivery time and route, a possibly insured conveyance of an item of freight; (ii) may order such a conveyance, in return for which a bill–of–lading is issued; etc. A receiver . . . A logistics firm in turn . . . A bill–of–lading . . . ∎

The above exemplified narrative brings a very incomplete (. . .) excerpt of a real narrative. To construct pleasing narratives is hard. They are usually long. *Narratives* should normally be part of 'developer'–to–'client' deliverables.

We shall have more to say about narratives in this paper.

4.3.4 *Formal Specification Document Parts.* By a *formal specification* we understand a formalisation of a narrative — and, vice–versa, we can view a narrative as an explication of a formalisation.

Example:

$$
\begin{array}{l}
\textbf{type} \\
\quad \mathrm{PMJ} = \mathrm{Uid} \underset{m}{\overrightarrow{\;}} \mathrm{PMR} \\
\quad \mathrm{PMR} = \mathrm{Admin} \times (\mathrm{TiAid} \underset{m}{\overrightarrow{\;}} \mathrm{Entry}) \\
\quad \mathrm{Entry} = \mathrm{Invw} \mid \mathrm{Anal} \mid \mathrm{Diag} \mid \mathrm{Trea} \mid \mathrm{Resp}
\end{array}
$$

[9]Do not confuse PMR entries with terminology entries.

...
value
search: Criterion → PMJ → PMR-**set**
...

The above formalisation presents just a very tiny part of a formal model∎

Formal specifications should normally be part of 'developer'–to–'client' deliverables.

We shall have more to say about formalisation in this paper.

4.3.5 *Discussion.* The *rough sketch, terminology, narrative* and *formal specification document parts* are indeed parts of documents. That is: One can either, "ascetically" try "stick" with just one kind of document part (ie. document part type) for each separate document. Or one may try "merge" two, more or all of these document part types into one overall document. Too much separation can lead to a stilted and repetitive style of documentation which, when read (ie. studied) may lead to unnecessarily repetitive cross–referencing (whether turning pages of one or more separately bound paper documents, or through electronic hyper links). Too little separation, with no distinction as to which document type is being dealt with, may lead to confusion.

4.4 Analytic Document Parts

Validation is about getting the right product
Verification is about getting the product right
`Barry W. Boehm, 1981 [Boehm 1981]`

The main (1–2) aims & objectives of disciplined development of domain, requirements and software design documents, and of informally and formally relating phases, stages and steps of development (cf. Section 4.1) are those of validation and verification. (3) In addition an objective of formalising also domain descriptions is to start the development of theories of application domains. (4) Finally, when designing computing systems we often need ascertain various performance measures: In the domain as well as of desired software. All are efforts which, when properly conducted, lead to analytic documents. Issues (1–4) are dealt with, cursorily, in Sections 4.4.3, 4.4.4, 4.4.1, and 4.4.2, respectively.

Thus analytic documents parts are texts which are like meta–descriptions: They deal with properties of proper descriptions, they refer to such and are "about" such.

4.4.1 *Theory Formation Document Parts.* By *theory formation document parts* we understand collections of theorems (lemmas, propositions), auxiliary informal and formal descriptions, and theorem (etc.) proofs — where these theorems (etc.) refer to proper descriptions.

Example: Such as the concept of bill–of–lading, its designated transport, and the notion of "approach" have been defined one can show that freight items, when transported according to the prescriptions of the bill–of–lading, are progressing towards their destination in a monotonically "approaching" manner∎

The theorems (etc.) are usually expressed in some logic, one that is commensurate with the logic of the formal specification languages used in the descriptions, and the theorems usually do not rely on other than auxiliary and occasional extensions to already existing formalisations.

Very little, other than the possibility of their mere existance, can otherwise be said about theory formation document parts: This is so since we are in general dealing with any conceivable kind of theory being formed, and because it takes any conceivable ingenuity to create such theories.

4.4.2 *Ancillary Model Document Parts.* By an *ancillary model* we understand comprehensive informal and (especially) formal predicative descriptions that are not ("immediate") descriptions of a domain, requirements or software design universe of discourse, but of abstract meta–concepts. These meta–concepts are typically concerned with behavioural properties of the domain, requirements or software design universe of discourse. The concepts are here termed meta–concepts in that they are not immediately observable by universe of discourse stake–holders. Further, the purpose of the ancillary model is to use the model for predictive purposes — so proofs or abstract interpretations (ie. some form of computations) of such predictions (theorems, etc.) are part of the ancillary model.

Example: Such as the design for the handling of disk storage accesses have been defined one can show that the average disk access time will be statistically distributed as follows ... ▪

Ancillary models typically center around operations research models rather than computing science models. In a sense one can claim that the ancillary models are "orthogonal" to the universe of discourse descriptions to which their statements apply.

Very little, other than the possibility of their mere existance, can otherwise be said about ancillary model document parts: This is so since we are in general dealing with any conceivable kind of model being required, and because it takes any conceivable ingenuity to create such models.

4.4.3 *Validation Document Parts.* By *validation* we understand an informal process of establishing an identification between a description and the actual (or even vurtual, computing) world being described.

Examples: Usually informal domain and requirements descriptions are validated by stake–holders through ("walk–through") readings of informal document parts. Software may be tested by performing "test runs" with test data▪

4.4.4 *Verification Document Parts.* By *verification* we understand either a formal proof, or a substantial, trustworthy model checking effort, that convinces the reader, the client, that some claimed property of a universe of discourse description holds, or that a step, stage or phase of development is correct, ie. that a pair of "adjacent" descriptions stands in a desired relationship. Cf. Section 4.1.

Examples: (1) The given domain requirements form a proper projection of a given domain model. (2) The software architecture design if a correct implementation of the domain requirements. (3) The program organisation design is a correct implementation of the software architecture and the interface and machine requirements▪

4.4.5 *Discussion.* The structure and contents of analytic documents is far from a well–understood area of software engineering. Sure, much effort has gone into program correctness, but as this section intimates, there is much more to analytic documents than just proofs. Furthermore, the correctness and other property proofs that are being demonstrated are few, are wide apart, and, when applied to seemingly large scale, for example software systems, usually cover highly simplified situations. In other words: Much work need still be done for anyone to be able to say definitive words about the compostion of analytic documents — other than the utter need for doing more !

4.5 Discussion

Software engineering seems to be rather immature wrt. documentation when, as an engineering discipline, it is compared with for example nautical engineering: The development documentation that is developed in connection with ship building and the documents that follows delivered ships is, respectively are, subject to strinency criteria that have yet to find their way into most software houses development practices. But then, of course, ships can be insured; and such insurance facilitators as Lloyd's Register of Shipping and Norwegian Veritas monitor & control long standing, highly professional, ie. responsible and mature, development practices.

The present section has tried to shed some light, in the context of our evolving *"imagery"* of software engineering literacy, as to what may constitute similarly professional, ie. responsible and mature, software development documentation practices.

5. ON THE ONTOLOGICAL NATURE OF DESCRIPTION

TO BE WRITTEN

5.1 *"What is a Description ?"*

To us, in the context of computing, a *description* is a `text`, formal or informal, possibly adorned with some graphics (diagrams, graphs, or other), which *'by describing'* identifies something, a model. It may single out (i) a `thing`, an `entity`, in some domain, a `physical component` of a "world state", a `component` of a `computable state`, something that can eventually be given a *mathematical model;* that is, which singles out a `phenomenon` that can be manifested, ie. shown to exist, including to occur, physically or could be represented inside a computer. Or the `text` may single out (ii) a `concept` that abstracts such physical phenomena, or a `procedure`, a `task`, an `action` — carried out by humans or by mechanical or other machinery. Or the `text` identifies (iii) an `event`, a 'transition' from one "state of the world" to another; etcetera.

5.2 On the Philosophical Nature of 'Describing'

The above paragraph, containing the three "alternatives" (i–ii–iii), is our, albeit feeble, attempt to encircle, to circumscribe, to "ring in", to delineate, what is meant by a *description.* The problem basically is that one can never, "algorithmically", express the predicate which consistently and completely determines whether something is a description or not. It is, in its deeper essence, a problem that has occupied, and still occupies philosophers [Mellor and Oliver 1997]. So there is probably scant hope that we, as computing scientists, can succeed.

The (i-ii-iii) represents just one of possibly many ways of modelling, of understanding, ie. of describing certain universes of discourse. There are other approaches. The whole field of 'Conceptual Modelling' is ripe with fascinating papers on data modelling, function modelling, knowledge modelling.[10] In this paper we have chosen the `RAISE` Specification Language (`RSL`) as our means of expression.[11]

5.3 *"What can be Described"*

Since what we shall wish to describe is eventually to be related to something that shall exist inside computers we can *'Gordian knot'* easen our problem of *"What can be Described"* by simply relating the universe of discourse (to be described) to mathematics: If we can think of some way of expressing *"that which is to*

[10] `UML` [Jacobson et al. 1999; Booch et al. 1999b; Booch et al. 1999a], sets up one, fixed variety of modelling techniques from which to select. `VDM--SL` [Bjørner and Jones 1978; Bjørner and Jones 1982; Jones 1990; Fitzgerald and Larsen 1997] sets up another frame. `RAISE` [George et al. 1992; George et al. 1995], and `Z` [Spivey 1988; Ince 1988; Imperato 1991; Potter et al. 1991; Spivey 1992; McDermid and Whysall 1992; Hayes 1993; Wordsworth 1993; Diller 1994; Rann et al. 1994; Ratcliff 1994; Barden et al. 1994; Bottaci and Jones 1995; Bowen 1996; Potter et al. 1996; Woodcock and Davies 1996; Jacky 1997] set up frameworks similar to `VDM--SL` — with `RAISE` further providing schemes, classes and objects such that `RAISE` can be said to be object–oriented, as well as providing means for expressing currency according to the `CSP` paradigm [Hoare 1978; Hoare 1985; Roscoe 1997].

[11] We would have liked to also show examples that reflect the `Duration Calculi`, the `PetriNet` [Jensen 1985; Reissig 19], the `Statechart` [Harel 1987; Harel et al. 1987], and the algebraic semantics paradigms `CASL` [Mosses 1996; Mosses 1997; Mosses 1999; Bidoit et al. 1998].

be described" to some mathematical formula, including to some computer science concept, then perhaps we have a workable solution to our problem ?

In Section 5.1 we "stacked the card" in favour of the above resolution, not only by mentioning mathematics and computers, but by making sure that the three alternatives together corresponded to "standard" computer science concepts: (i) corresponds, roughly, at least that was the intention, to information (ie. data): Values, types and state components; (ii) similarly corresponds to functions over, behaviours around, such components; and (iv) corresponds to "spontaneous" state transitions, as caused, say by non–deterministically occurring inputs, etc.

5.4 Principle of Description

This will then be our *Principle:* When searching for things to be, or that can be described, select first those phenomena which can be made to correspond to some mathematical, including to some computing science concept. Describe those phenomena, informally (rough sketch, narrate, "terminologise") as well as formally (formal specification). In doing so the describer will be forced, yes even encouraged, to subsequently describe less obviously describable phenomena■

5.5 The *'Triptych'* of Descriptions

The rest of this section will take its departure point in Michael Jackson's *theory of descriptions* [Jackson 1995b]. That theory hinges on three meta concepts: *Designations, definitions* and *refutable assertions.* They relate as follows: (1) First, in describing, one wishes to "point to", to *designate* "things" (phenomena, artifacts, etc.) that somehow, physically or conceptually 'exists', and *describe* those. (2) Then, in summarising classes (collections, aggregations) of similarly designateable phenomena, or in "lifting" such to concepts they somehow represent, one *defines* these concepts. (3) Finally, since such *defined* notions are, by nature, not real, not physically manifestable, one usually expresses possibly *refutable assertions* about them. These assertions serve to endow the defined concepts with further qualities, with attributes that may, or may not hold, but which initially, as we define them, are believed to hold. If, later, it turns out that the assertions do not hold, then we have to abandon the assertions and come up with new ("and improved") ones !

Principle: Definitions are not interesting unless they can be endowed with possibly refutable assertions. *"One has not described anything of interest unless it may potentially be refuted"*■

5.6 On Concepts

$\boxed{\text{TO BE WRITTEN}}$

5.7 On Concept Formation

$\boxed{\text{TO BE WRITTEN}}$

5.8 Discussion

$\boxed{\text{TO BE WRITTEN}}$

6. DESCRIPTIONS

The first subsections of this section, although they may use the term 'universe of discourse', are formulated to apply first to domains (to be described). However, they apply, inter alia, also, with minor textual adjustments, to such universes of discourse as requirements and software design.

6.1 Designations

In this and the following two sections we shall implicitly be quoting from [Jackson 1995b].

6.1.1 *Designation Descriptions.* By a *designation description* (for short: *designation*) we mean a triple: A *designation term*, a *designation recognition rule*, and an *identification*. A *designation term* is a simple name. A *designation recognition rule* is a text which purports to *designate* something. The *designation identification* "links" the *designation recognition rule* to that "something" !

To "create" a *designation* we thus set down two items: a *designation description*, and a *designation identification*. The former, the *designation description*, is usually of the form:

—Designated term: `dt`.

—Recognition rule: `dt`*s satisfy the following informally stated properties: ...* — where we leave all terms undefined, ie. assume them well-known !

That is: *Recognition rules* are expressed only in terms that are otherwise well understood, ie. part of the folklore. If a "recognition rule" thus contains a term that is elsewhere *defined*, then it is not a "recognition rule" but becomes a *definition*.

Example:

—Designated term: `rail_unit`, or just `U`.

—Recognition rule: *A* `rail_unit` *is a composition of an even number of parallel positioned 'rails' (long, narrow, profiled iron bars) separated such that one can always identify pairs of rails of the composition that are at a specified distance (the 'rail gauge'), and otherwise held together by a set of 'sleepers'.*

When we write `rail_unit` we mean the extensional meaning (the type) of that term∎

It is not always clear from [Jackson 1995b] what is meant: The type or a specific value of that type.

Example:

—Designated term: `linear_rail_unit`, or just `U`, for which a predicate, to be assumed, holds: `is_linear_rail_unit(u)` for all `u` in the extension `U`, ie. `u:U`.

—Recognition rule: *A* `linear_rail_unit` *is a pair of parallel positioned 'rails' (long, narrow, profiled iron bars) separated at a specified distance (the 'rail gauge') and held together by a set of 'sleepers'.* (The predicate `is_linear_rail_unit(u)` "arises" from the recognition rule.)

The latter example is a specialisation of the former∎

A *designation identification* is usually of the form: "That 'thing' (u) there is a U. So is that 'thing' (u') over there !" — thus pointing out a *designation set,* instances of values that together become part of a type.

6.1.2 *Some Observations.* In general a *designation description* can be formalised; but one cannot formalise the *identification* — as it relates a formal world to an inherently informal world.

As illustrated in the two previous examples, the latter one of which was a specialisation of the former, a *designation* may extensionally denote a class (of things) which can be subdivided into subclasses.

Example:

—Designated term: `rail_unit`.
—Recognition rule: — as for the above, previous `rail_unit` example.

—Designated term: `linear_rail_unit`.
—Recognition rule: — as for the above, previous `linear_rail_unit` example, and additionally: *Thus a* `linear_rail_unit` *has two "ends" and a single (two–way) link between these "ends". Any one other* `rail_unit` *may be* `connected` *to either of these "ends" (and if so, then the "end" is called a* `connectors`*).*

—Designated term: `switchable_rail_unit`.
—Recognition rule: — as for `rail_unit` just above, and additionally: *A* `switchable_rail_unit` *has three* `connectors` *which "define" (allow, permit) two links through the* `switchable_rail_unit`*.*

—Designated term: `simple_crossover_rail_unit`.
—Recognition rule: — as for `rail_unit` just above, and additionally: *A* `simple_crossover_rail_unit` *has four* `connectors` *which "define" (allow, permit) two links through the* `simple_crossover_rail_unit`*.*

—Designated term: `switchable_crossover_rail_unit`.
—Recognition rule: — as for `rail_unit` just above, and additionally: *A* `switchable_crossover_rail_unit` *has four* `connectors` *which, together with the switching ability of the unit, "define" (allow, permit) four links through the* `switchable_crossover_rail_unit`*.*

—Designated term: `connector`.
—Recognition rule: *A* `connector` *is that which allows two* `rail_units` *to be connected, "end-to-end".*

Further recognition rules deals with `connectors` and `rail_units`. [2] *A* `connector` *is any "end" of a* `rail_unit` *to which other* `rail_units` *may be 'connected'. Any one* `connector` *is shared by at most two* `rail_units`*. A* `rail_unit` *is mutually exclusive either a* `linear_rail_unit` *or a* `switchable_rail_unit` *or a* `simple_crossover_rail_unit` *or a* `switchable_crossover_rail_unit`*.* See axiom [2,3] in a formalisation of the above in Section 6.1.3.∎

As we shall see later, in Section 6.3, doubt may be raised as to whether some of the above recognition rule text parts express assertions. (Designations state facts, not assertions.) For example 'has two "ends" ' (or three, or four). These parts are part of recognition rules, but what about ([2]): 'Any one connector is shared by at most two rail units' ? Or the part right after it: '... mutually exclusive ...' ? For the last (the mutual exclusion) we can claim it to be a fact, hence part of a recognition rule. For the 'at most two', the case is a bit more complicated. And then, when we formalise the whole thing, as we shall see below, and when such a formalisation is compared to that of a refutable assertion, then we shall see that, formally speaking, the difference is almost invisible.

Thus we must be prepared for the eventuality that the pragmatics of making a distinction between designations, definitions and refutable assertions can not be carried visibly over into formal models of the things being designated or defined, or for which assertions are expressed.

6.1.3 *Formalisation.* Now which of the above alternative ways of designations are we going to formalise ? That question is dealt with now.

Typically we formalise a *designation description* as shown in the next example. It allows for the general case of several alternatively expressible designations.

We choose to introduce just one type (ie. sort), U, for rail units, and to let (recognition rule oriented) observer functions and suitable axioms help separate rail units into its partition of its more specialised rail units.
Example:

```
type
   U, C, L
value
   obs_Cs: U → C-set
   obs_Ls: U → L-set
   is_linear, is_switch, is_simple_cross, is_switch_cross: U → Bool
axiom
   [1] ∀ u:U, ∃ c,c′,c″,c‴:C •
           card{c,c′,c″,c‴}=4 ∧ obs_Cs(u)⊆{c,c′,c″,c‴} ⇒
              is_linear(u) ⇒ obs_Cs(u)⊆{c,c′} ∧ card obs_Ls(u)=1 ∨
              is_switch(u) ⇒ obs_Cs(u)={c,c′,c″} ∧ card obs_Ls(u)=2 ∨
              is_simple_cross(u) ⇒ obs_Cs(u)={c,c′.c″,c‴} ∧ card obs_Ls(u)=2 ∨
              is_switch_cross(u) ⇒ obs_Cs(u)={c,c′,c″,c‴} ∧ card obs_Ls(u)=4,
   [2] ∀ c:C • card{ u | u:U • c ∈ obs_Cs(u) } ≤ 2
   [3] let lus = {| u:U • is_linear(u) |},
           sus = {| u:U • is_switch(u) |},
           cus = {| u:U • is_simple_cross(u) |},
           scus = {| u:U • is_switch_cross(u) |} in
       lus ∩ sus = {} ∧ lus ∩ cus = {} ∧ lus ∩ scus = {} ∧
       sus ∩ cus = {} ∧ sus ∩ scus = {} ∧ cus ∩ scus = {} end
```

U denotes the possibly infinite set of all *designations* satisfying the rail_unit *recognition rule*. C stand for the possibly infinite set of all *designations* satisfying the connector *recognition rule*. L denotes the concept of link. The predicates is_linear, is_switch, is_simple_cross, and is_switch_cross further constrain the rail_unit *recognition rule* — as indicated by the **axioms**.∎

6.1.4 *Observer Functions and Identification.* The *observer functions*, eg. obs_Cs, and *obs_Ls*, are the closest counterpart to the undefined terms of *recognition rules* that we have. These *observer functions* cannot be defined. They are postulated. They "arise" as the result of *identification*. Given an actual *universe of discourse* — to which the above *designations* are said to apply — one can now "define" these *observer functions* by "walking out and into" the *universe of discourse* and providing the *identification*.

To get a better grasp of possible relationships between recognition rules and observer functions let us illustrate yet an example.

Example: A railway net consists of [one or more] lines and [two or more] stations. A line is a linear sequence of one or more linear rail units. A station is any composition (connection) of rail units. [A line connects exactly two distinct stations.] The above (excluding the [bracketed] parts) may be claimed to be a definition, and we shall soon turn to a treatment of definitions. But we claim that each of the ([un–bracketed] parts of the) above notions can be designated. The *slanted texts* above may not look like *recognition rules*, but they are ! One can indeed "walk out, into" the railway system domain and, with a sweeping hand, express: *That "thing" there is a linear rail unit. That one, next to it, is likewise. That particular sequence of those two linear units, I just pointed to, form (part of) a line. This "thing" here is a rail unit of a station. It is a crossover. Here is the connector that separates a line from a station. No rail unit is both a rail unit of a line and of a station, or of two otherwise distinct lines or stations.* Etcetera.

type
  N, L, S, U, C, L
value
  obs_Ls: $N \rightarrow$ L-**set**, obs_Ss: $N \rightarrow$ S-**set**, obs_Us: $(N|L|S) \rightarrow$ U-**set**
axiom
  $\forall$ n:N, l,l′:L, s,s′:S •
    card obs_Ls(n) $\geq 1 \wedge$ **card** obs_Ss(n) $\geq 2 \wedge$
    {l,l′} $\subseteq$ obs_Ls(n) $\wedge$ {s,s′} $\subseteq$ obs_Ss(n) $\Rightarrow$
      l$\neq$l′ $\Rightarrow$ obs_Us(l) $\cap$ obs_Us(l′) = {} $\wedge$
      s$\neq$s′ $\Rightarrow$ obs_Us(s) $\cap$ obs_Us(s′) = {} $\wedge$
      obs_Us(l) $\cap$ obs_Us(s) = {} $\wedge$
      exactly_two_distinct_stations(l,obs_Ss(n))

where we leave out the definition of exactly_two_distinct_stations∎

6.1.5 *Designations: Mathematical & Computing Entities.* From a formal point of view, ie. from the points of views of mathematics and computer & computing science, which are the kinds of designations ? Which kinds of mathematical entities are they ? Or, in the jargon of computing: Which types of computer & computing science entities are they ?

Mathematical entities:. When viewed mathematically, are they scalars, like numbers (integers, reals (or complex), rationals, transcendental), or truth values, etc. Or are they composite, like sets, Cartesians, lists, maps or functions (or algebras, etc.) ? And if the latter, then which are their component elements.

Computing entities:. When viewed computer & computing science–wise, are they values (of, for example the above mentioned mathematical kind (where models are like algebras)). Or are they types of these (ie. types as we know them in computer science: Simple lattices, Scott domains, or otherwise) ? Or are they events — whatever they are ? If lists, then what do these lists model: Behaviours (of processes, in terms of traces of actions and/or events, etc.), or other ? In any case, when we model computing entities (values, types, events, (process) behaviours, semantic algebras), then we model them in terms of the above mentioned mathematical entities.

Awareness:. We do not intend to give a full answer to the question: Which kind (type) of formal entitites can designations be modelled by ? We only advice that the practising, professional software engineer be reasonably well–versed in these matters: Modelling designations formally in terms of mathematical entities – perhaps couched in the computing jargon of for example values, types, events, (process) behaviours, semantic algebras, or other.

Some guidelines:. In the following we shall try to stay clear, if at all scientifically possible, of the deeper problems of conceptual modelling, viz.: [Ganter and Wille 1999; Sowa 1999], as currently studied in computer & computing science, and in various philosophical discourses, viz.: [Mellor and Oliver 1997]. But we must — since it cannot be avoided [if we are at all to cover some ground] — postulate some possibilities of concept modelling (since that is what it, in essence is all about). We do so in order to identify some designation (etc.) principles and techniques,

• • •

Some *designations* are specific, “one of a kind” things (“that rail unit there”), ie. specific values; or they are specific events, or specific behaviours (etc.). Or *designations* are types or models (ie. algebras) over these.

Some designations are those of components of contexts and states. Contexts are entities whose properties — whose attributes, whose values — remain statically fixed over time. States are such whose values change with time — are dynamic. Examples of contexts are: *Road* and *rail nets* (when viewed topologically and over not too large time periods), similarly for airline *time tables.* Examples of states are: *Road,* rail and air traffic, hence *trucks,* trains and aircrafts, and the *hubs* where they meet and passenger embarkation and debarkation, or *freight receipt, transfer* and *delivery* can take place, are state components. A *bill–of–lading* is somewhere “in-between”: The *route* along which a *freight item* should be conveyed is, we assume, static, but the *bill–of–lading* is (probably) marked (“updated”) to show the (believed) current (or last recorded) position of the *freight item* of which it is the *bill–of–lading.*

But, in any case, all these examples can be modelled as typed, usually composite values, fixed or variable. These values we consider inert: They do not change by themselves. Some external action has to change them.

Designations thus could, alternatively, be the, or a, specific action (active phenomenon, function, operation, task, procedure) of *inscribing a freight item for conveyance with a logistics firm, loading it onto the truck, unloading it,* etc. In this case we refer to the designations as function values.

Or designations could be the events of *truck departure* from, resp. *arrival at a* *hub.*

Or designations could be part or the entire (process) behaviour *from inscription to unloading* described in terms of the specific sequence of inert and dynamic phenomena. In this case we refer to the designations as behaviours (a certain kind of function values).

• • •

The above explication present just one kind of conceptual modelling approach. It is "slanted" towards CSP and RAISE [Roscoe 1997; George et al. 1995]. So:

Principle: When setting out, on a first description, identify which conceptual modelling framework you intend to work within ∎

Techniques: Among the conceptual modelling frameworks that can be believably[12] offered are:

—VDM, Z [Fitzgerald and Larsen 1997; Woodcock and Davies 1996],

—RAISE [George et al. 1992; George et al. 1995],

—CafeOBJ,CASL [Futatsugi and Diaconescu 1998; Mosses 1999],

We list three groups: The first are solely model-oriented, the last is solely property—oriented (in the algebraic semantics style), and RAISE basically offers both styles. If neither of these are fully applicable, then be careful, very careful in choosing some "integrated formal specification" approach which takes a "little from this specification language", a "little from that other language", and so forth. If the underlying paradigms are compatible, then fine, else there will be problems in securing an integrated semantics [Araki et al. 1999] ∎

A "Large" Example. We illustrate the implications of the above by giving a formal definition in which we then identify various "designateable" quantities. The example expands on the *slanted text explications* given between the pair of • • •s above:

type
  F, B, T, R, L, H, ...
value
  h,t:**Int**
type
  HIdx = {| 1..h |}, TIdx = {| 1..t |}
channel
  { ht[i,j]:(F×B) • i:HIdx, j:TIdx }
value
  inscribe: FIdx × F → L → ... L × B
  load: F×B → j:TIdx × j:HIdx → **in**,**out** { ht[i,j] | i:HIdx } **Unit**
  truck: j:TIdx → T → **in**,**out** { ht[i,j] | i:HIdx } **Unit**
  truck(i)(tr) ≡ (...;truck(i)(unload(tr)(i,j)) **assert:** /∗ truck is at hub j ∗/) ⌈⌉ (...)
  unload: T → i:HIdx × j:TIdx → **out** { ht[i,j] | i:HIdx } **Unit**
  unload(tr)(i,j) ≡ **let** (f,b):(F×B)•is_in(j,b) **in** ht[j,i]!(f,b);remove(f,b)(tr) **end**
  is_in: j:HIdx × b:B → **Bool** /∗ j is mentioned in b ∗/

[12] By 'believably' we mean: Frameworks for which software engineering texts, manuals and tools are offered or can be offered.

remove: F×B → T → T
hub: i:HIdx → Hub → **in**,**out** { ht[i,j] | j:TIdx } **Unit**
hub(j)(hu) ≡ ⌷{ hub(j)(add(ht[i,j]?)(hu)) | i:TIdx } ⌈⌉ (...)
add: F×B → H → H

F, B, T, R, L, H, etc., are type names of sets of *freight items, bills–of–materials, truck configurations, roads, logistics firm configurations, hub configurations,* etc. — they stand for (1) configuration[13] components where we give their type since we are interested in any values of those types. The *logistics system* consists of *h hub*s (here truck yards), *t truck*s, and the ability (modelled using channels) of *truck*s to dock at any *hub.* The names inscribe, load, unload, etc., name (2) specific functions that effect the inscription, loading and unloading actions. The names remove, is_in and add name specific, auxiliary functions. *truck, hub, load* and *unload* are processes — ie. functions that specify behaviours — the first two are cyclic (never ending). The syntactic pair of composite constructs **ht[j,i]!(f,b)** and **ht[i,j]?**, specifies synchronisation and communication events (in CSP [Hoare 1978; Hoare 1985; Roscoe 1997]) The event concept is thus, using the specific formal description language used here, RSL [George et al. 1992; George et al. 1995], expressed in terms of channels, synchronised output/input and communication∎

We remind the reader that the above shows just an example of how designations can be modelled — in one (the RAISE/CSP) paradigm.

Formal Modelling. It is not the intention of the present section (Section 6.1.5) to suggest neither principles nor techniques for exactly how to formally model designations. Such principles and techniques are the subject of [Bjørner 2001g; Bjørner 2001a; Bjørner 2001j]. Instead it is the aim of the current section to point out the following principle:

Principle: In considering which "things" to designate, and, in a larger context, which "things" to describe, the describer, ie. the software engineer is well served in considering whether that which is to be described informally: Rough sketched, terminologised and narrated — whether that — can be given a formal model∎

Certainly, if it can, it can most likely be described, that is: Is something possibly worth describing.

6.1.6 *Discussion. Principle:* In building up designations — within the conceptual paradigm (ie. framework), say, of RAISE — one must decide whether one is trying to designate the type of all those "things" that are "similar" to a few designations, ie. satisfy their common recognition rule — as for *F, B, T, R. L,* and *H* above — or one is trying to define only a specific value (one particular "thing") — as for *inscribe, load, truck, unload, is_in, remove, hub* or *add* above∎

This distinction is not, we find, made sufficiently clear in [Jackson 1995b]. Probably because that book, possibly wisely so, and definitely explicitly so, avoids "committing" itself to any specific formal specification paradigm.

In addition to modelling designations as *type*s, *channel*s, and *value*s, one could, in the chosen conceptual framework of RAISE, think of other designation models, for

[13]By configuration components we mean such which, when represented inside computer "end" as data, context constants or state variables

example: *schemes*, *classes*, *objects*, and *variables*. Other conceptual frameworks would favour similar or quite dissimilar choices — and the general *designation* principles of [Jackson 1995b] apply to any such paradigm.

We have, so far, only covered one of the three aspects of *descriptions: Designations.* Two more remain to be covered: *Definitions* and *refutable assertions.* Yet we have been able to demonstrate, in the formal example immediately above, that just dealing with *designations* one can come a long way.

One may rightfully argue (i) whether all of what that last, the formal example showed was indeed a model only of *designateable* entities, and (ii) whether, as we shall soon see, introducing some *definitions* might result in a different, perhaps more easily read description. One may also argue (iii) whether some of the axioms are in fact *refutable assertions.* We will return to these issues in the closing 'Discussion' subsection of this section.

6.2 Explicit Definitions

First: Designations represent one form of definition. In any description all terms that are special to the *universe of discourse* being described must be *defined,* either through *designations* or by *explicit definitions.*

6.2.1 *Definitions in lieu of Designations: "The Narrow Bridge".* The example above contained only designateable quantities, or so we claimed. But that may not always be possible, or convenient.

Sometimes it is easier to *define* a `term` by giving it a *definition,* bur (notably) using already *defined* (including *designated*) `terms.`

Example: A railway line is a linear, acyclic, sequence of linear rail units, that is a sequence where adjacent element of the sequence are rail units that share exactly one connector. Here the terms linear rail unit and connector are already designated terms. And line is being defined in terms of those designations. Also formally:

$$
\begin{array}{l}
\textbf{type} \\
\quad \mathrm{U, C} \\
\quad \mathrm{L1'} = \mathrm{U}\text{-}\textbf{set} \\
\quad \mathrm{L1} = \{|\ \mathrm{us{:}L1'} \bullet \mathrm{wf\_L1(us)}\ |\} \\
\quad \mathrm{L2'} = \mathrm{L}^* \\
\quad \mathrm{L2} = \{|\ \mathrm{us{:}L2'} \bullet \mathrm{wf\_L2(us)}\ |\} \\
\textbf{value} \\
\quad \mathrm{obs\_Cs{:}\ U} \to \mathrm{C}\text{-}\textbf{set} \\
\quad \mathrm{wf\_L1{:}\ U}\text{-}\textbf{set} \to \textbf{Bool} \\
\quad \mathrm{wf\_L1(us)} \equiv \\
\qquad \exists\ \mathrm{ul{:}U^*} \bullet \textbf{len}\ \mathrm{ul} = \textbf{card}\ \mathrm{us} \wedge \textbf{elems}\ \mathrm{ul} = \mathrm{us} \wedge \mathrm{wf\_L2(ul)} \\
\quad \mathrm{wf\_L2{:}\ U^*} \to \textbf{Bool} \\
\qquad \forall\ \mathrm{u{:}U} \bullet \mathrm{u} \in \textbf{elems}\ \mathrm{ul} \Rightarrow \mathrm{is\_linear(u)} \wedge \\
\qquad \forall\ \mathrm{i{:}}\textbf{Nat} \bullet \{\mathrm{i,i{+}1}\} \subseteq \textbf{inds}\ \mathrm{ul} \Rightarrow \\
\qquad\quad \textbf{card}(\mathrm{obs\_Us(ul[\,i\,])} \cap \mathrm{obs\_Us(ul[\,i{+}1\,])}){=}1 \wedge \\
\qquad\quad \textbf{len}\ \mathrm{ul} > 1 \Rightarrow \mathrm{obs\_Us(}\textbf{hd}\ \mathrm{ul)} \cap \mathrm{obs\_Us(ul[\,}\textbf{len}\ \mathrm{ul\,])} = \{\}
\end{array}
$$

Recall that an earlier axiom which stated that for any connector there are at most two units sharing that connector. That axiom is assumed above.∎

Technique: Definitions must be expressed in terms of designated and/or defined terms, and, ultimately definitions must be based on designations∎

Principle: Develop alternative definitions∎

Here we developed L1 and L2.

Although one may claim that lines could be designations, since, in a sense, they are tangible, we have here defined the concept of line.

We follow the advice of Michael Jackson [Jackson 1995b] when we raise it to a principle:

Principle: Seek as few dsignations as possible: Enough to define all the other possibly designateable as well as all the desirable abstract, non–tangible concepts∎

6.2.2 *Definition of Abstract, Intangible Concepts.* We shall next define some abstract, intangible concepts.

Example: A rail unit defines a concept of path. A path (through a rail unit) represents the ability for a train to move along that 'path', through the rail unit.[14] (Notice that we do neither, at present, designate nor define what we mean by train nor move.) *We define (ie. we model) a path as a pair of connectors.* (Based on the definition of path we then proceed to define further intangible, ie. not physically manifest, concepts.) *With any rail unit we can associate a state: A possibly empty set of paths. And, finally, we can associate, with any rail unit its state space: The set of all the states that a rail unit may "be in" over its lifetime as a rail unit.*

type
  U, C, P = C×C
  Σ = P-**set**, Ω = Σ-**set**
axiom
  $\forall$ (c,c′):P • c≠c′
value
  obs_Σ: U → Σ, obs_Ω: U → Ω

In Section 6.3 we shall look at possible relations between designations and definitions of the railway system∎

6.2.3 *How Much, How Little to Define ?.* The question is now: How much to *define ?* For the *universe of discourses* of *requirements* and *software design* there is a 'utility' concept: We know what we aim at, so we *define* at least that; but why more than that ? For the *universe of discourses* of *application domains* we do not, "as a matter of principle", know what we are aiming at, so here we "go for more": As long as some interesting ideas are being *defined,* then why not ? It is only, we believe, in "roaming around", "experimenting" with *definitions,* and, later, with *refutable assertions,* that we may "discover" most interesting, most relevant, most "fitting" domain concepts. It is in this "exploration", and based on *definitions* that we can expect to build theories of specific domains.

Principle: In constructing *domain models,* ie. *descriptions* of *domains, designate* according to the *"narrow bridge"* principle, and then *define* as many abstract concepts as long as their *definition* (and those of attendant *refutable assertions* help)

[14] When we earlier spoke of a concept of link, that that could, in a sense, be seen as an abstract concept: Much like a pair of opposite direction paths.

"reveal" further concepts∎

A last example may illustrate the last point.

Example: (i) A route is a sequence of connected rail units. (ii) A route is an open route if the states of all rail units of the route are such that there are paths in those states that also connect (in on direction or another). (iii) Given a unit within a station define as reachable from the station unit all those lines that are incident upon and (thus) emanate from that station, and for which there is a route between the unit and the line. (iv) Given any two rail units, an 'origin' and a possible 'destination', of a railway network define a more general notion of reachability, one that is satisfied if there are open routes from the 'origin' to the 'destination'. And so forth. We do not presently show a formalisation of the above, but refer to [Bjørner 2001i]. The present example aimed at illustrating the definition of a number of (viz.: four) concepts without a–priori making sure that these concepts will serve a useful purpose∎

6.2.4 *Discussion.* Choosing an appropriate balance between a reasonably small set of designations and an appropriate (large, "rich") set of definitions — choosing such a balance — is an art ![15] No definite advice can be given, other than perhaps the ones given above, which when followed, helps ensure an "appropriate balance".

We have now covered two of the informal and formal description principles and their formalisation techniques. From the examples so far we can already now conclude that one cannot see from the formulas whether a description formalises a designation or a definition !

That is: The *pragmatics* of distinguishing between designation and definitions is lost in the formulas ! Thus it is important with some *syntax* to head of the informal and formal texts in order to alert the reader to which is what !

6.3 Refutable Assertions

6.3.1 *Assertions Related to Designations and Definitions.* In the example above, where we described railway nets of lines and stations, we put in some [bracketed sentence parts]. Those parts did not designate manifest "things". They expressed suitable, desirable or other relations of designations (and, later, of definitions). We shall refer to such constraints as refutable assertions.

Examples: That *a line connects exactly two distinct stations* is one such example of a refutable assertion. It implied that there had to be *at least two stations in a railway net* — another refutable assertion; and *at least one line* — yet another refutable assertion. We asserted elsewhere that *every connector is shared by at most two (otherwise distinct) rail units;* that *linear rail units have two connector and define one link;* that *switchable rail units have three connectors (and "define" two links);* etc. These sentence parts can all be considered refutable assertions∎

Technique: Refutable assertions must be expressed in terms of either designated or defined terms, or both kind of terms∎

Designations, in a sense, are facts. And facts cannot be refuted. Definitions are definitions, and as such must be accepted. But definitions may see general and may thus need to be characterised ("tied down") through some form of constraining

[15] Of course: The hedge 'appropriate balance' justifies the property 'is an art'.

assertions.

Example: We continue the rail unit path, state and state space example given earlier. *The paths of a rail unit, ie. the paths of any of its states, must mention only connectors of that unit.*

axiom

$\forall$ u:U,c:C,σ:Σ •
  obs_Σ(u) $\in$ obs_Ω(u) $\wedge$
  $\sigma \in$ obs_Ω(u) $\Rightarrow \forall$ (c,c$'$):C$\times$C • (c,c$'$) $\in \sigma \Rightarrow$ {c,c$'$}$\subseteq$obs_Cs(u)

The above assumes an earlier axiom expressing the fact that the connectors of a path are always distinct ∎

This is a refutable assertion: One might think of cases where there could be connectors of a path in some unit's state that were not connectors of that unit, or could one ? We basically will not know till someone one day shows us an understanding of a railway net that "favours" such an interpretation !

6.3.2 *Analysis.* What was claimed, above, as examples of refutable assertions certainly were assertions. Whether they will be refuted remains to be seen, but they are potentially refutable. Take the example of *any connector* being "constrained" *to be shared by at most two rail units.* If one was to show, in future, that there were indeed connectors that were shared by three (or more) rail units, then we had indeed refuted the assertion. But is it likely ?

Example: A station is connected to at least one line ∎

Refuting this assertion would amount to allowing railway nets with completely isolated stations. Is that likely ?

Example: For any station, there is at least one potentially open route from, and at least one such route to, some other station of the net ∎

If that assertion is refuted then there will be at least one station from, or into which one cannot "travel" ! Is that likely ?

6.3.3 *"Dangling" Assertions.* It is much too easy to write down texts that "weave a web" of pseudo–designations, pseudo–definitions and pseudo–assertions.

Example: Suppose the following text was all we had: *A bill–of–lading lists the transport hubs where the referenced freight item first loaded unto a conveyor, is being transferred from one conveyour to a next, and is finally unloaded. A transport according to a bill–of–lading does not visit a hub more than at most once.* Then what were we to mean ?

Suppose further that we went straight ahead and "formalised" the above *slanted text:*

type
  Hub, Hn, F, Fn, Con
  BoL$'$ = (F $\times$ (Hn $\times$ Hn* $\times$ Hn))
  BoL = {| b:BoL$'$ • wf_BoL(b) |}
  Transport = F $\times$ (Con $\times$ Hub)*
value
  wf_BoL: BoL$'$ $\rightarrow$ **Bool**
  wf_BoL(f,(o,hl,d)) $\equiv$ **card**({o,d} $\cup$ **elems** bol)=2+**len** bol

obs_Fn: $F \rightarrow Fn$
load,unload: $F \times BoL \rightarrow Con \rightarrow Con$
transfer: $F \times BoL \rightarrow (Con \times Con) \rightarrow (Con \times Con)$
M: $BoL \rightarrow$ Transport-**infset**

What is the problem ? The problem is that we have taken the liberty of mentioning such possibly designateable phenomena as *Hub, H[ub] n[ame], F[reight], F[reight] n[ame], Con[veyor], load, unload* and *transfer* and given signature to the function values without having attempted the slightest bit of designations ! We have likewise defined a *M*eaning function which defines the meaning of a bill–of–lading as a possibly infinite set of defined *Transport*s.

The assertion *does not visit a hub more than at most once* is thus pretty meaningless.∎

Technique: Make sure all terms of an assertion are either designated, defined or otherwise bound in the assertion.∎

Example: With respect to the previous example we should thus also have properly sub-typed the *Transport* type, etc.∎

6.3.4 *Discussion.* Assertions are like axioms: Self evident truths — until refuted by observations made in the referenced universe of discourse. Assertions are thus not facts, nor are they theorems. Theorems are predicate expressions that can be proven to follow from designations, definitions and (the axioms of) refutable assertions. Theorems (lemmas, propositions) together with designations, definitions, the axioms of assertions and other theorems (lemmas, propositions) form a theory of the described universe of discourse.

6.4 Discussion

We have related the description principle concepts of designations, definitions and refutable assertions to (albeit sketchy examples of) formal specifications. We have, most recently above, seen how formal specifications, cf. the latest formalisation example above, without proper, necessarily informal descriptions that adhere to Jackson's principles, can too easily "drug" us into believing that the model identification principle (cf. Section 2.4) can be skipped.

We ourselves readily admit to have far too often, for anybody's good, fallen into that trap !

7. CONCLUSION

We have *woven a "story"* of *semiotics, documents* and *descriptions* — the latter presenting *models,* the four concepts listed in the title of this paper. We have focused on those aspects of these four concepts that relate to what we call the concept of *software engineering literacy:* (1) An ability to clearly know and announce which one or more of many different rôles[16] a model is to serve. (2) An ability to express clearly the pragmatics, the semantics and possible syntactic concerns of the artifact being developed. (3) An ability to structure the documentation into informative, descriptive and analytical parts, and to substructure these parts into design briefs, contracts and synopses; into rough sketches, terminologies, narratives and formal specifications; and into theory formation, validation and verification parts. And (4) an ability to describe the chosen universe of discourse: To designate, to define and to assert.

We may have given a few abstract and formal models, but those examples were only brought in order to show that it could be done: That the previous paragraph's may concepts lead to formalisable model parts. The formal examples were not shown in order to reveal abstraction (ie. formal specification) principles or techniques.

7.1 What has be Achieved ?

We believe that we have achieved:

—To point out that there are many linguistic, including model consideration issues that are to more–or–less all to be present in any one software development phase, stage and step.

—To link these: Models, semiotics, documents and descriptions into a meaningful whole. That is, to show that they are all tightly inter–related — and more inter––related, seen from the literacy, the linguistics point of view, that these four clusters of issues (models, semiotics, documents and descriptions) are to abstract principles and techniques (which we have covered in a likewise comprehensive manner in another paper [Bjørner 2001g]).

—To enunciate a number of development principles and techniques.

—To show that there is much more, also more than we have shown, to Michael Jacksons 'theory of descriptions' as covered in [Jackson 1995b]. Our main observations have, on this matter, been the following:

  —The ideas that descriptions are either indicative (as for domain), putative (as for requirements), or imperative (as for software design, especially program code), are obviously fine for informal descriptions. But formal descriptions presently lack modal operators to express these modalities.

  —Jackson deliberately and studiously avoids formalism other than an occasional use of simple first order predicate logic. But, for scalable and trustworthy, manageable software developments, we mist today meta–require that domain, requirements and (upper level) software design documents also be formally

[16]Iconic, analogic, analytic; prescriptive, descriptive; extensional, intentional, understanding, inspirational, awareness, educational, training, and/or implementational

expressed. And here [Jackson 1995b]'s ideas of designations, definitions and refutable assertions give little guidance on how to formalise.

—The ideas of designations, definitions and refutable assertions, are obviously fine for informal descriptions. But formal descriptions presently (and similar to above) lack modal operators to express these modalities.

7.2 A Critique

Which are the parts of this paper with which we are less happy (than with other parts) ?

—Since pragmatics is "all that which is not semantics and syntax" (sic !), the concept of pragmatics seems to appear less well–delineated than those of semantics and syntax.
This probably has to be so, but we could wish a more terse delineation of pragmatics. We could, as part of such an analysis of the concept of pragmatics, to see it analysed into more or less "orthogonal" issues, such as was partially attempted.

—Semantic models form a wide spectrum, and we have just covered some: (i) Models, denotations and semantic (types and) values; (ii) static and dynamic semantics, and (iii) abstract interpretation.
We are not quite happy with the distinction static versus dynamic semantics, but it is an accepted distinction. Yet, as for Cousot's cited work on abstract interpretation, the spectrum is more "smooth" than just its two "end points" (cf.: static and dynamic).

—We have not had the space to illustrate the concept of configurations, compositions of "state"–like entities, from static, contextual (environment), to dynamic, storage components — such as we know then from programming language semantic models (where we further operate with such entities as continuations).
In our [Bjørner 2001g] we shall have more to say on configurations: contexts and states. We refer to standard text books on programming language semantics for proper treatments of environment, storage and continuation issues [Schmidt 1986a; Schmidt 1986b; de Bakker 1995; Gunther 1992; Mosses 1992; Winskel 1993; Nielson and Nielson 1992].

7.3 The Main Message

The main message of this paper has been to advocate a concept of software engineering literacy based on semiotics, documents and descriptions; to show the elements of this literacy; and to enunciate some attendant software development principles and techniques.

We could wish for further studies of this and related concepts of software engineering literacy.

ACKNOWLEDGMENTS

The author gratefully acknowledges the inspiration drawn from the works of Michael Jackson, and is otherwise happy to similarly acknowledge the insistance of the IFIP Working Group 2.3 (on Programming Methodology) that we primarily focus on proper methods for software development. The current author is especially heartened by what he considers a constructive attitude of IFIP WG 2.3, one that helps us not jumping on too fashionable band wagons, but keep insisting on simple foundations not "supported" by syntactic diagrams with no semantics.

REFERENCES

Abrial, J.-R. 1996. *The B Book.* Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England.

Araki, K., Galloway, A., and Taguchi, K. Eds. 1999. *IFM'99: Integrated Formal Methods* (London. England, June 1999). Springer–Verlag. Proceedings of 1st Intl. Conf. on IFM.

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

Barden, R., Stepney, S., and Cooper, D. 1994. *Z in Practice.* BCS Practitioner Series. Prentice Hall.

Bidoit, M., Sannella, D., and Tarlecki, A. 1998. Architectural specifications in CASL. In *AMAST '98, Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology, Manaus*, Volume 1548 of *LNCS* (1998), pp. 341–357. Springer-Verlag.

Bjørner, D. 2001a. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001j; Bjørner 2001d; Bjørner 2001b; Bjørner 2001e; Bjørner 2001h; Bjørner 2001i; Bjørner 2001c].

Bjørner, D. 2001b. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001a; Bjørner 2001j; Bjørner 2001d; Bjørner 2001e; Bjørner 2001h; Bjørner 2001i; Bjørner 2001c].

Bjørner, D. 2001c. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001a; Bjørner 2001j; Bjørner 2001d; Bjørner 2001b; Bjørner 2001e; Bjørner 2001i; Bjørner 2001h].

Bjørner, D. 2001d. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001a; Bjørner 2001j; Bjørner 2001b; Bjørner 2001e; Bjørner 2001h; Bjørner 2001i; Bjørner 2001c].

Bjørner, D. 2001e. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001a; Bjørner 2001j; Bjørner 2001d; Bjørner 2001b; Bjørner 2001h; Bjørner 2001i; Bjørner 2001c].

Bjørner, D. 2001f. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001g; Bjørner 2001a; Bjørner 2001j; Bjørner 2001d; Bjørner 2001b; Bjørner 2001e; Bjørner 2001h; Bjørner 2001i; Bjørner 2001c].

BJØRNER, D. 2001g. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001a; Bjørner 2001j; Bjørner 2001d; Bjørner 2001b; Bjørner 2001e; Bjørner 2001h; Bjørner 2001i; Bjørner 2001c].

BJØRNER, D. 2001h. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001a; Bjørner 2001j; Bjørner 2001d; Bjørner 2001b; Bjørner 2001e; Bjørner 2001i; Bjørner 2001c].

BJØRNER, D. 2001i. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001a; Bjørner 2001j; Bjørner 2001d; Bjørner 2001b; Bjørner 2001e; Bjørner 2001h; Bjørner 2001c].

BJØRNER, D. 2001j. 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. This paper is one of a series of papers currently being submitted for publication: [Bjørner 2001f; Bjørner 2001g; Bjørner 2001a; Bjørner 2001d; Bjørner 2001b; Bjørner 2001e; Bjørner 2001h; Bjørner 2001i; Bjørner 2001c].

BJØRNER, D. 2001k. *"What is a Method ?" — A Study of Some Aspects of Software Engineering*. IFIP WG2.3. MacMillan, Oxford, UK. Programming Methodology: Recent Work by Members of IFIP Working Group 2.3. Eds.: Annabelle McIver and Carrol Morgan. To be published.

BJØRNER, D. 2002. *Software Engineering: Theory & Practice*. (Publisher is being contacted). These Lecture Notes represent the author's *Chef d'Œvre* — the summary of more than 25 years of research, development and teaching.

BJØRNER, D. AND JONES, C. 1978. *The Vienna Development Method: The Meta-Language*, Volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag.

BJØRNER, D. AND JONES, C. 1982. *Formal Specification and Software Development*. Prentice-Hall.

BOEHM, B. 1981. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ., USA.

BOOCH, G., JACOBSON, I., AND RUMBAUGH, J. 1999a. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison–Wesley, Addison Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, USA.

BOOCH, G., JACOBSON, I., AND RUMBAUGH, J. 1999b. *The Unified Modeling Language User Guide*. Object Technology Series. Addison–Wesley, Addison Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, USA.

BOTTACI, L. AND JONES, J. 1995. *Formal Specification Using Z: A Modelling Approach*. International Thomson Publishing, London.

BOWEN, J. P. 1996. *Formal Specification and Documentation Using Z: A Case Study Approach*. International Thomson Computer Press.

COFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from http://www.brics.dk/Projects/CoFI/.

COUSOT, P. 1996. Abstract Interpretation. *ACM Computing Surveys 28*, 2, 324–328.

COUSOT, P. 1997a. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation (Extended Abstract). *Theoretical Computer Science 6*, 25. Electronic Notes: `http://www.elsevier.nl/locate/entcs/volume6.html`, Elsevier Science.

COUSOT, P. 1997b. Design of Semantics by Abstract Interpretation. In *Mathematical Foundations of Programming Semantics* (23–26 March 1997). MFPS XIII, Carnegie Mellon Univ., Pittsburgh, Pennsylvania, USA.

COUSOT, P. AND COUSOT, R. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In 4^{th} *POPL*, Principles of Programming and Languages (1977), pp. 238–252. ACM Press.

DE BAKKER, J. 1995. *Control Flow Semantics*. The MIT Press, Cambridge, Mass., USA.

DIACONESCU, R. AND FUTATSUGI, K. 1996. Logical Semantics of CafeOBJ. Research Report IS-RR-96-0024S, JAIST. .

DILLER, A. 1994. *Z: An Introduction to Formal Methods* (2nd ed.). John Wiley & Sons.

ERSHOV, A. 1977a. On the essence of translation. *Computer Software and System Programming 3*, 5, 332–346.

ERSHOV, A. 1977b. On the partial computation principle. *Information Processing Letters 6*, 2 (April), 38–41.

ERSHOV, A. 1978a. Mixed computation in the class of recursive program schemata. *Acta Cybernetica 4*, 1, 19–23.

ERSHOV, A. 1978b. On the essence of compilation. In E. NEUHOLD Ed., *Formal Description of Programming Concepts*, pp. 391–420. North-Holland.

FITZGERALD, J. AND LARSEN, P. G. 1997. *Developing Software using VDM–SL*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England.

FUTAMURA, Y. 1971. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2*, 5, 45–50.

FUTAMURA, Y., NOGI, K., AND TAKANO, A. 1991. Essence of generalized partial computation. *Theoretical Computer Science 90*, 1, 61–79. Also in: D. Bjørner and V. Kotov: Images of Programming, North-Holland, 1991.

FUTATSUGI, K. AND DIACONESCU, R. 1998. *CafeOBJ Report — Definition of the Language*. To appear in 1998 as book in the AMAST series at World Scientific .

GANTER, B. AND WILLE, R. January 1999. *Formal Concept Analysis — Mathematical Foundations*. Springer-Verlag. ISBN: 3540627715, 300 pages, Amazon price: US $ 44.95.

GEORGE, C., HAFF, P., HAVELUND, K., HAXTHAUSEN, A., MILNE, R., NIELSEN, C. B., PREHN, S., AND WAGNER, K. R. 1992. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England.

GEORGE, C., HAXTHAUSEN, A., HUGHES, S., MILNE, R., PREHN, S., AND PEDERSEN, J. S. 1995. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England.

GRIES, D. Ed. 1990. *Beauty is Our Business* (1990). Springer-Verlag. Festschrift for Edsger W. Dijkstra.

GUNTHER, C. 1992. *Semantics of Programming Languages*. The MIT Press, Cambridge, Mass., USA.

HAREL, D. 1987. Statecharts: A visual formalism for complex system. *Science of Computer Programming 8*, 3, 231–274.

HAREL, D., PNUELI, A., SCHMIDT, J., AND SHERMAN, R. 1987. On the formal semantic of statecharts. In *Proc. of IEEE Symp. on Logic in Computer Science* (1987).

HAYES, I. J. 1993. *Specification Case Studies* (2nd ed.). Prentice Hall International Series in Computer Science.

HOARE, C. 1978. Communicating sequential processes. *Communications of the ACM 21*, 8 (Aug.).

HOARE, C. 1985. *Communicating Sequential Processes*. Prentice-Hall International.

HONDERICH, T. 1995. *The Oxford Companion to Philosophy*. Oxford University Press, Walton St., Oxford OX2 6DP, England.

IMPERATO, M. 1991. *An Introduction to Z*. Chartwell-Bratt.

INCE, D. C. 1988. *An Introduction to Discrete Mathematics and Formal System Specification*. Oxford Applied Mathematics and Computing Science Series. Oxford University Press, UK.

JACKSON, M. A. 1991. Description is Our Business. In *VDM '91: Formal Software Development Methods* (October 1991), pp. 1–8. Springer-Verlag.

JACKSON, M. A. 1994a. Problems, Methods and Specialisation. *Software Engineering Journal*, 249–255.

JACKSON, M. A. 1994b. *Software Development Method*, Chapter 13, pp. 215–234. Prentice Hall Intl. Festschrift for C. A. R. Hoare: *A Classical Mind*, Ed. W. Roscoe.

JACKSON, M. A. 1995a. Problems and requirements (software development). In *Second IEEE International Symposium on Requirements Engineering (Cat. No.95TH8040)* (1995), pp. 2–8. IEEE Comput. Soc. Press. .

JACKSON, M. A. 1995b. *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. ISBN 0-201-87712-0; xiv + 228 pages.

JACKSON, M. A. 1997. The meaning of requirements. *Annals of Software Engineering 3*, 5–21.

JACKSON, M. A. 2001. *Problem Frames — Analysing and structuring software development problems*. ACM Press, Pearson Education. Addison–Wesley, Edinburgh Gate, Harlow CM20 2JE, England.

JACKY, J. 1997. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press.

JACOBSON, I., BOOCH, G., AND RUMBAUGH, J. 1999. *The Unified Software Development Process*. Object Technology Series. Addison–Wesley, Addison Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, USA.

JENSEN, K. 1985. *Coloured Petri Nets*, Volume 1–2–3 of *EATCS Monographs in Theoretical Computer Science*. Springer–Verlag, Heidelberg.

JONES, C. 1990. *Systematic Software Development using VDM* (Second ed.). Prentice Hall International.

JONES, N. D., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. C.A.R.Hoare Series in Computer Science. Prentice Hall International.

MCDERMID, J. A. AND WHYSALL, P. 1992. *Formal System Specification and Implementation using Z*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK. Withdrawn.

MELLOR, D. H. AND OLIVER, A. May 1997. *Properties*. Oxford Readings in Philosophy. Oxford Univ Press, . ISBN: 0198751761, 320 pages, Amazon price: US $ 19.95.

MOSSES, P. D. 1992. *Action Semantics*. Cambridge University Press: Tracts in Theoretical Computer Science. .

MOSSES, P. D. 1996. CoFI: The Common Framework Initiative for algebraic specification. *Bull. EATCS* 59 (June), 127–132.

MOSSES, P. D. 1997. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97, Proc. Intl. Symp. on Theory and Practice of Software Development*, Volume 1214 of *LNCS* (1997), pp. 115–137. Springer-Verlag.

MOSSES, P. D. 1999. CASL: A Guided Tour of its Design. Documents/CASL/GuidedTour, in [CoFI]; also available as slides: Presentations/GuidedTour and Presentations/Overview, in [CoFI].

NAKAGAWA, A. T., SAWADA, T., AND FUTATSUGI, K. 1997. *CafeOBJ manual (for system version 1.3)*. 142 pages.

NAKAGAWA, K. F. A. 1997. An Overview of CAFE Specification Environment – An Algebraic Approach for Creating, Verifying, and Maintaining Formal Specifications over Net-works. In *ICFEM'97: International Conference on Formal Engineering Methods* (November 1997). IEEE Computer Society: IEEE CS Press.

NIELSON, H. R. AND NIELSON, F. 1992. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons Ltd., Baffins Lane, Chishester West Sussec PO19 1UD, England.

POTTER, B. F., SINCLAIR, J. E., AND TILL, D. 1991. *An Introduction to Formal Specification and Z.* Prentice Hall International Series in Computer Science.

POTTER, B. F., SINCLAIR, J. E., AND TILL, D. 1996. *An Introduction to Formal Specification and Z* (2nd ed.). Prentice Hall International Series in Computer Science.

RANN, D., TURNER, J., AND WHITWORTH, J. 1994. *Z: A Beginner's Guide.* Chapman & Hall, London.

RATCLIFF, B. 1994. *Introducing Specification Using Z: A Practical Case Study Approach.* International Series in Software Engineering. McGraw-Hill.

REISSIG, W. 19?? *??? Petri Nets ???* Springer-Verlag.

ROSCOE, A. 1997. *Theory and Practice of Concurrency.* Prentice–Hall.

SCHMIDT, D. 1986a. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon.

SCHMIDT, D. 1986b. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon.

SOWA, J. F. August 17, 1999. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Pws Pub Co. ISBN: 0534949657, 512 pages, Amazon price: US $ 70.95.

SPIVEY, J. M. 1988. *Understanding Z: A Specification Language and its Formal Semantics*, Volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press.

SPIVEY, J. M. 1992. *The Z Notation: A Reference Manual* (2nd ed.). Prentice Hall International Series in Computer Science.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages.* The MIT Press, Cambridge, Mass., USA.

WOODCOCK, J. C. P. AND DAVIES, J. 1996. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science.

WORDSWORTH, J. B. 1993. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley Publishing Company.

ZAVE, P. AND JACKSON, M. A. 1997. Four dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology 6*, 1 (January), 1–30. .