Principles and Techniques of Abstract Modelling
Some Basic Classifications
Towards a Methodology of Software Engineering

Dines Bjørner

Informatics and Mathematical Modelling, Technical University of Denmark

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

General Terms: Abstraction, Property Oriented Abstraction, Model Oriented Abstraction, Formal Specification, Method, Principle, Technique, Tools: VDM, Z, RSL, B

Additional Key Words and Phrases: Representational Abstraction, Operational Abstraction, Denotation, Computation, Hierarchy, Composition, Configuration, Context, State, Time, Space

The problem to be dealt with in this paper is that of proposing a normative basis for abstract modelling as applied, typically, to the informal and formal description of domains, requirements and software of computing systems.

We cover and contrast such concepts as (i) property and model orientedness, and representational and operational abstraction; (ii) denotational and operational semantics; (iii) hierarchical and compositional development and presentation; (iv) configurations as aggregates of contexts and states; (v) time, space, time & space, parts & wholes, systems, connectors & components; (vi) the modalities of time, of belief and knowledge, of intention and commitment, etc.; and, finally, (vii) events and concurrency.

Some of the contents of this paper may seem well–known, but then we believe that not only are other parts of the current paper reasonably novel, but their juxtaposition in one paper contributes to a clarification of abstract modelling issues. Besides, several of the topical sections (i–vii) contain, we believe, new material of methodological concern not normally associated with software engineering methodology.

This paper should be read in conjunction with, or after having read, [Bjørner 2001f].

Contents

1 Introduction . . . **5**
- 1.1 The Problem to be Addressed . . . 5
- 1.2 A Software Engineering Triptych . . . 5
- 1.3 On Abstraction . . . 7
- 1.4 Related Work . . . 7

Special: This paper is an extract of chapters 5 and 9 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

1.5 Structure of Paper . . . 8

2 Representational and Operational Abstraction 12

2.1 Property and Model Oriented Abstraction . . . 12
2.2 Property Oriented Abstraction . . . 12
2.2.1 Algebras . . . 12
2.2.2 Sorts, Observers and Generators . . . 12
2.2.2.1 Sorts, Observers and Invariants . . . 13
2.2.2.2 Generators . . . 13
2.2.3 Principles and Techniques . . . 14
2.2.4 Discussion . . . 14
2.3 Model Oriented Representational Abstraction . . . 14
2.3.1 Sets . . . 15
2.3.2 Cartesians . . . 15
2.3.3 Lists . . . 16
2.3.4 Maps . . . 17
2.3.5 Functions . . . 18
2.3.6 Discussion . . . 20
2.3.6.1 Self–fulfilment . . . 20
2.3.6.2 "Hedges" . . . 21
2.3.6.3 Sufficiency . . . 21
2.3.6.4 Historical Background . . . 21
2.4 Mostly Model Oriented Operational Abstraction . . . 21
2.4.1 Explicit Function Definitions . . . 22
2.4.2 Pre/Post Function Definitions . . . 23
2.4.3 Axiomatic Function Definitions . . . 26
2.4.4 Discussion . . . 28
2.5 Discussion . . . 28
2.5.1 Understanding . . . 28
2.5.2 Utter Simplicity & Elegance vs. Algorithmitisation . . . 28
2.5.3 Representational and Operational Abstraction is not Enough ! . . . 29

3 Denotations and Computations 30

3.1 Semantics of Specifications . . . 30
3.2 Denotations . . . 30
3.2.1 Valuation versus Denotation . . . 30
3.2.2 Interpretation, Evaluation and Elaboration . . . 30
3.2.3 Discussion . . . 30
3.3 Computations . . . 30
3.3.1 Structural Operational Semantics . . . 30
3.3.2 Mechanical Semantics . . . 30
3.3.3 Discussion . . . 30
3.4 Discussion . . . 30

4 Hierarchies and Compositions 31

4.1 Development versus Presentation . . . 31
4.1.1 "Top–down versus Bottom–up" . . . 31
4.1.2 Development . . . 31
4.1.3 Presentation . . . 31
4.1.4 Discussion . . . 31
4.2 Hierarchies . . . 31
4.2.1 The Hierarchy Concept . . . 31
4.2.2 Hierarchy Examples . . . 31
4.2.3 Discussion . . . 31
4.3 Compositions . . . 31
4.3.1 The Composition Concept . . . 31

4.3.2 Composition Examples 31
4.3.3 Discussion 31
4.4 Blended Styles 31
4.5 Discussion 31

5 Configurations: Contexts and States **32**
5.1 The Problem 32
5.1.1 Static Contexts 32
5.1.2 Dynamic States 32
5.1.3 Discussion 32
5.2 Example I: Language Specification 32
5.2.1 Semantic Values 32
5.2.2 Syntactic Values 32
5.2.3 SemanticFunctions 32
5.2.4 Discussion 32
5.3 Example II: A Reactive System 32
5.3.1 Reactive System Context 32
5.3.2 Reactive System State 32
5.3.3 Reactive System Transition Function 32
5.3.4 Discussion 32
5.4 Discussion 32

6 Time and Space, Parts and Wholes, Systems and Languages **33**
6.1 Ontologies of Time 33
6.1.1 The Problem 33
6.1.2 Time — A Possible Axiomatic Approach 33
6.1.3 Discussion 34
6.2 A Mereology of Parts and Wholes 34
6.2.1 The Problem 34
6.2.2 Parts & Wholes — A Possible Axiomatic Approach 34
6.2.3 Discussion 34
6.3 Towards an Ontology of Spatial Matters 34
6.3.1 The Problem 34
6.3.2 Topology — A Possible Axiomatic Approach 34
6.3.3 Discussion 35
6.4 Spatial Phenomena 35
6.4.1 General 35
6.4.2 Example I: Airspace 35
6.4.2.1 Narrative 35
6.4.2.2 Formal Model 35
6.4.3 Example II: Railway Net 36
6.4.3.1 Narrative 36
6.4.3.2 Formal Model 36
6.4.4 Discussion 37
6.5 Temporal Phenomena 37
6.5.1 General 37
6.5.2 Example: Air Traffic 37
6.5.2.1 Narrative 37
6.5.2.2 Formal Model 37
6.5.3 Discussion 37
6.6 Monotonicity and Continuity 37
6.7 Combinations of Time and Space 37
6.7.1 Co–Existence of Time and Space — A Possible Axiomatic Approach 38
6.7.2 Trading–off Between Time and Space 39
6.8 On Notions of Systems 39
6.8.1 The Problem 39

6.8.2 An Ontology of Systems . . . 39
6.8.2.1 Systems . . . 39
6.8.2.2 Components . . . 40
6.8.2.3 Mechanical System Processes . . . 40
6.8.3 Discussion . . . 42
6.8.3.1 Languages and Systems . . . 42
6.8.3.2 General . . . 43
6.9 Discussion . . . 43

7 Modalities **44**
7.1 Modal, Epistemic and Deontic Logics — A Survey . . . 44
7.1.1 Possible Worlds . . . 44
7.1.2 Modal Operators . . . 44
7.2 Logics of Time . . . 44
7.2.1 Simple Temporal Logic . . . 44
7.2.2 Interval Temporal Logic . . . 44
7.2.3 The Duration Calculi . . . 44
7.3 Modal Logics of Necessity and Possibility . . . 44
7.4 Epistemic Logics of Belief and Knowledge . . . 44
7.5 Deontic Logics of Intention and Commitments . . . 44
7.6 Discussion . . . 44

8 Events and Concurrency **45**
8.1 Processes: Events and Parallelism . . . 45
8.2 CSP–like Process Modelling . . . 45
8.3 Other Modelling Techniques . . . 45
8.4 Discussion . . . 45

9 Conclusion **46**
9.1 What has been Achieved ? . . . 46
9.2 A Critique . . . 46
9.3 Future Work . . . 46

1. INTRODUCTION

Abstraction is a technique used (i) by scientists to help model phenomena of the nature surrounding us, (ii) by artists when interpreting a conceived essence of human nature, and (iii) by mathematicians to express, as one example, properties of two or more previously established mathematical theories. A mathematical theory is in itself, by its very nature, an abstraction: It establishes a number of definitions and an axiom system, formulates theorems and proves these.

1.1 The Problem to be Addressed

Computer software represents abstractions of the application area to which it is being applied. The software makes (oftentimes too) gross assumptions about that world. When "too gross" the software is a failure: Does not fit with the actual world of the users of that software.

In developing software we have (amongst others) two concerns: Functionality and efficiency. The software must prescribe functions that relate to the actual world, and the software rendition of these must, as a basis for executions, be reasonably efficient.

The two concerns have often lead to program code whose documentation is opaque. As a result, if the annotated code is all the documentation there is, the code is a poor basis for maintenance. As a legacy it becomes near useless.

The idea is therefore to additionally provide specifications of abstract designs — that prescribe the functionality — from which (so–called efficient) concrete code prescriptions can be developed. The maintenance engineer now first studies the abstract design specification, then the concrete code. Software design thus comes to consists of several stages and steps of development.

So much for an initial motivation of abstraction.

• • •

The problem to be addressed by this paper is that of presenting a number of abstraction principles and techniques, and to briefly relate on associated (abstraction support) tools that help the software engineer in obtaining humanly readable specifications — and notably such principles and techniques that apply to a larger spectrum of software development than just software design: Also to requirements specification and, as we shall motivate shortly, to domain description.

1.2 A Software Engineering Triptych

Before software[1] can be designed, their requirements must be engineered. And before requirements can be expressed, we must understand the domain.

The abstract modelling principles, techniques and tools covered in this paper are claimed applicable across the spectrum of all three engineering "universes of discourse": Domain, requirements and software design.

—By <u>domain</u> we understand the *application area*, the *"actual world"* in which some desired (or actual) *software*[1] is to reside (resp. resides).

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

Thus the ‘domain’ of the abstraction principles, techniques and tools of this paper is all the three universes of discourse: (i) The engineering of actual application domains, (ii) the engineering of their requirements and (iii) the engineering of their software design

—By domain engineering we understand a set of science–based 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 — whatsoever — to any (subsequently) *required software*, let alone to any *design* of such *software*[1].
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.

In addition to the abstract modelling principles, techniques and tools of this paper there is a rich set of domain specification abstraction principles and techniques. These are treated in a separate paper [Bjørner 2001g].

—By requirements engineering we understand a set of science–based engineering activities which results in a set of *models*, ie. a set of *documents informing* about, *describing*, and *analysing* what is expected from *software*[1] 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*[1].
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.

In addition to the abstract modelling principles, techniques and tools of this paper there is a rich set of requirements specification abstraction principles and techniques. These are treated in a separate paper [Bjørner 2001j].

—By software[1] design we understand a set of science–based 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*[1] 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*.

The specification universe of software design calls for increasingly concrete, ie. less abstract prescriptions. In our currently planned series of methodology papers: [Bjørner 2001f], the present paper, and [Bjørner 2001a; Bjørner 2001j], we shall not go into software design abstraction principles and techniques. We refer to the treatments of [Bjørner and Jones 1982; Jones 1990a; George et al. 1995; Fitzgerald and Larsen 1997].

—By software engineering we thus understand a confluence of domain engineering, requirements engineering and software design.

1.3 On Abstraction

So[2] *abstraction is applied in the process of describing complex phenomena. One "simplifies" reality, leaving out empirical observations that are deemed irrelevant for whatever purposes the abstraction is to serve. Abstraction arises from a conscious decision to advocate certain desired phenomena as being fundamental, by exposing their similarities and leaving out their dissimilarities.*

Since software documents "are" not the reality, but only descriptions of a perceived actual world, they are abstractions. They are not the "real thing".

1.4 Related Work

Although most textbooks and monographs on formal software development methods[3] contain much information on abstract modelling, it appears, at least to this author, that more emphasis has been placed on technicalities than on identifying what exactly is meant by 'formal method', and in enunciating attendant principles and techniques. The wisdom of the books referenced in footnote 3, in our opinion, has not been crystallised in clear and crisp principle and technique enunciations.

Some papers come close to what we aim at: [Jackson 1995a]

MORE TO COME

[]

MORE TO COME

[]

MORE TO COME

[]

MORE TO COME

[]

MORE TO COME

[2]This *paragraph* "abstracts" Tony Hoare's introductory words in [Hoare 1972].

[3]Viz. the following `VDM` [Bjørner and Jones 1982; Jones 1990a; Fitzgerald and Larsen 1997; Jacky 1997], `Z` [Imperato 1991; Potter et al. 1991; Diller 1994; Bowen 1996; Woodcock and Davies 1996; Jacky 1997], `RAISE` [George et al. 1992; George et al. 1995] and `B` [Abrial 1996a] books.

Most papers contribute one, or another specific principle or technique, or both, or, most often, a tool to support an existing principle or technique: Together these many papers "lay brick upon brick", but few papers try to gather the parts into a whole.

With this paper we would like to see more papers that try the latter: *Gather principles and techniques into methods,* and, *methodologically,* "compare" such methods.

1.5 Structure of Paper

• Section 2 treats the subject of Representational and Operational Abstraction. It does so in basically two parts spread over three subsections:

* Section 2.2: *Property Oriented Abstraction,*
* Section 2.3: *Representational Abstraction,* and
* Section 2.4: *Operational Abstraction.*

The latter two sections cover model orientedness — and hence one can separate representation from operation. Something that is not possible, really, in for example, the "classical" algebraic semantics approaches to property oriented specifications.

Section 2.1 refers to definitions of the notions of *Property Oriented Abstraction* (Section 2.2) and *Model Oriented Abstraction* (Sections 2.3–2.4).

* *Property Oriented Abstractions* are cursorily dealt with in Section 2.2.
* We otherwise refer to a rich literature on algebraic specification [Mosses 1996; Mosses 1997; Mosses 1999b; Bidoit et al. 1998; Diaconescu and Futatsugi 1996; Nakagawa 1997; Nakagawa et al. 1997; Futatsugi and Diaconescu 1998].

  On one hand that literature does not really forward the principles and techniques of abstract modelling. Rather it focuses on technicalities. On the other hand we claim that the principles and techniques of Sections 3–6, which illustrate their points through either property oriented or model oriented abstractions, (thus) cover such property oriented abstract modelling principles and techniques.
* Section 2.3 (*Representational Abstraction*) now illustrates the notions of *Set Abstractions* (Section 2.3.1), *Cartesian Abstractions* (Section 2.3.2), *List Abstractions* (Section 2.3.3), *Map Abstractions* (Section 2.3.4), and *Function Abstractions* (Section 2.3.5).

  We emphasise — to the unaware reader — that a distinction is here being made between function abstraction and operation abstraction. In pursuing the former we abstract values of tangible or intangible phenomena as functions, In pursuing the latter we abstract the definition of operations (functions, predicates) over the former !
* Section 2.4 (*Operational Abstraction*) then illustrates the notions of *Explicit Function Definitions* (Section 2.4.1), *Pre/Post Function Definitions* (Section 2.4.2), and *Axiomatic Function Definitions* (Section 2.4.3).

The last two sections (2.3 and 2.4) summarise [Bjørner and Jones 1978; Bjørner and Jones 1982; Jones 1990a; Fitzgerald and Larsen 1997], and are brought here, in "ultra–brief" form, in order that the present paper present a reasonably comprehensive treatment of the subject of abstraction.

• Section 3 'survey' treats the subject of Denotations and Computations in two subsections:

* Section 3.2 (*Denotations*) briefly covers the use of denotational semantics notions in general abstractions. This denotational tradition was started by and still relies on the basic ideas of Strachey and Scott: [Strachey and Scott 1970; Scott 1970; Scott and Strachey 1971; Scott 1972; Strachey 1973; Strachey 1974; Scott 1981]: *Valuation versus Denotation* (Section 3.2.1) and *Interpretation, Evaluation and Elaboration* (Section 3.2.2).
* Section 3.3 (*Computations*) similarly briefly covers the use of computational semantics notions in general abstractions. The *Mechanical Semantics* (Section 3.3.2) tradition was started by and still relies on the basic ideas of Landin: [Landin 1964; Landin 1965b; Landin 1965d; Landin 1965c; Landin 1965a; Landin 1966c; Landin 1966b; Landin 1966a]. The *Structured Operational Semantics* (Section 3.3.1) tradition was started by and still relies on the basic ideas of Plotkin: [Plotkin 1981].

• Section 4 treats the subject of Hierarchies and Compositions in two subsections. This section basically has no previous abstract specification modelling literature to refer to, hence rely on. Of course software design work, notably by David Parnas, is always worth referring to: [Parnas 1972b; Parnas 1972a] and others, nicely collected in [Parnas et al. 2001].

* Section 4.2 (*Hierarchies*) presents principles and techniques for the "top down" decomposition of problems (whether domain, requirements or software design models) — whether when being developed (ie. "solved") or being presented: From "larger" concepts "smaller" ones are being identified.
* Section 4.3 (*Compositions*) presents principles and techniques for the "bottom up" composition of problems (whether domain, requirements or software design models) — whether when being developed (ie. "solved") or being presented. From "smaller" concepts "larger" ones are being identified.

Development or presentation of actual specifications usually represent a mix of both approaches. But if neither are clearly understood and elucidated, there is the risk of a confused, "messy" development and/or presentation (cum documentation).

• Section 5 treats the subject of Configurations: Contexts and States in basically two subsections:

* Section 5.1.1 (*Static Contexts*) covers a concept of 'context'. A 'context' is binding of (so–called constant) identifiers to fixed values.
* Section 5.1.2 (*Dynamic States*) covers a concept of 'state'. A 'state' is a summary of an interpretation, a computation, up to the point(s) to which we attach that notion of 'state'. Usually a 'state' is a binding, in a model, of (so–called variable) identifiers of the specification to their "current" values. Thus we can say that the 'state' notion dealt with here is a temporal (hence dynamic) notion.

Section 5 then proceeds to exemplify combinations, in a spectrum from "fully" static, via "semi–static/semi–dynamic", to "fully dynamic", of these concepts. We have known of the distinction between contexts and states from, for example, the semantics descriptions of the semantics of imperative programming languages — where they are referred to, usually, as environments, respectively stores. What we aim at in Section 5 is to propagate the principles and techniques of environment and store modelling to other descriptions than programming language semantics.

• Section 6 treats the subjects of Time, Space, Mereology and Systems. Ontology is the study and knowledge of being. We single out, and treat as ontologies, the notions of time, space, mereology and systems, because of their central rôle also in philosophy, in general, and in the philosophies and theories of science, in particular.

* We survey *Ontologies* of Time and Space in two subsections:
  * Time (Section 6.1), and
  * Space (Section 6.3).
* In–between Sections 6.1 and 6.3, in Section 6.2, we cover a notion of Mereology. This is done in order to prepare for ontologies of space (ie. topology) and systems.
* After this, Section 6.5 (*Temporal Phenomena*) gives examples of modelling temporal phenomena,
* and Section 6.4 (*Spatial Phenomena*) gives examples of modelling spatial phenomena.
* Section 6.7 relate the two, time and space, as they are indeed, classically,

  One set of issues is that of *Monotonicity and Continuity* (Section 6.6). Where classical, mathematical calculus, since Leibniz and Newton had their calculi of differentials and integrals, computer and computing science, till now, has lacked "counterparts" of such calculi. Section 6.6 will sketch beginnings.
* Section 6.8, finally, extends the mereology and topology ontologies to cover a notion of Systems, Components, and Connectors.

• Section 7 treats the subjects of Modalities: The logics of time, of belief and knowledge, of intention and commitment, etc.

MORE TO COME

• Section 8, finally, treats the subject of Events and Concurrency.

MORE TO COME

• • •

The paper thus brings together a number of strands of abstraction and modelling: The technicalities of property and model oriented specifications (Section 2), the principles and techniques of denotational and computational modelling (Section 3), the principles and techniques of hierarchical and compositional modelling (Section 4), the principles and techniques of configurational modelling: Contexts and states (Section 5), the principles and techniques of temporal and spatial modelling, of modelling topological artifacts and modelling systems, components and connectors. (Section 6), and the principles and techniques of modelling modalities (such as time, belief and knowledge, intention and commitment, ***, and *** (Section 7).

Other abstraction and modelling principles and techniques relate more specifically to respective phases and stages of software development, and are treated in separate papers: *Domain modelling* [Bjørner 2001a], and *requirements modelling* [Bjørner 2001j]. [Bjørner 2001f] aims at providing a common didactic basis for descriptions.

It is the proper balance between and judicious choice amongst these diverse abstraction and modelling principles and techniques that we believe are the contributions of this paper, its "predecessor" [Bjørner 2001f] and its "sequels" [Bjørner 2001a; Bjørner 2001j]. *"Together they stand, together they fall !*

2. REPRESENTATIONAL AND OPERATIONAL ABSTRACTION

A conventional way of viewing the computable aspects of an actual ("the real") world is to "divide" it, that world, into entities ("things") and manipulations of (queries [observations] over, functions on, operations on) these entities. In mathematics we would say that there are values and there are functions that apply to values and yield values — and that functions can be values too. Speaking of functions and values already has put us on a course of model orientedness: Values have representations, and functions on, or over, values is 'operational' thinking. Hence we shall, as concerns our treatment of model orientedness, "divide" our presentation into two parts: Representational, respectively Operational Abstraction.

2.1 Property and Model Oriented Abstraction

For the general topic of modelling we refer to [Bjørner 2001f]'s Section 2.2. In the following we rely on that section's definitions and discussion of the two concepts: Property and Model Oriented Modelling.

Section 2.2 deals with Property Oriented Abstraction, whereas Sections 2.3 and 2.4 deal with Model Oriented Abstraction.

2.2 Property Oriented Abstraction

Property oriented abstractions collectively cover both representational and operational abstractions. In "one fell swoop" one both describe properties of operations and properties of values when presenting a property oriented abstraction — at least when doing so in the algebraic semantics style.

2.2.1 *Algebras.* An algebra, generally, is a usually infinite set of values and a usually finite set of operations on these. An algebra is presented when defining a set of equations involving both values and operations.

2.2.2 *Sorts, Observers and Generators.* We present values by defining their abstract types, ie., their sorts. Observers are functions which "extract", or compute, components values from, respectively over, composite values. In the example below we shall most illustrate sorts and observers. Generators are functions which "construct" values of the postulated sorts. We shall postpone, till Section 2.4.3, the illustration of generator functions and the definition of their properties. We present operations as axioms governing both values and operations. In contrast to conventional algebraic semantics approaches[4] we favour a rather free form of axiom expressions, such as is allowed in the formal specification language RSL [George et al. 1992; George et al. 1995]. Using RSL allows us to make use of its set notation. That is: Since set theory can be considered an adjunct to mathematical logic, we consider axioms expressed using set notation as properly property oriented. Sometimes we even go beyond sets and makes use of RSL's simple notion of sequences (ie. lists) — as in [7] below.

[4]Examples are: The CafeOBJ approach [Diaconescu and Futatsugi 1996; Nakagawa 1997; Nakagawa et al. 1997; Futatsugi and Diaconescu 1998], and the COFI/CASL approach [CoFI ; CoFI Language Design Task Group 1998; Mosses 1996; Mosses 1997; Mosses 1999a; Bidoit et al. 1998].

2.2.2.1 *Sorts, Observers and Invariants.* In presenting the types (ie. sorts) and values of an algebra (ie. an abstract data type, say A) we often need to express *constraints* (ie. *well—formedness* conditions, say wf_A) on composite values, ie. on values composed from values of different sorts. The example shown now illustrates this kind of *invariants* over the values of the algebra.

Example: [1] A railway system rail net consists of one or more lines and two or more stations. [2] Nets, lines and stations consists of rail units. [3] Units of lines or stations of a net are units of the net. [4] Rail units have connectors: Two if linear, three if simple switches, four if either simple cross–overs or if cross–over switches. [5] At most two units of a rail net share same one connector. [6] Lines are linear sequences of one or more linear units. [7] A set of units form a linear sequence if they connect properly. [8] A line connects exactly two different stations in that there will be units at either end of the line which share a connector with units in respective stations.

type
[1,2,4] N, L, S, U, C
value
[1] obs_Ls: N → L-**set**, obs_Ss: N → S-**set**,
[2] obs_Us: N|L|S → U-**set**, obs_Cs: U → C-**set**,
[4] is_linear: U → **Bool**, is_switch: U → **Bool**, is_cross: U → **Bool**
[6] is_seq_lin: U-**set** → **Bool**
[7] is_lin_seq: U-**set** → **Bool**
axiom
  ∀ n:N,s:S,ℓ:L,u:U,c:C •
[1] **card** obs_Ss(n) ≥ 2 ∧ **card** obs_Ls(n) ≥ 1 ∧
[3] (ℓ∈ obs_Ls(n) ⇒ obs_Us(ℓ) ⊆ obs_Us(n) ∧
    s ∈ obs_Ss(n) ⇒ obs_Us(s) ⊆ obs_Us(n)) ∧
[4] **cases card** obs_Cs(u) **of**
    2 → is_linear(u),3 → is_switch(u),4 → is_cross(u) **end** ∧
[5] **card** { u | u:U • c ∈ obs_Cs(u) } ≤ 2 ∧
[6] **let** us = obs_Us(l) **in card** us ≠ 0 ∧ is_lin_seq(us) ∧
      ∀ u:U • u ∈ us ⇒ is_linear(u) **end**
value
[7] is_lin_seq(us) ≡
      let q:U* • **len** q = **card** us ∧ **elems** q = us ∧
      ∀ i:**Nat** • {i,i+1} ⊆ **inds** q ⇒
        card(obs_Cs(q(i)) ∩ obs_Cs(q(i+1))) = 1 ∧
        len q > 2 ⇒ **card**(obs_Cs(q(1)) ∩ obs_Cs(q(**len** q))) = 0
      end

We leave expressing [8] as an exercise. More axioms need be expressed than are presented here [Bjørner 2001i] ▪

2.2.2.2 *Generators.* As mentioned above we shall postpone till Section 2.4.3 our treatment of generators and, hence, the exemplification of the interplay, in property oriented, axiomatic definitions, between observers and generators in the defnition of both and either !

2.2.3 *Principles and Techniques.* We shall only summarise a few principles and techniques of algebraic semantics specifications — and otherwise refer to the already referenced literature.

Values (of types) have no importance in isolation. Neither have functions. "Together they stand." When expressing properties of values (ie. types) and functions — as abstractly as possible, in order to capture their very essence — we need not be concerned with any conceivable efficiency of representing these values, respectively of "executing" these functions. In fact we need not, for many aspects of domains that we wish to describe be concerned with their computability. We can capture this in the following:

Principle: Separation of Concerns: First focus on `what` the value and function properties are, then, after having expressed and understood these, focus on `how` to represent, respectively "execute" these ▪

Principle: Algebraic, or Full Abstraction: When setting out on the very first descriptions, say typically of domain intrinsics, choose property oriented abstraction, ie. choose to follow the principle of separation of concerns ▪

Techniques: Algebraic Specification: Property oriented abstraction techniques focus on listing (i) the abstract types (ie. sorts), (ii) the observer, generator and auxiliary function signatures, and (iii) the axioms over values of these sorts and over these functions, hence of defining an algebra ▪

2.2.4 *Discussion.* Thus we do not present commonly know principles and techniques of constructing equational axioms — such techniques as are defining the "meaning of" values (ie. types) and functions by defining effects of observer functions on values, the latter being expressed using generator functions. This we consider standard knowledge from schools of, and textbooks on, algebraic semantics (cf. Footnote 4).

Instead we shall, in this paper, focus on "near—full" abstraction using the model oriented approach. We shall be using the same specification language as above, RSL [George et al. 1992; George et al. 1995; George et al. 2001]. The forthcoming textbook *Software Engineering: Theory & Practice* [Bjørner 2002] will, also, mainly be using RSL, but will, as also indicated in the present paper, also be presenting examples in a few other formal notations (viz.: DC (Duration Calculus), and various formalised and to be formalised extensional subsets of RSL).

2.3 Model Oriented Representational Abstraction

The meaning of well–formed RSL specification parts, such as shown in the current paper, is always a possibly empty, possibly singleton, possibly infinite set of algebras, namely bindings of names (identifiers of types, variables, functions, etc., to their mathematical model, such as now detailed.

Whereas the algebraic models of property oriented specifications were implicitly postulated, they are now explicitly presented by model oriented specifications.

Principle: The Principle of Model Oriented Representational Abstraction expresses a representation of values (and hence types, viewed simply as set of values) in terms of mathematical concepts such as sets, Cartesians, lists, maps and functions (where maps are a special kind of functions) such that the choice most "closely" models, or emphasises, properties of these values (types) ▪

2.3.1 *Sets. Principle cum Techniques:* When a phenomenon, a "thing" in a software engineering, or in an actual, world, being specified in the model oriented manner, most "aptly" can be expressed as a *[o] possibly variable, possibly void, possible infinite 'size' aggregation of unordered, distinct* — but not necessarily individually uniquely and simply distinguishable — elements (ie. subsidiary phenomena), and when manipulations of such (the primary) phenomena most "fittingly" can be expressed in terms of *[i] collecting them into other such phenomena (set union), [ii] constructing such a phenomenon of elements common to two or more such phenomena (intersection), [iii] selecting arbitrary (immediate) elements of such phenomena or inquiring as to the presence (or not) of a specific element of such a phenomenon (set element membership), [iv] making such a phenomenon, from amongst two, such that its elements are in one, but not the other (set complement), [vii] inquiring whether one such phenomenon is "contained" in another such (subset),* and *[viii] inquiring as to the number of components in such a phenomenon (set cardinality),* then models of such phenomena in terms of the set data type seem "appropriate" ∎

Example: The example of Section 2.2.2 should suffice as an example of set oriented abstraction ∎

2.3.2 *Cartesians. Principle cum Techniques:* When a phenomenon, a "thing" in a software engineering, or in an actual, world, being specified in the model oriented manner, most "aptly" can be expressed as a *fixed 'size' aggregation* of (typically different typed) elements (ie. subsidiary phenomena), and when manipulations of such (primary) phenomena most "fittingly" can be expressed in terms of *[i] grouping (not necessarily distinct) such typed elements into (not necessarily ordered) such (the primary, fixed) phenomena (composition, grouping: "$(\circ,\circ,\ldots,\circ)$"), [ii] selecting specific components from such (primary) phenomena (decomposition),* and *[iii] inquiring as to whether two such (primary) phenomena "are the same or not" (equality),* then models of such (primary) phenomena in terms of the Cartesian product data type seem "appropriate" ∎

Example: We continue the example of Section 2.2.2 (and, by implication, Section 2.3.1 above). A rail unit affords train traffic in one or more directions (paths) through the unit. We model such directions by ordered (Cartesian) pairs of connectors. At any one time a unit is either in one such state (σ) (of permitted, or possible, directions), or in another (σ'). Over a lifetime any rail unit thus defines a state space (ω).

type
  $\mathrm{P} = \{|\ (\mathrm{c},\mathrm{c}'):\mathrm{C}\times\mathrm{C} \bullet \mathrm{c}\neq\mathrm{c}'\ |\}$
  $\Sigma = \mathrm{P}$-**set**
  $\Omega = \Sigma$-**set**
value
  $\mathrm{obs\_}\Sigma\mathrm{:\ U} \to \Sigma$
  $\mathrm{obs\_}\Omega\mathrm{:\ U} \to \Omega$
axiom
  $\forall\ \mathrm{u:U} \bullet \mathrm{obs\_}\Sigma(\mathrm{u}) \in \mathrm{obs\_}\Omega(\mathrm{u}) \wedge$
    let $\mathrm{cs} = \mathrm{obs\_Cs(u)}$ **in** $\forall\ \sigma:\Sigma\bullet\sigma \in \mathrm{obs\_}\Sigma(\mathrm{u}) \Rightarrow \forall\ (\mathrm{c},\mathrm{c}'):(\mathrm{C}\times\mathrm{C})\bullet(\mathrm{c},\mathrm{c}') \in \sigma$ **end**

An example within the example: The linear and the switch units of Figure 1 thus

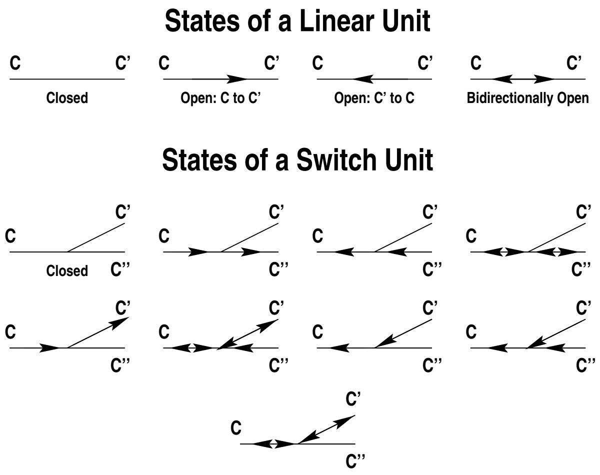

Fig. 1. Sample Rail Unit States

— in their most abstract form, as one might consider it in a domain intrinsics description — have possible states as indicated. The switch states can be enumerated ($c' \equiv c_/$, and $c'' \equiv c_|$.):

$$\begin{array}{ll}\omega_{g_s} : \{ & \{\}, \{(c,c_|)\}, \{(c_|,c)\}, \{(c,c_|),(c_|,c)\}, \\ & \{(c,c_/)\}, \{(c,c_/),(c_/,c),(c_|,c)\}, \{(c_/,c)\}, \{(c_/,c),(c_|,c)\}, \\ & \{(c,c_/),(c_/,c)\} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}\end{array}$$

ω_{g_s} ideally models a general switch. Any particular switch ω_{p_s} may have $\omega_{p_s} \subset \omega_{g_s}$[5] ▪▪

2.3.3 *Lists. Principle cum Techniques:* When a phenomenon, a "thing" in a software engineering, or in an actual, world, being specified in the model oriented manner, most "aptly" can be expressed as a *variable, possibly void, possibly infinite 'size' aggregation of ordered, not necessarily distinct* elements (ie. subsidiary phenomena), and when manipulations of such (primary) phenomena most "fittingly" can be expressed in terms of *[i] juxtaposing two or more such phenomena into other such (primary) phenomena (concatenation), [ii] selecting, by position, the first, or all but the first (immediate) components of such (primary) phenomena (indexing, head, tail),* and *[iii] inquiring as to the number of components in such a (primary) phenomenon (length),* then models of such phenomena in terms of the list data type seem "appropriate" ▪

Example: We continue the examples of Section 2.2.2 (ie. Section 2.3.1) and Section 2.3.2. Train normally travels along open routes. A route is a sequence of one or more unit paths. A path of a unit is open if it is in the current state of that unit. A route is open if all its paths are open.

type

[5]Nothing is said about how a state is determined: Who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible or virtual (ie. invisible, intangible) signals up or down the rail away from the switch.

$R' = (U \times P)$**-set**
$R = \{|\ r{:}R' \bullet \mathrm{wf\_R}(r)\ |\}$
value
$\mathrm{wf\_R}$: $R' \rightarrow$ **Bool**
$\mathrm{wf\_R}(r) \equiv$
$\mathrm{is\_lin\_seq}(\{u|(u,p){:}(U{\times}P){\bullet}(u,p) \in r\}) \wedge$
$\forall\ (u,(c,c')){:}(U{\times}P){\bullet}(u,(c,c')) \in r \Rightarrow \{c,c'\} \subseteq \mathrm{obs\_Cs}(u)$
$\mathrm{open\_R}$: $R \rightarrow$ **Bool**
$\mathrm{open\_R}(r) \equiv$
$\forall\ (u,p){:}(U{\times}P){\bullet}(u,p) \in r \Rightarrow p \in \mathrm{obs\_\Sigma}(u)$

Routes are indeed abstractions. They really cannot be designated, but only, as here, defined.[6] And modelling a route as a list is thus also an abstraction ∎

2.3.4 *Maps. Principle cum Techniques:* When a phenomenon, a "thing" in a software engineering, or in an actual, world, being specified in the model oriented manner, most "aptly" can be expressed as a *variable, possibly void, possibly infinite, 'size' aggregation of otherwise uniquely* (definition set) *distinguishable (and enumerable)* (range set) elements (subsidiary phenomena), and when manipulations of such (the primary) phenomena most "fittingly" can be expressed in terms of *[i] collecting them into other such phenomena (union), [ii] selecting range values of such phenomena based on their unique definition set elements (ie. subsidiary phenomena) (function (ie. map) application)*, and *[iii] projecting such (the primary) phenomena onto, or to not include specific subsets of definition set elements (projection)* (set complement, one way or the other) then models of such phenomena in terms of discrete, definition set enumerable functions, ie. the map data type, seem "appropriate" ∎

Examples: We show a number of examples.

* Time Tables: We continue the examples of Sections 2.2.2 (ie. Section 2.3.1), 2.3.2, and 2.3.3. A train time table associates to every train (by its name) the arrival and departure times from uniquely distinguishable, but otherwise temporally ordered stations (by names). Hence essentially two abstractions lend themselves directly:

type
$TT_{m'} = Tn \ \overrightarrow{m}\ (Sn \ \overrightarrow{m}\ (T \times T))$
$TT_m = \{|\ tt{:}TT_{m'} \bullet \mathrm{wf\_TT}_m(tt)\ |\}$
value
$\mathrm{wf\_TT}_m$: $TT_{m'} \rightarrow$ **Bool**
type
$TT_{\ell'} = Tn \ \overrightarrow{m}\ (Sn \times (T \times T))^*$
$TT_\ell = \{|\ tt{:}TT_{\ell'} \bullet \mathrm{wf\_TT}_\ell(tt)\ |\}$
value
$\mathrm{wf\_TT}_\ell$: $TT_{\ell'} \rightarrow$ **Bool**

[6] We refer to Michael Jackson's [Jackson 1995b] "lexicon" entries on *Designations* and *Definitions* and [Bjørner 2001f]'s corresponding Sections 5.1 and 5.2.

where the well–formedness criteria include an appropriate temporal ordering of station visits ▪

* Banks: A bank has customers (clients), k:K, and clients have numbered, c:C, accounts, a:A. Two or more clients may share accounts. Clients, possibly assisted by bank clerks, effect transactions on banks. Several such may be pending, that is: Waiting to be performed during some future time interval — such as for example (i) monthly printing of last months' transactions, (ii) regularly movement of funds from one account, say an income account, to a mortgage account, (iii) [friendly] reminder of such non–automatic payments, (iv) [stern] reminder of payments of overdrafts, etc.

  type
  K, C, A, Tran
  $\mathrm{B}' = \mathrm{SHA} \times \mathrm{ACC} \times \mathrm{PEN}$
  $\mathrm{B} = \{|\ \mathrm{b:B}' \bullet \mathrm{wf\_B(b)}\ |\}$
  $\mathrm{SHA} = \mathrm{K} \underset{m}{\rightarrow} \mathrm{C}\text{-}\mathbf{set}$
  $\mathrm{ACC} = \mathrm{C} \underset{m}{\rightarrow} \mathrm{A}$
  $\mathrm{PEN} = \mathrm{TI} \underset{m}{\rightarrow} \mathrm{R}$
  $\mathrm{R} = \mathrm{C} \underset{m}{\rightarrow} \mathrm{Tran}\text{-}\mathbf{set}$
  $\mathrm{TI} = \{|\ (\mathrm{t,t'}):(\mathrm{T}\times\mathrm{T}) \bullet \mathrm{larger\_or\_equal(t',t)}\ |\}$
  value
  $\mathrm{wf\_B}: \mathrm{B}' \rightarrow \mathbf{Bool}$
  $\mathrm{wf\_B}(\xi,\alpha\mathrm{s},\pi) \equiv$
  $\cup\ \mathbf{rng}\ \xi = \mathbf{dom}\ \alpha\mathrm{s} \wedge \cup\{\{\ \mathrm{c|c:C\bullet c} \in \mathbf{dom}\ \mathrm{r}\} \mid \mathrm{r:R} \in \mathbf{rng}\ \pi\}\subseteq\mathbf{dom}\ \alpha\mathrm{s}$

  There may be other bank configuration (context and state) components ▪

* Hierarchical Directories: A directory associates, to a number of distinct directory names, either a resource (other than a directory), or a sub–directory, which is a directory.

  type
  Dn, R
  $\mathrm{D} = \mathrm{Dn} \underset{m}{\rightarrow} (\mathrm{R} \mid \mathrm{D})$

  Recursion "stops" when all of a directory's associations are non–directory resources ▪

The examples hint at maps "being the real workhorse" of abstractions ! ▪

2.3.5 *Functions. Principle cum Techniques:* When a phenomenon, a "thing" in a software engineering, or in an actual, world, being specified in the model oriented manner, most "aptly" can be expressed, will, as a *function,* that is, as "something", f, which when applied to "another thing", an element (a subsidiary phenomenon) of its definition set ($\mathcal{D}(f)$), yields a "third thing", a result (another subsidiary phenomenon) of its range ($\mathcal{R}(f)$) — where, in contrast to maps, the definition set, generally, is not decidable, that is: Cannot be enumerated per se ("in and by itself")[7] — then models of these phenomena in terms of functions seem appropriate ▪

[7]That is: We can speak of the meta–functions $\mathcal{D}$ and $\mathcal{R}$, but $\mathcal{D}$, for example, is not "algorithmically" expressible. We put double quotes around the term 'algorithmically' since (i) the same

Examples: We show three examples.

* Train Traffic: We continue the examples of Sections 2.2.2 (ie. Section 2.3.1), 2.3.2, and 2.3.3. Train traffic is continuous over time, and for each time point a finite number of distinct trains are positioned along routes of the rail net. That is, traffic is seen here as a function from time to positions of trains, the latter being abstracted as maps.

type
  TN
  $\mathrm{TF}' = \mathrm{T} \to (\mathrm{N} \times (\mathrm{TN} \underset{m}{\overrightarrow{\to}} \mathrm{R}))$
  $\mathrm{TF} = \{|\ \mathrm{rf{:}TF}' \bullet \mathrm{wf\_TF(tf)}\ |\}$
value
  wf_TF: $\mathrm{TF}' \to$ **Bool**
  $\mathrm{wf\_TF(tf)} \equiv \ldots\ \mathrm{wf\_Movement}(\mathrm{t},\mathrm{t}')(\mathrm{tf})\ \ldots$

The traffic well–formedness predicate (wf_TF) must at least express monotonicity of traffic. The wf_Movement, provided that suitable, “infinitesimally close” time points (t,t′) are given, expresses this monotonicity. The ellipses (...), amongst others, express that there are no ghost trains[8]. Figure 2 illustrates seven cases

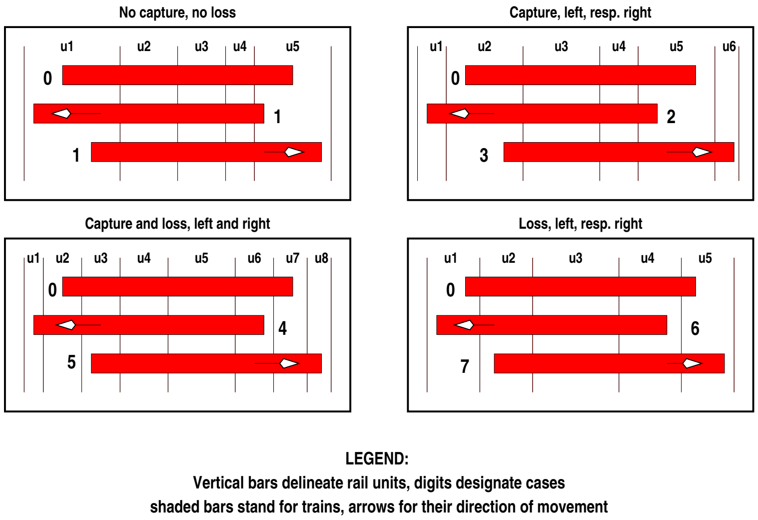

Fig. 2. Train Movement

of such “monotonicity” ■

for the corresponding, decidable map operators, **dom** and **rng**, was possible, even for infinite “size” maps due to their being a meta–representation possible for maps, namely one in terms of enumerated sets of pairs; but (ii) no such meta–representation is possible for what is essentially λ–definable functions.

[8]Trains do not disappear only to reappear.

* Train Scheduling: One can explain any observation of train traffic, ie. discretised traffic, in terms of a rail net and a time table: Whether the traffic occurs on that net and on, before or after time. Thus, given a train time table and a rail net one can postulate a function which generates all possible traffics that adhere to the net and the time table.

  type
  $\quad$ dTF′ = T $\overrightarrow{m}$ (N × (Tn $\overrightarrow{m}$ R))
  $\quad$ dTF = {| dtf:dTF′ • wf_dTF(dtf) |}
  value
  $\quad$ wf_dTF: dTF′ → **Bool**
  $\quad$ schedule: N × TT → dTF-**infset**
  $\quad$ off_time: TT × dTF → (T $\overrightarrow{m}$ Tn-**set**)

  One can verbalise the above as: A time table denotes functions from nets to potentially infinite sets of (discretised) traffics. That is: We look at time tables as syntax and seek the semantics of those kinds of syntax. And, as we shall later review, semantics is expressed as functions ∎

* Bank Transaction Processing: Bank transactions are expressible syntactically and their semantics can be given:

  type
  $\quad$ T /* Time */
  $\quad$ M == mk_M(n:**Nat**) /* Monies */
  $\quad$ Tran == mk_Deposit(c:C,m:M)
  $\qquad\quad$ | mk_Withdraw(k:K,c:C,n:**Nat**)
  $\qquad\quad$ | mk_Transfer(fk:K,fc:C,n:**Nat**.tc:C)
  $\qquad\quad$ | ...
  $\quad$ Resp == ok | nok | M
  $\quad$ A = M × (Tran × T)* × ...
  value
  $\quad$ I: Tran → B → T $\overset{\sim}{\to}$ B × Resp

  This transactions, ie. syntax, are (is) seen as essentially state to state transformer functions (ie. semantics). We refer to Section 2.4.1 for details on the semantics interpretation function I ∎

The examples further show that functions, in addition to maps, offer essential means for abstraction ∎

2.3.6 *Discussion.* Several issues can be raised. Already now. Before having even covered operational abstractions ! Some are dealt with in Sections 2.3.6.1–2.3.6.4 immediately below. More will follow in Section 2.5 at the end of this major section (Section 2).

2.3.6.1 *Self–fulfilment.* Since we have, from mathematics, the mostly discrete concepts of sets, Cartesians, lists and maps, then why not make use of these concepts in software engineering ? So this is what we do. Then we can also avail ourselves of the informal, or even formal, mathematical reasoning techniques that come with using mathematics. And then the five *Principles cum Techniques* can be construed as self–serving, as having been individually worded so as to lead to the "obvious" conclusions.

2.3.6.2 *"Hedges".* We have double quoted some terms: "apt", "fittingly" and "appropriate". When one double quotes, it is usually a hedge, a way of expressing some doubt, or a way of saying things "tongue in cheek". We can not formalise aptness, fitting and appropriateness. That is: There is not objective measure that allows us to claim that one model choice is more apt, fitting or appropriate than another. It remains, after all, a matter of aesthetics !

2.3.6.3 *Sufficiency.* Who says that exactly these five kinds of aggregations: Sets, Cartesians, lists, maps and functions, are the first or only ones to suggest ? Again there is no hard, objective evidence other than their being firmly rooted in mathematics — where they obviously seems to have gained some currency. Another mathematical structure that could be suitable candidates are, or in fact were: Algebras. And algebras were our model choice when we aim at property oriented abstractions. Yet other mathematical structure choices could be: Categories, and many find that an exciting possibility, viz. [Goguen 1973; Burstall and Goguen 1977; Sannella and Tarlecki 1986; Rydeheard and Burstall 1991].. So "the jury is still out": There may be other good mathematical structures that could form basis for abstractions in software engineering.

2.3.6.4 *Historical Background.* There are two aspects here: *(I) Mathematics:* We have already, above, in Section 2.3.6.1, mentioned the essence of mathematics as a background for model oriented abstractions. *(II) Computing Science:* `VDM` (since 1973) [Bjørner and Jones 1978; Jones 1980; Bjørner and Jones 1982; Jones 1986; Jones 1990b; Fitzgerald and Larsen 1997] provided perhaps the first model oriented means for abstract specification. The programming language `SETL` [Schwartz 1973] (at about the same time), independently, provided sets as a basis for programming. Later `VDM` was joined by `Z` (around 1980) [Spivey 1988; Imperato 1991; McDermid and Whysall 1992; Spivey 1992; Wordsworth 1993; Diller 1994; Barden et al. 1994; Rann et al. 1994; Ratcliff 1994; Bottaci and Jones 1995; Potter et al. 1996; Woodcock and Davies 1996; Jacky 1997] and `B` (around 1990) [Abrial 1996b].

2.4 Mostly Model Oriented Operational Abstraction

In the previous section, Section 2.3, we have introduced principles for why to select and how to express model oriented abstractions of *types* and *values* of phenomena in either an actual or an software engineering world. These phenomena values do not exist in a vacuum. They exist, anthropomorphically speaking, because they are being manipulated by *operations,* being "created", "massaged", and "inspected" (queried) by functions and predicates. In this section we shall investigate principles and techniques for expressing, for defining, such operations — "more or less" abstractly !

We will discuss the issues of operational abstraction in the reverse order of a perceived "level of abstractness": Most "concrete" first ! We do so because we think that this will provide a best way of contrasting the three "styles" of function definitions: We believe that most readers will be most familiar with the explicit way of defining functions. Hence we start with that.

2.4.1 *Explicit Function Definitions.* Model oriented explicit function definitions are definitions which have a *function definition body* and where that *function definition body* expresses *calculations* over values of model oriented types such as integers, Booleans, sets, Cartesians, lists and maps.

Principle: We choose explicit function definitions, typically, when abstractly modelling, ie. defining the denotational or computational semantics of syntactic quantities. That is: When we wish to express denotations (through functional composition) or computations (through sequences of actions) ∎

Section 3 focuses on Denotational and Computational Semantics. The explicit function definition style, is of course, like a functional programming specification. So we use this style much more widely that the other styles.

Technique: Typically denotational and computational semantics definitions are expressed in a model oriented manner, and hence uses "up to" the full variety of model oriented techniques: Abstract set, Cartesian, list, map and function values, as well as all the corresponding primitive operators on sets, Cartesians, lists, maps and functions ∎

Examples: We take up the example of banking from Sections 2.3.4 and 2.3.5. In Section 2.3.5 we "promised" to detail the semantics of banking transactions.

$$
\begin{array}{l}
\mathrm{I(tran{:}mk\_Deposit(c,m))}(\xi,\alpha\mathrm{s},\pi)\mathrm{(t)} \equiv \\
\quad \mathbf{if}\ \mathrm{c} \in \mathbf{dom}\ \alpha\mathrm{s} \\
\quad\quad \mathbf{then} \\
\quad\quad\quad \mathbf{let}\ (\mathrm{m}',\mathrm{t}\ell,...)=\alpha\mathrm{s(c)},\mathrm{ca}=(\mathrm{m}'+\mathrm{m},\mathrm{t}\ell^\frown\langle(\mathrm{tran,t})\rangle,...)\ \mathbf{in} \\
\quad\quad\quad ((\xi,\alpha\mathrm{s} \dagger [\ \mathrm{c} \mapsto \mathrm{ca}\ ],\pi),\mathrm{ok})\ \mathbf{end} \\
\quad\quad \mathbf{else}\ ((\xi,\alpha\mathrm{s},\pi),\mathrm{nok}) \\
\quad \mathbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{I(tran{:}mk\_Withdraw(k,c,n))}(\xi,\alpha\mathrm{s},\pi)\mathrm{(t)} \equiv \\
\quad \mathbf{if}\ \mathrm{k} \in \mathbf{dom}\ \xi \wedge \mathrm{c} \in \xi(\mathrm{k}) \wedge \mathbf{let}\ (\mathrm{m},,...)=\alpha\mathrm{s(c)}\ \mathbf{in}\ \mathrm{largeq(m,mk\_M(n))}\ \mathbf{end} \\
\quad\quad \mathbf{then} \\
\quad\quad\quad \mathbf{let}\ (\mathrm{m}',\mathrm{t}\ell,...)=\alpha\mathrm{s(c)},\mathrm{ca}=(\mathrm{m}'-\mathrm{mk\_M(n)},\mathrm{t}\ell^\frown\langle(\mathrm{tran,t})\rangle,...)\ \mathbf{in} \\
\quad\quad\quad ((\xi,\alpha\mathrm{s} \dagger [\ \mathrm{c} \mapsto \mathrm{ca}\ ],\pi),\mathrm{mk\_M(n)})\ \mathbf{end} \\
\quad\quad \mathbf{else}\ ((\xi,\alpha\mathrm{s},\pi),\mathrm{nok}) \\
\quad \mathbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{I(tran{:}mk\_Transfer(fk,fc,n,tc))}(\beta)\mathrm{(t)} \equiv \\
\quad \mathbf{if}\ \mathrm{fk} \in \mathbf{dom}\ \xi \wedge \mathrm{fc} \in \xi(\mathrm{fk}) \wedge \mathbf{let}\ (\mathrm{m},,...)=\alpha\mathrm{s(fc)}\ \mathbf{in}\ \mathrm{largeq(fm,mk\_M(n))}\ \mathbf{end} \\
\quad\quad \wedge\ \mathrm{tc} \in \mathbf{dom}\ \alpha\mathrm{s} \\
\quad\quad \mathbf{then} \\
\quad\quad\quad \mathbf{let}\ (\mathrm{fm},\mathrm{ft}\ell,...) = \alpha\mathrm{s(fc)},\ (\mathrm{tm},\mathrm{tt}\ell,...) = \alpha\mathrm{s(tc)}, \\
\quad\quad\quad\quad \mathrm{fca} = (\mathrm{fm} - \mathrm{mk\_M(n)},\mathrm{ft}\ell^\frown\langle(\mathrm{tran,t})\rangle,...), \\
\quad\quad\quad\quad \mathrm{tca} = (\mathrm{tm} + \mathrm{mk\_M(n)},\mathrm{tt}\ell^\frown\langle(\mathrm{tran,t})\rangle,...)\ \mathbf{in} \\
\quad\quad\quad ((\xi,\alpha\mathrm{s} \dagger [\ \mathrm{fc} \mapsto \mathrm{fca},\mathrm{tc} \mapsto \mathrm{tca}\ ],\pi),\mathrm{ok})\ \mathbf{end} \\
\quad\quad \mathbf{else}\ ((\xi,\alpha\mathrm{s},\pi),\mathrm{nok}) \\
\quad \mathbf{end}
\end{array}
$$

$$\mathrm{largeq}\colon \mathrm{M} \times \mathrm{M} \to \mathbf{Bool}$$

$\text{largeq}(\text{mk\_M}(n),\text{mk\_M}(n')) \equiv n \geq n'$

It will not work to define the latter essentially in terms of function composition as that would not record the proper transactions on the transaction files — as required by some requirements statement:

$$
\begin{array}{l}
\text{I}(\text{tran:mk\_Transfer}(\text{fk},\text{fc},\text{n},\text{tc}))(\beta)(\text{t}) \equiv \\
\quad \textbf{if } \text{fk} \in \textbf{dom } \xi \wedge \text{fc} \in \xi(\text{fk}) \wedge \textbf{let } (\text{m},,...)=\alpha\text{s}(\text{fc}) \textbf{ in } \text{largeq}(\text{fm},\text{mk\_M}(\text{n})) \textbf{ end} \\
\qquad \wedge\ \text{tc} \in \textbf{dom } \alpha\text{s} \\
\qquad \textbf{then} \\
\qquad\quad \textbf{let } (\beta',\text{r}) = \text{I}(\text{mk\_Withdraw}(\text{fk},\text{fc},\text{n}))(\beta)(\text{t}) \textbf{ in} \\
\qquad\quad \text{I}(\text{mk\_Deposit}(\text{tc},\text{r}))(\beta')(\text{t}) \textbf{ end} \\
\qquad \textbf{else } (\beta,\text{nok}) \\
\quad \textbf{end}
\end{array}
$$

Had we omitted the "proper recording of transactions" requirements, then we could have almost had a proper compositional semantics ∎

2.4.2 *Pre/Post Function Definitions.* Model oriented pre/post function definitions are definitions which express pre conditions (ie. properties) on the function input arguments, and post conditions (ie. properties) between the function input arguments and the result values — where these properties are expressed as properties over values of model oriented types such as integers, Booleans, sets, Cartesians, lists and maps. Thus we can expect that the explicit and the pre/–post definition styles relate closely.

Principle: There are often two, independent reasons, not just one good reason, for using the pre/–post function definition style:

* One, the "good" one, concerns what it is we wish to express. And if we wish to express proof oriented properties, but remain in a model oriented specification style, the we choose the pre/–post function definition style.
* The other, the independent (thereof) reason is abstraction: If we can achieve a shorter, more concise specification that with an explicit definition, then, again, we choose the pre/–post function definition style.

We observe that the two reasons are independent. That both are pragmatically determined. That the former pertains to "future use" of the definitions, and the latter to "present comprehension" of the same ∎

Technique: Since the pre/–post function definition style, as defined above, is expressed in the model oriented style it "inherits", as did the explicit function definition style, its techniques from those of the model oriented abstraction and modelling style. In addition, the uses of this style are, usually, in terms of the conjunction of several Boolean propositions as shown in the next example ∎

Example: We shall give two examples that illustrate the above "spectrum" of choices.

* Bank Transactions:
  This example primarily illustrates the "proof oriented" reason for choosing the pre/–post function definition style.

To express, by pre/post conditions, the semantics of the banking transactions explicitly defined in the previous section, we introduce a few auxiliary functions. First we recall the Bank configuration:

type
  K, C, T, Tran
  M == km_M(n:**Nat**)
  $B' = SHA \times ACC \times PEN$
  B = {| b:B′ • wf_B(b) |}
  SHA = K $\overrightarrow{m}$ C-**set**
  ACC = C $\overrightarrow{m}$ A
  A :: sm:M × stℓ:TL × ...
  $TL' = TranT^*$
  TranT :: str:Tran × st:T
  TL = {| tℓ:TL′ • wf_TL(tℓ) |}
value
  wf_TL: TL → **Bool**
  wf_TL(tℓ) ≡ ∀ i,j:**Nat** • {i,j} ⊆ **inds** tℓ⇒ i < j ⇒ st(tℓ(i)) < st(tℓ(j))
  fst: A $\tilde{\rightarrow}$ TranT*, lst: A $\tilde{\rightarrow}$ TranT
  fst(m,tℓ⌢⟨tt⟩) ≡ tℓ
  lst(m,tℓ⌢⟨tt⟩) ≡ tt

value
  I(tran:mk_Deposit(c,m))(ξ,αs,π)(t) **as** ((ξ′,αs′,π′),r)
    pre c ∈ **dom** αs ∧
    post
      dom αs′ = **dom** αs ∧ sm(αs′(c)) = sm(αs(c))+m ∧
      fst(stℓ(αs′(c))) = stℓ(αs(c)) ∧ lst(stℓ(αs′(c))) = ⟨(tran,t)⟩ ∧ ... ∧
      ∀ c′:C • c ≠c′ ⇒
        sm(αs′(c′)) = sm(αs(c′)) ∧ stℓ(αs′(c′)) = stℓ(αs(c′)) ∧ ... ∧
      ξ = ξ′ ∧ π = π′ ∧ r=ok

  I(tran:mk_Withdraw(k,c,n))(ξ,αs,π)(t) **as** ((ξ′,αs′,π′),r)
    pre k ∈ **dom** ξ ∧ c ∈ ξ(k) ∧
      largeq(sm(αs(fc)),mk_M(n))
    post
      dom αs′ = **dom** αs ∧ sm(αs′(c)) = sm(αs(c))−m ∧
      fst(stℓ(αs′(c))) = stℓ(αs(c)) ∧ lst(stℓ(αs′(c))) = ⟨(tran,t)⟩ ∧ ... ∧
      ∀ c′:C • c ≠c′ ⇒
        sm(αs′(c′)) = sm(αs(c′)) ∧ stℓ(αs′(c′)) = stℓ(αs(c′)) ∧ ... ∧
      ξ = ξ′ ∧ π = π′ ∧ r=mk_M(n)

  I(tran:mk_Transfer(fk,fc,n,tc))(ξ,αs,π)(t) **as** ((ξ′,αs′,π′),r)
    pre fk ∈ **dom** ξ ∧ fc ∈ ξ(fk) ∧ tc ∈ **dom** αs
      ∧ largeq(sm(αs(fc)),mk_M(n))
    post
      dom αs′ = **dom** αs ∧

$\mathrm{sm}(\alpha s'(fc))=\mathrm{sm}(\alpha s(fc))-\mathrm{mk\_M}(n)/\ \mathrm{sm}(\alpha s'(tc))=\mathrm{sm}(\alpha s(tc))+\mathrm{mk\_M}(n) \wedge$
$\mathrm{fst}(\mathrm{st}\ell(\alpha s'(fc))) = \mathrm{st}\ell(\alpha s(fc)) \wedge \mathrm{lst}(\mathrm{st}\ell(\alpha s'(fc))) = \langle(\mathrm{tran},t)\rangle \wedge ... \wedge$
$\mathrm{fst}(\mathrm{st}\ell(\alpha s'(tc))) = \mathrm{st}\ell(\alpha s(tc)) \wedge \mathrm{lst}(\mathrm{st}\ell(\alpha s'(tc))) = \langle(\mathrm{tran},t)\rangle \wedge ... \wedge$
$\forall\ c':C \bullet c' \notin \{fc,tc\} \Rightarrow$
$\quad \mathrm{sm}(\alpha s'(c')) = \mathrm{sm}(\alpha s(c')) \wedge \mathrm{st}\ell(\alpha s'(c')) = \mathrm{st}\ell(\alpha s(c')) \wedge ... \wedge$
$\xi = \xi' \wedge \pi = \pi' \wedge r=ok$

Observe how we studiously avoid explicit use of model oriented (notably set and list) operations in the post conditions as compared to the explicit function definition bodies of the example of Section 2.4.1. Trying to avoid that forced us to introduce the auxiliary (first and last) functions, as well as equipping the Cartesian types with selectors (sm, stℓ). We attempt this avoidance in order to emphasise properties, without being "fully, abstractly" axiomatic ▪

* A Simple Telephone Exchange:
This example[9] primarily illustrates the elegance and abstraction reason (conciseness) for choosing the pre/–post function definition style.
There are the notions of: Subscribers, s:S; connections, c:C, between two or more subscribers; actual, a:A, and requested, r:R, connections between two or more subscribers; and a telephone exchange system, x:X, with actual and requested connections such that all actual connections are requested.

type
  S
  C = {| ss | ss:S-**set** • **card** ss ≥ 2 |}
  R = C-**set**
  A = C-**set**
  X = {| (r,a) | (r,a):R×A • a ⊆ r ∧ ⋂ a = {} |}

There is a notion of telephone exchange system efficiency, a constraint that governs its operation, hence the state, at any one time. The efficiency criterion says that all requested call that can actually be connected are so:

value
  eff_X: X $\xrightarrow{\sim}$ **Bool**
  eff_X(r,a) ≡ ∼∃ a′:A • a ⊂ a′ ∧ (r,a′) ∈ X

Now there is the notion of subscriber actions: making a (possibly multi-party) call, terminating (hanging up on) a call, and inquiring whether a line is busy. Let us model them as if they were the denotation of commands "being executed" in a state *x:X*.

type
  Cmd == Call | Hang | Busy
  Call′ = mk_Call(p:S,cs:C)

[9] The example is "borrowed" from *J.C.P. Woodcock and M. Loomes'* book *Software Engineering Mathematics, Pitman, London, 1988* [Woodcock and Loomes 1988]. Permission to "lift" this example (converting it, however, from Z into RAISE, and providing it, with a property oriented solution) has yet to be formally applied for. It has been informally granted, at Uppsala University, Sweden, verbally by J.C.P. Woodcock (February 28, 2001).

$$\begin{array}{l}
\text{Call} = \{\mid \text{c:Call}' \bullet \textbf{card}\ \text{cs(c)} \geq 1 \} \\
\text{Hang} = \text{mk_Hang(s:S)} \\
\text{Busy} = \text{mk_Busy(s:S)}
\end{array}$$

A multi-party call involves a (primary, s) caller and one or more (secondary, ss) callers. Enacting such a call makes the desired connection requested. If none of the callers are already engaged in an actual connection then the call can be actualised. A multi-party call cannot be made by a caller who has already requested "other" calls.

$$\begin{array}{l}
\textbf{value} \\
\quad \text{int_Call: Call} \stackrel{\sim}{\rightarrow} \text{X} \stackrel{\sim}{\rightarrow} \text{X} \\
\quad \text{int_Call(mk_Call(p,cs))(r,)}\ \textbf{as}\ (\text{r}',\text{a}') \\
\quad \textbf{pre}\ \text{p} \notin \bigcup \text{r} \\
\quad \textbf{post}\ \text{r}' = \text{r} \cup \{\{\text{p}\} \cup \text{cs}\} \wedge \text{eff_X(r}',\text{a}')
\end{array}$$

It takes one person, one subscriber, to terminate a call.

$$\begin{array}{l}
\textbf{value} \\
\quad \text{int_Hang: Hang} \rightarrow \text{X} \stackrel{\sim}{\rightarrow} \text{X} \\
\quad \text{int_Hang(mk_Hang(p))(r,a)}\ \textbf{as}\ (\text{r}',\text{a}') \\
\qquad \textbf{pre}\ \text{exist c:C} \bullet \text{c} \in \text{a} \wedge \text{p} \in \text{a} \\
\qquad \textbf{post}\ \text{r}' = \text{r} \setminus \{\text{c}|\text{c:C} \bullet \text{c} \in \text{r} \wedge \text{p} \in \text{c}\} \wedge \text{eff_X(r}',\text{a}')
\end{array}$$

A line (that is: a subscriber) is only 'busy' if it (the person) is engaged in an actual call.

$$\begin{array}{l}
\textbf{value} \\
\quad \text{int_Busy: S} \rightarrow \text{X} \stackrel{\sim}{\rightarrow} \textbf{Bool} \\
\quad \text{int_Busy(mk_Busy(p))(,a)}\ \textbf{as}\ \text{b} \\
\qquad \textbf{pre true} \\
\qquad \textbf{post if}\ \text{b}\ \textbf{then}\ \text{p} \in \bigcup \text{a}\ \textbf{else}\ \text{p} \notin \bigcup \text{a}\ \textbf{end}
\end{array}$$

We challenge (as a very simple exercise) the reader to write the explicit function definitions for the above telephone transactions. The reader will then find that the explicit function definitions for all the above except the last require more text than the pre/–post conditions ▪

We trust that the reader, through careful reading of these two examples and the text of the *principle* that enunciates the pre/–post function definition style, can extract the claimed relations ▪

2.4.3 *Axiomatic Function Definitions.* Although this kind of function definition was already exemplified, and most properly belongs under the property oriented abstractions, we review it here, in the context of "then other two" function definition styles — just exemplified. We do so in order to best (we believe) contrast these (altogether three) styles. (Besides, we actually did not present any proper function definitions in Section 2.2.2 !)

Axiomatic function definitions are typically algebra oriented presentations, that is: definitions, which express relations, usually in an equational logic manner, over "combinations" of applications of observer and generator functions. These functions usually only assume abstract data type, ie. sort, defined values.

Principles: We refer back to the first principles of this Section 2, those, in Section 2.2.3, of *Separation of Concerns*, of *Algebras* and *Algebraic Specification* ∎

Technique: The "combinations", alluded to, usually express equalities between function application terms where the "outermost" function is an observer function and "innermost" functions are generator functions ∎

Example: In a simpleminded patient medical record "world" — as we find it in actual life — time, T, stamped information, I, is inserted, initially into empty, later into non–empty records, M. Such inserted information can be retrieved, in this simpleminded example, by search based on the time of an insertion argument. No two separately inserted pieces of information can have the same time stamp.

type
  T, I, M
value
  $\mathrm{obs\_T}\colon \mathrm{I} \rightarrow \mathrm{T}$
  $\mathrm{cre}\colon \rightarrow \mathrm{M}$
  $\mathrm{emp}\colon \mathrm{M} \rightarrow \mathbf{Bool}$
  $\mathrm{ins}\colon \mathrm{I} \times \mathrm{M} \rightarrow \mathrm{M}$
  $\mathrm{sea}\colon \mathrm{T} \times \mathrm{M} \stackrel{\sim}{\rightarrow} \mathrm{I}$
  $\mathrm{con}\colon \mathrm{T} \times \mathrm{M} \rightarrow \mathbf{Bool}$

cre, emp, ins, sea, and con, are operations that designate creation (cre) of initially empty (emp) records, testing for emptiness (emp) of such, inserts (ins) information into records, searches (sea), by time, for possibly "previously" (than the search operation) inserted such time stamped information, respectively inquires whether the exists to information in a record which has (is of) the given time stamp.

Let us first define invariants on the medical record "structure":

axiom
  $\forall\ \mathrm{t{:}T,i,i'{:}I,m{:}M} \bullet$
    $\sim\mathrm{con(t,emp())} \wedge$
    $\mathrm{con(t,ins(i,m))} \equiv \mathrm{t{=}obs\_T(i)} \vee \mathrm{con(t,m)} \wedge$
    $\forall\ \mathrm{ins(i'',ins(i''',m')){:}M} \bullet \mathrm{obs\_T(i'')} \neq \mathrm{obs\_T(i''')} \wedge$
    $\mathrm{ins(i,ins(i',m))} = \mathrm{ins(i',ins(i,m))}$

Let us then define the operations:

axiom
  $\forall\ \mathrm{t{:}T,i{:}I,m{:}M} \bullet$
    $\mathrm{emp(cre())} \wedge$
    $\sim\mathrm{emp(ins(i,m))} \wedge$
    $\mathrm{con(t,ins(i,m))} \wedge \sim\mathrm{con(t,m)} \Rightarrow \mathrm{sea(t,ins(i,m)){=}i}$

The last axiom could be expressed in alternative ways ([1,2]):

axiom
  $\forall\ \mathrm{t{:}T,i{:}I,m{:}M} \bullet$
[1] $\mathrm{t{=}obs\_T(i)} \Rightarrow \mathrm{sea(t,ins(i,m)){=}i}$

[2] con(t,ins(i,m)) $\wedge$ t=obs_T(i) $\Rightarrow$ sea(t,ins(i,m))=i

Which way is chosen is sometimes a matter of taste ! ∎

2.4.4 *Discussion.* We see, as earlier expressed, how a conventional algebraic semantics presentation (as usually deployed for property oriented formal descriptions) — how such a definition — expresses the properties of generator and observer functions, as well as of the values to which they apply, in terms of an interplay between all three: As simple terms or equations between terms, where terms usually is an observer function being applied to a terms which contains a generator function expression. This is in contrast to the more "free form" axioms that depend on model oriented uses of, for example, sets, Cartesians, and lists. We illustrated the latter, the "free form" axioms, in the value (ie. type) invariants of Section 2.2.2 and in the pre/–post condition definitions of functions in Section 2.4.2.

Principle: As a matter of principle we leave it up to the definer to choose which of these axiomatic forms to deploy ∎

A judicious use of model oriented types often "softens", makes more palatable, the more frugal uses of "non free form" axioms (expressed as equations between values of functional terms).

2.5 Discussion

We continue the line of observations started in Section 2.3.6 (for representational abstractions) and continued in Section 2.4.4 (for operational abstractions). The current, final observations apply to the spectrum of property oriented and model oriented abstractions.

2.5.1 *Understanding.* Property oriented specifications, usually emphasise logic. Many uses of logic tends to be formalistic, ie. reduced, or reduceable, to "playing" with symbols. Such property oriented specifications tend, to some readers, to be "too abstract" — in the sense of not being that easily understandable. Mathematics without understanding may be OK, but some say no. Software engineering without understanding "what is going on", without being able to firmly, although not formally, to relate "what is going on" amongst — what is being expressed by — the formulas with respect to an actual, or an algorithmic, world, is not OK ! So that may be why model oriented specification is preferable in many cases. Mathematics started with Geometry. And one could "see" geometry ! We claim that through appropriate textual and graphical notation (the latter is not shown in this paper) one can "see & feel" sets, Cartesians, lists, maps, and — albeit only to some rather simplified extent also — functions. It is somehow harder to see the import of untyped axioms — hence we tend to favour typed axioms and allowing simple-minded uses of for example set and list types and values in axiomatically presented property oriented specifications.

The example of Section 2.4.3 is a case in point. If the example had not been that simple, and if we had not attached the informal description, or remarks, "around" that example, some might claim that it was not that "obvious" as to what was going on.

2.5.2 *Utter Simplicity & Elegance vs. Algorithmitisation.* It seems to us that one should never embark on any software development project, whatever its final scale is, without initially being able to express the essence of that software, its rooting in some domain, on a page or two. Anything larger than a page or two may not be comprehensible, may not be understandable. And if one cannot understand the very bases that software applies to, then serious doubt ought be raised as to whether one can have any trust in any resulting software. It seems to us that expressing the very intrinsics of domains in the property oriented, ie. algebraic way, helps one to achieve, and induces, utter simplicity & elegance.

We would claim that the example of Section 2.4.3 represents a reasonable example of utter simplicity, and thus, to us, the elegance we are striving for.

2.5.3 *Representational and Operational Abstraction is not Enough !.* For us to achieve the utter simplicity & elegance — of general specifications — as "announced" in the previous section, it is not enough to just secure representational and operational abstraction. The next sections will present a number of independent, ie. "orthogonal" criteria by which to structure and express a general specification. When we say 'general specification', we mean it to distinguish the specification from specific domain, or requirements, or software design specifications. For these latter, specific, specifications, further abstraction and modelling criteria can be offered. For domain and for requirements specifications we present such in [Bjørner 2001a] and [Bjørner 2001j].

3. DENOTATIONS AND COMPUTATIONS

3.1 Semantics of Specifications

3.2 Denotations

3.2.1 *Valuation versus Denotation*

3.2.2 *Interpretation, Evaluation and Elaboration*

3.2.3 *Discussion*

3.3 Computations

3.3.1 *Structural Operational Semantics*

3.3.2 *Mechanical Semantics*

3.3.3 *Discussion*

3.4 Discussion

4. HIERARCHIES AND COMPOSITIONS

4.1 Development versus Presentation

4.1.1 *"Top–down versus Bottom–up"*

4.1.2 *Development*

4.1.3 *Presentation*

4.1.4 *Discussion*

4.2 Hierarchies

4.2.1 *The Hierarchy Concept*

4.2.2 *Hierarchy Examples*

4.2.3 *Discussion*

4.3 Compositions

4.3.1 *The Composition Concept*

4.3.2 *Composition Examples*

4.3.3 *Discussion*

4.4 Blended Styles

4.5 Discussion

5. CONFIGURATIONS: CONTEXTS AND STATES

5.1 The Problem

5.1.1 *Static Contexts*

5.1.2 *Dynamic States*

5.1.3 *Discussion*

5.2 Example I: Language Specification

5.2.1 *Semantic Values*

5.2.2 *Syntactic Values*

5.2.3 *SemanticFunctions*

5.2.4 *Discussion*

5.3 Example II: A Reactive System

5.3.1 *Reactive System Context*

5.3.2 *Reactive System State*

5.3.3 *Reactive System Transition Function*

5.3.4 *Discussion*

5.4 Discussion

6. TIME AND SPACE, PARTS AND WHOLES, SYSTEMS AND LANGUAGES

6.1 Ontologies of Time

TO BE WRITTEN

6.1.1 *The Problem*

TO BE WRITTEN

6.1.2 *Time — A Possible Axiomatic Approach.* Following [van Benthem 1991] we assume some point structure $p, p', p'' : P$ with some ordering relation $<$.

Time is transitive: if $p{<}p'$ and $p'{<}p''$ then $p{<}p''$. Time may not loop, is not reflexive: $p \not< p$. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time. Time can be left-linear, ie. linear "to the left" of a given time. One could designate a time axis as beginning at some time, that is: having no predecessor times. And one can designate a time axis as ending at some time, that is: having no successor times. General, past and future successors (predecessors, respectively successors in daily talk) can be fined. Time can be dense: Given any two times one can always find a time between them. Discrete time can be defined:

axiom
[TRANS: Transitivity]
$\forall$ p,p′,p″:P • p < p′ < p″ ⇒ p < p″
[IRREF: Irreflexitivity]
$\forall$ p:P • p $\not<$ p
[LIN: Linearity]
$\forall$ p,p′:P • (p=p′ ∨ p<p′ ∨ p>p′)
[L−LIN: Left Linearity]
$\forall$ p,p′,p″:P • (p′<p ∧ p″<p) ⇒ (p′<p″ ∨ p′=p″ ∨ p″<p′)
[BEG: Beginning]
∃ p:P • ∼∃ p′:P • p′<p
[END: Ending]
∃ p:P • ∼∃ p′:P • p<p′
[SUCC: Successor]
[PAST: Predecessors]
$\forall$ p:P,∃ p′:P • p′<p
[FUTURE: Successor]
$\forall$ p:P,∃ p′:P • p<p′
[DENS: Dense]
$\forall$ p,p′:P (p<p′ ⇒ ∃ p″:P • p<p″<p′)
[DENS: Converse Dense] ≡ [TRANS: Transitivity]
$\forall$ p,p′:P (∃ p″:P • p<p″<p′ ⇒ p<p′)
[DISC: Discrete]
$\forall$ p,p′:P • (p<p′ ⇒ ∃ p″:P • (p<p″ ∧ ∼∃ p‴:P • (p<p‴<p″)))
∧
$\forall$ p,p′:P • (p<p′ ⇒ ∃ p″:P • (p″<p ∧ ∼∃ p‴:P • (p″<p‴<p′)))

An SPO, Strict Partial Order, is a point structure satisfying TRANS and IRREF. TRANS, IRREF, and SUCC implies infinite models. TRANS and SUCC may have finite, "looping time" models.

6.1.3 *Discussion*

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

6.2 A Mereology of Parts and Wholes

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

6.2.1 *The Problem*

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

6.2.2 *Parts & Wholes — A Possible Axiomatic Approach.* Axioms (0–5): An incomplete ontology of mereology. Axiom (0): A base axiom. Axiom (1): The Mereology defines a notion of individuals. These are later to be made into components (in the preceding sometimes also referred to as parts — and therefore also here), and systems. Axioms (2–6): Defines proper parts, immediate proper, ie. direct parts, disjointness of parts, and atomic (or simple) individuals (ie. parts). Axioms (2–3): Assymmetry and transitivity. Axiom (6): Atomic individuals.

axiom
$\forall$ i,i′,i″:I •
  (0) equal(i,i′): **if** i ″`equal`″ i′ **then true else false end**
  (1) is_individual(i) $\equiv$ equal(i,i)
  (2) proper_part(i,i′) $\Rightarrow$ ∼proper_part(i′,i)
  (3) proper_part(i,i′) $\wedge$ proper_part(i′,i″) $\Rightarrow$ proper_part(i,i″)
  (4) direct_part(i,i′) $\equiv$
    proper_part(i,i′) $\wedge$ ∼$\exists$ i‴:I • proper_part(i‴,i′) $\wedge$ proper_part(i,i‴)
  (5) disjoint(i,i′) $\equiv$
    ∼(equal(i,i′) $\vee$ $\exists$ i″:I • proper_part(i″,i) $\wedge$ proper_part(i″,i′))
  (6) simple_individual(i) $\equiv$ is_individual(i) $\wedge$ ∼$\exists$ i″:I • proper_part(i″,i)

6.2.3 *Discussion*

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

6.3 Towards an Ontology of Spatial Matters

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

6.3.1 *The Problem*

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

6.3.2 *Topology — A Possible Axiomatic Approach.* We continue the axiom system of mereology of Section 6.2.2.

Axioms (0–10) define an ontology of topology. Axioms shall serve to express: Two individuals can be connected; multiple connectors between individuals must be possible.

axiom
$\forall$ i,i′,i″:I,c,c′:C •

/∗ connector is co ∗/
  (7) $\mathrm{co(c)} \equiv \exists\ \mathrm{i''',i'''':I\bullet\sim e(i''',i'''')\wedge cs(c,i''')\wedge cs(c,i'''')}$
/∗ connects is cs ∗/
  (8) $\mathrm{cs(c,i)} \wedge \mathrm{cs(c,i')\Rightarrow\sim(pp(i,i')\vee pp(i',i))}$
  (9) $\mathrm{cs(c,i)\wedge cs(c,i')\wedge pp(i,i'')\wedge d(i'',i')\Rightarrow cs(c,i'')}$
  (10) $\mathrm{cs(c,i)\wedge cs(c,i')\wedge cs(c,i'')\Rightarrow\sim(d(i,i')\wedge d(i',i'')\wedge d(i'',i))}$

The axioms, cf. Figure 3, express: Axiom (7): Connectors connects something. Axiom (8): An individual cannot be connected to its whole. Axiom (9): An individual whose whole is disjoint with an individual connected to the part has also its whole connected to that individual. Axiom (10): Connectors cannot fork.

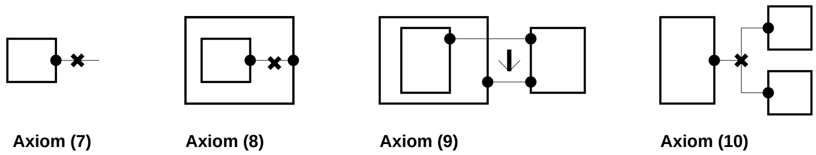

Fig. 3. Topology: 7, 8 and 10 Disallowed ($\times$); 9: Upper implies ($\downarrow$) Lower

6.3.3 *Discussion*

6.4 Spatial Phenomena

We start or abstract modelling of actual world uses of time and space by first exemplifying *spatial phenomena*. We have chosen to do so, reverting the order: Time and space, since the examples of *temporal* phenomena, in these examples, rely on their spatial "counterparts".

6.4.1 *General.* We are not interested in abstractions of the trivial examples of geometric phenomena such as for example the spherical co–ordinates, or the three dimensional modelling of mechanical parts. The examples to be shown go well beyond such conventional CAD/CAM applications.

6.4.2 *Example I: Airspace.* We present a very simple model of air space.

6.4.2.1 *Narrative.* An air space (AS), consists of airports (A), that have unique names (An). Each airport is, amongst other things, characterised by an airdome (D). From each airport there leads a number of uniquely named (Wn), air corridors (W), to other (named) airports.

6.4.2.2 *Formal Model.* A simple model is presented. D and W stand for dense point spaces.

type
  A, An, D, W, Wn, V, S
  $\mathrm{AS'} = \mathrm{A}\ \overrightarrow{_{m}}\ (\mathrm{D} \times (\mathrm{An}\ \overrightarrow{_{m}}\ (\mathrm{Wn}\ \overrightarrow{_{m}}\ \mathrm{W})))$
  $\mathrm{AS} = \{|\ \mathrm{airspa{:}AS'} \bullet \mathrm{wf\_AS(airspa)}\ |\}$
value
  obs_An: $\mathrm{A} \rightarrow \mathrm{An}$

```
   wf_AS: AS′ → Bool
   wf_AS(airspa) ≡
      ∀ a:A • a ∈ dom airspa ⇒
         let oan = obs_An(a), (od,aws) = airspa(oan) in
         is_contained(a,od) ∧
         ∀ dan:An • dan ∈ dom aws ⇒
            dan ∈ dom airspa ∧
            ∀ wn:Wn • wn ∈ dom aws(dan) ⇒
               let (dd,) = airspa(dan), w = (aws(dan))(wn) in
               is_smooth(w) ∧ is_wf_connect(od,w,dd) ∧
         end end
   is_contained: A × D → Bool
   is_smooth: W → Bool
   is_wf_connect: D × W × D → Bool
axiom
```

6.4.3 *Example II: Railway Net.* We present a very simple model of rail nets.

6.4.3.1 *Narrative.* A rail net (N) consists of lines (L) and stations (S), and these again of rail units (U). Units are connected by connectors (C), and certain pairs of such connectors designate paths (P) through units.

6.4.3.2 *Formal Model.* We present just "barebone" aspects.

```
type
   N, L, S, U, C, P = C×C
value
   obs_Ls: N → L-set
   obs_Ss: N → S-set
   obs_Us: (N|L|S) → U-set
   obs_Cs: U → C-set
   obs_Ps: U → P-set
```

A Bezier curve is determined by a small set of (x, y, z) co–ordinates, and such a set can be observed from any rail unit. For a given path of a unit one can observe (ie. compute) its length. The left and right rails of a path may be inclined (ie. of different z co–ordinates), and a list of reals can be observed from any such path of a unit — with that list giving equidistantly spaced inclinations.

```
type
   Bezier
value
   obs_Bezier: U × P → Bezier
   obs_Length: U × P → Nat
   obs_Incl: U × P → Real*
```

Two connected units are smoothly connected, ie. share same tangent.

```
value
```

is_smooth_connection: U × U
is_smooth_connection(u,u′) ≡
  let {c} = obs_Cs(u) ∩ obs_Cs(u′) **in**
  let tan = tangent(c,u), tan′ = tangent(c,u′) **in** tan=tan′ **end end**
  pre: **card**(obs_Cs(u) ∩ obs_Cs(u′)) = 1

6.4.4 *Discussion*

TO BE WRITTEN

6.5 Temporal Phenomena

TO BE WRITTEN

6.5.1 *General*

TO BE WRITTEN

6.5.2 *Example: Air Traffic.* We present a very simple model of air traffic.

6.5.2.1 *Narrative.* Air traffic is a continuous function from time to pairs of air spaces and locations of flights.

6.5.2.2 *Formal Model.* F models flights (ie. aircrafts), and L their locations. Continuity is expressed by closeness predicates on pairs of times, respectively locations.

type
  T, F, L, Fn
  TF′ = T → (AS × (F $\overrightarrow{m}$ L))
  TF = {| tf:TF′ • wf_TF(tf) |}
value
  wf_TF: TF′ → **Bool**
  wf_TF(tf) ≡
    ∀ t,t′:T • {t,t′}⊆**dom** tf ⇒
      let (airspa,fls) = tf(t), (airspa′,fls′) = tf(t′) **in**
      ∀ f,f′:F • f ∈ **dom** fls ∧ f′ ∈ fls′ ⇒ obs_Fn(f)=obs_Fn(f′) ⇒
        ∀ t″:T • t<t″<t′ ⇒ t″ ∈ **dom** tf ∧
          let (airspa″,fls″) = tf(t″) **in**
          ∀ f″:F • obs_Fn(f″)=obs_Fn(f) ⇒ f″ ∈ **dom** fls″ ∧
          close(t,t″) ⇒ close(fls(f),fls″(f″)) **end end**
  obs_Fn: F → Fn
  close: T×T → **Bool**, close: L×L → **Bool**

6.5.3 *Discussion*

TO BE WRITTEN

6.6 Monotonicity and Continuity

TO BE WRITTEN

6.7 Combinations of Time and Space

TO BE WRITTEN

6.7.1 *Co–Existence of Time and Space — A Possible Axiomatic Approach.* We shall present an axiom system (Wayne D. Blizard, 1980 [Blizard 1990]) which relate abstracted entities to spatial points and time. Let $A, B, \ldots$ stand for entitites, $p, q, \ldots$ for spatial points, and t, τ for times. 0 designates a first, a begin time. Let t' stand for the discrete time successor of time t. Let $N(p,q)$ express that p and q are spatial neighbours. Let $=$ be an overloaded equality operator appicable, pairwise to entities, spatial locations and times, respectively. A_p^t expresses that entity A is at location p at time t. The axioms:

$$
\begin{array}{lrcll}
(I) & \forall A \forall t \exists p & : & A_p^t & \\
(II) & (A_p^t \wedge A_q^t) & \supset & p = q & \\
(III) & (A_p^t \wedge B_p^t) & \supset & A = B & \\
(IV) & (A_p^t \wedge A_p^{t'}) & \supset & t = t' & \\
(V\ i) & \forall p, q & : & N(p,q) \supset p \neq q & \text{Irreflexivity} \\
(V\ ii) & \forall p, q & : & N(p,q) = N(q,p) & \text{Symmetry} \\
(V\ iii) & \forall p \exists q, r & : & N(p,q) \wedge N(p,r) \wedge q \neq r & \text{No isolated locations} \\
(VI\ i) & \forall t & : & t \neq t' & \\
(VI\ ii) & \forall t & : & t' \neq 0 & \\
(VI\ iii) & \forall t & : & t \neq 0 \supset \exists \tau : t = \tau' & \\
(VI\ iv) & \forall t, \tau & : & \tau' = t' \supset \tau = t & \\
(VII) & A_p^t \wedge A_q^{t'} & \supset & N(p,q) & \\
(VIII) & A_p^t \wedge B_q^t \wedge N(p,q) & \supset & \sim (A_q^{t'} \wedge B_p^{t'}) &
\end{array}
$$

II–IV,VII–VIII The axioms are universally 'closed'.
We have omitted the usual $\forall A, B, p, q, t$s.

(I) For every entity, A, and every time, t, there is a location, p, at which A is located at time t.

(II) An entity cannot be in two locations at the same time.

(III) Two distint entities cannot be at the same location at the same time.

(IV) Entities always move: An entity cannot be at the same location at different times.

(V) These three axioms define N.

- **(V i)** Same as $\forall p :\sim N(p,p)$. Being a neighbour of is the same as being distinct from.
- **(V ii)** If p is a neighbour of q, then q is a neighbour of p.
- **(V iii)** Every location has at least two distinct neighbours.

(VI) The next four axioms determine the time successor function $'$.

- **(VI i)** A time is always distinct from its successor: time cannot rest. There are no time fix points.
- **(VI ii)** Any time successor is distinct from the begin time. Time 0 has no predecessor.
- **(VI iii)** Every non-begin time has an immediate predecessor.
- **(VI iv)** The time successor function $'$ is a one-to-one (ie. a bijection) function.

(VII) The *continuous path axiom:* If entity A is at location p at time t, and it is at location q in the immediate next time (t'), then p and q are neighbours.

(VIII) *No "switching":* If entities A and B occupy neighbouring locations at time t them it is not possible for A and B to have switched locations at the next time (t').

MORE TO COME

6.7.2 *Trading–off Between Time and Space*

TO BE WRITTEN

6.8 On Notions of Systems

The term 'system' is an elusive one. It is being used in all kinds of contexts. It is not always clear whether those who use this term have themselves properly understood it. That is: They usually fail to give a proper definition, one that allows the speaker or listener, the write or reader, to unambiguously determine whether something presented is a system or not.

6.8.1 *The Problem.* So that is the problem: To give a proper definition which allows us to decide, objectively, whether something is a system or not.

We feel confident that we can properly delineate mechanical systems: They are built up from indivisible mechanical components. These components are either welded, bolted, hinged, or clamped together, or they are otherwise connected by chained or directly fitted cogwheels, by prismatic or revolutionary joints, or by axes — as for robot arms. Thus we can make a distinction between componments and connectors (which, in a sense are also components). Typical examples of such mechanical systems are: An automobile's transmission system, a robot, a bicycle.

We feel less confident when a system is partly mechanical, partly materials flow oriented, partly electrical, as is for example a complete automobile. Here the gasoline tank, pipes, carburator, etc., interfaces with the engine, and the engine with the transmission system.

We feel even less confident about how to properly delineate a system when it is a software system: Which are the components, and which are the connectors ? Perhaps it is alright to claim that modules with procedures (classes with methods) are example components, and that sequential juxtaposition as well as procedure calling conventions and for example the CSP input/output synchronisation and communication primitives constitute connectors.

Add to all this the desire to group compositions of components into sub–systems, and these again into (sub–)systems, etc. Now which are the connectors between sub–system. And must all connectors connect components or (sub–)systems ?

6.8.2 *An Ontology of Systems.* Based on the above Problem statements we now present an axiom system for what certainly would qualify as adequate to cover mechanical systems.

6.8.2.1 *Systems.* We continue the axiom system of mereology of Section 6.2.2 and of space of Section 6.3.

Axioms (0–16): Define a systems theory. Axiom (11) Systems are individuals. Axiom (12) An individual — which is not a system itself, but is properly contained within the system, is said to be "in the system". Axiom (13) defines a connection (ie. actually its terminal) to be at a systems boundary if the connection connects two individuals of which one is inside (or is) the system and the other is of an individual disjoint from (ie. outside) the system. Axiom (14): Systems can have subsystems. Axiom (15): An open system has at least one connection "at the boundary" of the system. Axiom (16): A closed system has none !

axiom
$\forall$ i,i$'$:I,c:C •
system $\equiv$ sy,
(11) sy(i) $\Rightarrow$ ii(i)
in_system $\equiv$ isy,
(12) isy(i$'$,i) $\equiv$ pp(i$'$,i)$\wedge$sy(i)$\wedge\sim$sy(i$'$)
in_boundary $\equiv$ b,
(13) b(c,i) $\equiv$ co(c)$\wedge$sy(i)$\wedge$
$\exists$ i$''$,i$'''$:I•cs(c,i$''$)$\wedge$cs(c,i$'''$)$\wedge$isy(i$''$,i)$\wedge\sim$isy(i$'''$,i)
subsystem_of $\equiv$ ss,
(14) ss(i,i$'$) $\equiv$ sy(i)$\wedge$sy(i$'$)$\wedge$pp(i,i$'$)
open_system $\equiv$ opns,
(15) opns(i) $\equiv$ sy(i)$\wedge\exists$ c$'$:C•b(c,i)
close_system $\equiv$ clos,
(16) clos(i) $\equiv$ sy(i)$\wedge\sim$opns(i)

6.8.2.2 *Components.* Axioms (0–21): An ontology of components. Axiom (17): Components are individuals. Axiom (18): A sub-component concept. Axioms (19–20): If a connector is connected to a terminal and that terminal is connected to a component, then that connector connects to that component. And vice versa. Axiom (21): Physical systems are systems with possible components.

axiom
$\forall$ i,i$'$:I,t:T,c:C •
component $\equiv$ k,
(17) k(i) $\Rightarrow$ ii(i)
sub_component $\equiv$ sk,
(18) sk(i,i$'$) $\equiv$ k(i)$\wedge$k(i$'$)$\wedge$dp(i,i$'$)
connector_terminal $\equiv$ ct,
component_terminal $\equiv$ kt,
(19) ct(c,t)$\wedge$kt(i,t) $\Rightarrow$ cs(c,i)
(20) cs(c,i)$\wedge$k(i) $\Rightarrow$ $\exists$ t$'$:T•ct(c,t$'$)$\wedge$kt(i,t$'$)
phys_sys $\equiv$ ps,
(21) ps(i) $\equiv$ sy(i)$\wedge$(isy(i$'$,i)$\Rightarrow$k(i$'$))

6.8.2.3 *Mechanical System Processes.* Axioms (0–25) presents a few aspects of a process ontology for physical ("mechanical") systems. Axiom (22): Mechanisms are individuals. Axiom (23): Energy flow is along connections. Axiom (24): Energy flows in one direction. Axiom (25): Mechanical processes are systems and if an individual is inside the system then it is a mechanism.

axiom
$\forall$ i,i$'$:I,c:C •

mechanism ≡ m,
  (22) m(i) ⇒ ii(i)
energy_flow ≡ ef,
  (23) ef(c) ⇒ co(c)
ef_from_to ≡ ft,
  (24) $\mathrm{ft(c,i,i')} \equiv \mathrm{cs(c,i)} \wedge \mathrm{cs(c,i')}$
process ≡ p,
  (25) $\mathrm{p(i)} \equiv \mathrm{sy(i)} \wedge (\mathrm{isy(i,i')} \Rightarrow \mathrm{m(i')})$

Components are the cariers of processes. Core facets of components and processes can be formalised: For process control application by means of *control theoretic* notions[10]. For robotics applications by means of additional *robotics* notions[11]. For optimisation, scheduling and allocation problems, by means of *operations analysis* mathematics. For all of the above physical quantities find their way into *classical mathematical models.* But some components and some processes can also be modelled using the *computing science* techniques of this paper. And two two or more kinds of models mentioned above relate. The former often serve as subsections of the latter: Certain functions (etc.) whose signatures are given in the former kind of models find their proper definition in the computing science models. [12] In this section we shall take a brief look at how to "combine" these models.

Figure 4 Illustrates the above, rather "cursorily".

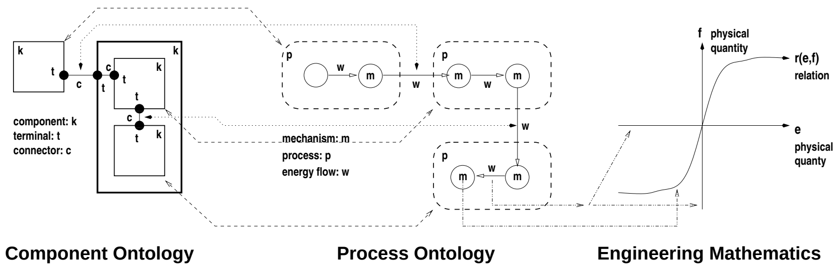

Fig. 4. An Ontology Projection

The component diagram is a syntactic entity — a piece of type and signature text when expressed in RSL. From components we "derive" processes — pieces of type

[10][Dorf 1989; Åström and Wittenmark 1989; Billingsley 1989]

[11][Paul 1981; Lee et al. 1983; Lee 1983b; Lee 1983a; Yap and Schwartz 1987; Trostmann et al. 1986; Boisonnat and Laumond 1989; Laugier 1988; Gorla and Renaud 1988; Gorla and Renaud 1988; Craig 1989; Schilling 1990; McKerrow 1990; Nakamura 1990; Schwartz and Sharir 1990; Baillieul et al. 1990; Brockett 1990b; Brockett 1990a; Borelly and Simon 1990; Yoshikawa 1990; Tsuji 1990; Sandler 1991]. See [Luenberger 1979; Hopcroft and Krafft 1987; Hopcroft 1986; Ljung 1987; Jaffe et al. 1991; Bjørner 1994; Willems 1991]

[12]We may thus, very abstractly, **pre/post** specify for example a scheduling problem in an RSL specification for a railway despatch sub-system. And we then hand this **pre/post** specification document over for further development by an operations analyst. Or we may, [Bjørner 1994], define the combined set of **pre/post** specifications for a robot's inverse kinematics, before we hand over the non-linear (or other) solutions to be found by robotics mathematicians.

and signature texts when expressed in RSL. The component diagram illustrates which processes communicate with which other processes. The process diagram is also a syntactic entity — a piece of type and signature text when expressed in RSL. The engineering mathematics diagram is here shown as for example a control engineer might sketch it. To us that part of the diagram are constituted by the process (and function) definitions.

Axioms (26–27): Projection of simple components onto processes. Axiom (26): Every atomic component must have a process description. Axiom (27): Each mechanism must be part of the process descrption of one component. Axiom (28): A mechanism can only be part of one process description of one component. Axioms (29–30): Energy that flows between processes must go through a connection. Axiom (29): For each connection between components, the processes of these components must interact via an energy flow. Axiom (30): Vice-versa: Energy flow between two component processes goes through a connection.

axiom
$\forall$ i,i$'$,i$''$,i$'''$:I,t,t$'$:T,c:C •
  component_process $\equiv$ cop,
    (26) k(i)$\wedge$si(i) $\Rightarrow$ $\exists$ i$'''$:I•p(i$'''$)$\wedge$cop(i,i$'''$)
    (27) m(i) $\Rightarrow$ $\exists$ i$'''$,i$''''$:I•p(i$''''$)$\wedge$isy(i,i$''''$)$\wedge$cop(i$'''$,i$''''$)
    (28) cop(i,i$'$)$\wedge$cop(i$''$,i$'''$)$\wedge\sim$e(i,i$''$) $\Rightarrow$ d(i$'$,i$'''$)
  connection_energy_flow $\equiv$ cef,
    (29) ct(c,t)$\wedge$ct(c,t$'$)$\wedge$kt(i,t)$\wedge$kt(i$'$,t$'$)$\wedge$cop(i,i$''$)$\wedge$cop(i$'$,i$'''$)
      $\Rightarrow$ $\exists$ c$'$:C•cef(c,c$'$)$\wedge$b(c$'$,i$''$)$\wedge$b(c$'$,i$'''$)
    (30) ef(c)$\wedge$p(i)$\wedge$b(c,i)$\wedge$cop(i$'$,i)$\wedge$p(i$'$)$\wedge$b(c,i$'$)$\wedge$cop(i$''$,i$'$) $\Rightarrow$
      $\exists$ c$'$:C•kt(i$'$,t)$\wedge$ct(c$'$,t)$\wedge$kt(i$'$,t$'$)$\wedge$ct(c$'$,t$'$)

We emphasise that the systems theory of this section (i) is rather tentative and (ii) is entirely "lifted" from [Borst et al. 1996]. We have covered this meta theory in order to make sure that we are sufficiently precise when we say 'system': Many use the term 'system', but with little precision. This is due, of course, to the fact that it is indeed a rather general concept, and that systems take many forms that appear to be different, but are not ! It has become "fashionable" to propose 'system theories'. In our opinion most has been "rubbish". The system meta theory of this section appears to be precise, yet general enough to cover the many different kinds of systems and system implementations that we shall encounter.

6.8.3 *Discussion*

TO BE WRITTEN

6.8.3.1 *Languages and Systems.* Languages are structures of syntactic and semantic types and their values, of denotations, which are functions from semantic values to semantic values, and of semantic functions that map syntactic values into denotations. Systems, seen from a physical and syntactic viewpoint, consists of parts that can be configured into components and these again into systems, where these components and systems may have terminals that connect to connectors that provide component and/or system connections. From a semantic viewpoint components and systems denote processes (sequences of internal and external events) that communicate (energy, of flow of control, or information — as the case may be) across connections, ie. between processes. We have conjectured, in this section, that systems can be viewed as languages. The parts, components, terminals, connectors and systems typically can be considered designations of syntactic terms, usually of nouns, of some professional language, of a field of activity, that which the system "is about". The communication events and the actions derived from these "inside" processes can be considered the denotations of other terms, usually verbs. Properties of systems usually correspond to "adverbial" and "adjectival" terms. We have hedged several claims above: "can be viewed", "can be considered", and "usually". Nothing is exact ! These are only interpretations, models. And there is no one way of interpretation, no unique models. But the reader must be prepared for "uncertainty": In any given abstraction and modelling situation the software engineer must choose among many offerings, sometimes — in fact quite often — contribute new ways of system interpretation: new mappings onto languages.

6.8.3.2 *General*

TO BE WRITTEN

6.9 Discussion

TO BE WRITTEN

7. MODALITIES

7.1 Modal, Epistemic and Deontic Logics — A Survey

7.1.1 *Possible Worlds*

7.1.2 *Modal Operators*

7.2 Logics of Time

7.2.1 *Simple Temporal Logic*

7.2.2 *Interval Temporal Logic*

7.2.3 *The Duration Calculi*

7.3 Modal Logics of Necessity and Possibility

—$\Box p$: It is necessary that p
—$\Diamond p$: It is possible that p

7.4 Epistemic Logics of Belief and Knowledge

—a knows that p
—a believes that p

7.5 Deontic Logics of Intention and Commitments

—$\Box p$: it ought to be that p
—$\Diamond p$: it is permisible that p

7.6 Discussion

8. EVENTS AND CONCURRENCY

TO BE WRITTEN

8.1 Processes: Events and Parallelism

TO BE WRITTEN

8.2 CSP–like Process Modelling

TO BE WRITTEN

8.3 Other Modelling Techniques

TO BE WRITTEN

8.4 Discussion

TO BE WRITTEN

9. CONCLUSION

9.1 What has been Achieved ?

9.2 A Critique

9.3 Future Work

ACKNOWLEDGMENTS

The author thanks *********

REFERENCES

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

ABRIAL, J.-R. 1996b. *The B Book: Assigning Programs to Meanings.* Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England.

ÅSTRÖM, K. AND WITTENMARK, B. 1989. *Adaptive Control.* Addison-Wesley Publishing Company.

BAILLIEUL, J., MARTIN, D., BROCKETT, R., DONALD, B., MURRAY, R., SASTRY, S. S., AND RAGHAVAN, M. 1990. *Robotics*, Volume 41 of *Proceedings of Symposia in Applied Mathematics.* American Mathematical Society, Rhode Island.

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.

BILLINGSLEY, J. 1989. *Controlling with Computers: Control Theory and Practical Digital Systems.* McGraw-Hill.

BJØRNER, D. 1994. *Formal Models of Robots: Geometry & Kinematics*, Chapter 3, pp. 37–58. Prentice-Hall International. Eds.: W.Roscoe and J.Woodcock, *A Classical Mind,* Festschrift for C.A.R. Hoare.

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. 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.

BLIZARD, W. D. 1990. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic 55*, 1 (March), 74–89.

BOISONNAT, J.-D. AND LAUMOND, J.-P. Eds. 1989. *Geometry and Robotics*, Volume 391 of *Lecture Notes in Computer Science* (May 1989). Springer-Verlag, Heidelberg, Germany.

BORELLY, J.-J. AND SIMON, D. 1990. Propositions d'architecture de contrôleur ouvert pour la robotique. Programme 6: Robotique, Image et Vision 1304 (Octobre), INRIA: Sophia Antipolis, Domaine de Voluceau, Rocquencourt, B.P.105, F–78153 Le Chesnay Cedex, France.

BORST, P., AKKERMANS, H., AND TOP, J. L. 1996. Engineering Ontologies. *Submitted to: International Journal of Human-Computer Studies (Special issue on Using Explicit Ontologies in KBS Development)*. The title is ambiguous: What is meant is: "Ontologies of Engineering Systems". Accessed from the Internet: http://ksi.cpsc.ucalgary.ca/IJHCS/VH/-borst-html/ijhcsdoc.html.

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.

BROCKETT, R. 1990a. Formal languages for motion description and map making. In J. BAILLIEUL, D. MARTIN, R. BROCKETT, B. DONALD, R. MURRAY, S. S. SASTRY, AND M. RAGHA-

VAN Eds., *see [Baillieul et al. 1990]*, Chapter 6, pp. 181–193. American Mathematical Society, Rhode Island.

BROCKETT, R. 1990b. Some mathematical aspects of robotics. In J. BAILLIEUL, D. MARTIN, R. BROCKETT, B. DONALD, R. MURRAY, S. S. SASTRY, AND M. RAGHAVAN Eds., *see [Baillieul et al. 1990]*, Chapter 1, pp. 1–20. American Mathematical Society, Rhode Island.

BURSTALL, R. AND GOGUEN, J. 1977. Putting theories together to make specifications. In *Proc. of (IJCAI) Int'l. Joint Conf. on AI*. Boston.

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

COFI LANGUAGE DESIGN TASK GROUP. 1998. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [CoFI].

CRAIG, J. (1986) 1989. *Introduction to Robotics — Mechanics and Control* (2nd. ed.). Electrical and Computer Engineering: Control Engineering. Addison-Wesley Publishing Company.

DAHL, O.-J., DIJKSTRA, E., AND HOARE, C. 1972. *Structured Programming*. Academic Press.

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.

DORF, R. 1967 (fifth ed. 1989). *Modern Control Systems*. Addison-Wesley Publishing Company.

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

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 .

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.

GEORGE, C. W., VAN, H. D., JANOWSKI, T., AND MOORE, R. 2001. *Case Studies using The RAISE Method*. FACTS (Formal Aspects of Computing: Theory and Software) and FME (Formal Methods Europe). Springer–Verlag, London. This book reports on a number of case studies using RAISE (Rigorous Approach to Software Engineering). The case studies were done in the period 1994–2001 at UNU/IIST, the UN University's International Institute for Software Technology, Macau (till 20 Dec., 1997, Chinese Teritory under Portuguese administration, now a Special Administrative Region (SAR) of (the so–called People's Republic of) China).

GOGUEN, J. 1973. A junction between computer science and category theory. Technical Report RC-4526, IBM Ths.J.Watson Research Center, Yorktown Heights, N.Y.

GORLA, B. AND RENAUD, M. 1988. Models of robot manipulators. In *See [Boisonnat and Laumond 1989]* (1988), pp. 306–335.

HOARE, C. 1972. Notes on data structuring. In *[Dahl et al. 1972]* (1972), pp. 83–174.

HOPCROFT, J. 1986. The Impact of Robotics on Computer Science. *Communications of the ACM 29*, 6 (June), 486–498.

HOPCROFT, J. AND KRAFFT, D. 1987. *The Challenge of Robotics for Computer Science*. Lawrence Erlbaum Associates, Hillsdale, N.J., USA.

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

JACKSON, D. 1995a. Structuring Z Specifications with Views. *ACM Transactions on Software Engineering and Methodology 4*, 4 (October), 365–389.

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.

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

JAFFE, M., LEVESON, N., HEIMDAHL, M., AND MELHART, B. 1991. Software requirements analysis for real-time process control systems. *IEEE Trans. Software Engineering 17*, 3 (March), 241–258.

JONES, C. 1980. *Software Development: A Rigorous Approach.* Prentice-Hall.

JONES, C. 1986. *Systematic Software Development Using VDM.* Prentice-Hall.

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

JONES, C. B. 1990b. *Systematic Software Development Using VDM* (Second ed.). Prentice-Hall International, Englewood Cliffs, New Jersey. ISBN 0-13-880733-7.

LANDIN, P. 1964. The mechanical evaluation of expressions. *Computer Journal 6*, 4, 308–320.

LANDIN, P. 1965a. An analysis of assignment in programming languages. Technical report, Univac Sys. Prgr. Res. Grp., N.Y.

LANDIN, P. 1965b. A correspondence between Algol 60 and Church's lambda-notation, (in 2 parts). *Communications of the ACM 8*, 2-3 (Feb.-March), 89–101 and 158–165.

LANDIN, P. 1965c. A generalization of jumps and labels. Technical report, Univac Sys. Prgr. Res. Grp., N.Y.

LANDIN, P. 1965d. Getting rid of labels. Technical report, Univac Sys. Prgr. Res. Grp., N.Y.

LANDIN, P. 1966a. A formal description of Algol 60. In *[Steel 1966]* (1966), pp. 266–294.

LANDIN, P. 1966b. A lambda calculus approach. In L. FOX Ed., *Advances in Programming and Non-Numeric Computations*, pp. 97–141. Pergamon Press.

LANDIN, P. 1966c. The next 700 programming languages. *Communications of the ACM 9*, 3, 157–166.

LAUGIER, C. 1988. Geometric Reasoning in Motion Planning. In *See [Boisonnat and Laumond 1989]* (1988), pp. 377–413.

LEE, C., GONZALEZ, R., AND FU, K. Eds. 1983. *Tutorial on Robotics (see [Lee 1983b; Lee 1983a])* (IEEE Computer Society, P.O.Box 80452, Worldway Postal Center, Los Angeles, California 90080, USA, 1983). IEEE Computer Society Press.

LEE, C. G. 1983a. Robot Arm Dynamics. In C. LEE, R. GONZALEZ, AND K. FU Eds., *see [Lee et al. 1983]*, pp. 93–102. IEEE Computer Society Press.

LEE, C. G. 1983b. Robot Arm Kinematics. In C. LEE, R. GONZALEZ, AND K. FU Eds., *see [Lee et al. 1983]*, pp. 47–65. IEEE Computer Society Press.

LJUNG, L. 1987. *System Identification: Theory for the User.* Information and Systems Sciences. Prentice-Hall.

LUENBERGER, D. 1979. *Introduction to Dynamic Systems Theory: Theory, Models & Applications.* Wiley.

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.

MCKERROW, P. 1990. *Introduction to ROBOTICS.* Addison-Wesley Publishing Company.

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. 1999a. CASL: A guided tour of its design. In *Recent Trends in Algebraic Development Techniques, Proc. 13th International Workshop, WADT '98, Lisbon, 1998, Selected Papers*, Volume 1589 of *LNCS* (1999), pp. 216–240. Springer-Verlag.

MOSSES, P. D. 1999b. 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 Networks. In *ICFEM'97: International Conference on Formal Engineering Methods* (November 1997). IEEE Computer Society: IEEE CS Press.

NAKAMURA, Y. 1990. *Theory of ROBOTICS*. Addison-Wesley Publishing Company.

PARNAS, D. 1972a. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (Dec.), 1053–1058.

PARNAS, D. 1972b. A technique for software module specification with examples. *Communications of the ACM 14*, 5 (May).

PARNAS, D., (EDITOR), D. M. W., AND (EDITOR), D. M. H. 2001. *Software Fundamentals: Colected Papers*. Addison–Wesley Pub. Co. 688 pages. ISBN 0201703696. Amazon price (August 2001) US`$ 49.95.

PAUL, R. 1981. *Robot Manipulators: Mathematics, Programming, and Control*. The MIT Press Series in Artificial Intelligence. The MIT Press, Cambridge, Mas. and London, England.

PLOTKIN, G. 1981. A structured approach to operational semantics. Technical report, Comp. Sci. Dept., Univ. of Edinburgh.

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.

RYDEHEARD, D. AND BURSTALL, R. M. 1991. *Computational Category Theory*. Prentice-Hall Intl.

SANDLER, B.-Z. 1991. *Robotics: Designing the Mechanisms for Automated Machinery*. Prentice-Hall Intl. Editions.

SANNELLA, D. AND TARLECKI, A. 1986. Extended ml: an institution-independent framework for formal program development. In *Category Theory and Computer Programming*, Volume 240 of *Lecture Notes in Computer Science*, pp. 364–389. Springer-Verlag, Heidelberg, Germany.

SCHILLING, R. 1990. *Fundamentals of Robotics, Analysis and Control*. Prentice-Hall International, Inc.

SCHWARTZ, J. 1973. The SETL language and examples of its use. Technical report, Courant Institute of Mathematics, New York University.

SCHWARTZ, J. AND SHARIR, M. 1990. Algorithmic motion planning in robotics. In J. LEEUWEN Ed., *[van Leeuwen 1990], vol. B.*, pp. 391–430. North-Holland Publ. Co., Amsterdam, The Netherlands.

SCOTT, D. 1970. Outline of a Mathematical Theory of Computation. In *Proc. 4th Ann. Princeton Conf. on Inf. Sci. and Sys.* (1970), pp. 169.

SCOTT, D. 1972. Mathematical concepts in programming language semantics. In *Proc. AFIPS, Spring Joint Computer Conference, 40* (1972), pp. 225–234.

SCOTT, D. 1981. Lectures on a mathematical theory of computation. Techn. Monograph 19, Programming Research Group.

SCOTT, D. AND STRACHEY, C. 1971. Towards a mathematical semantics for computer languages. In *Computers and Automata*, Volume 21 of *Microwave Research Inst. Symposia* (1971), pp. 19–46.

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.

STEEL, T. 1966. *Formal Language Description Languages*, IFIP TC-2 Work.Conf., Baden. North-Holland Publ.Co., Amsterdam.

STRACHEY, C. 1973. The varieties of programming languages. Techn. Monograph 10, Programming Research Group.

STRACHEY, C. 1974. Continuations: A mathematical semantics which can deal with full jumps. Techn. monograph, Programming Research Group.

STRACHEY, C. AND SCOTT, D. 1970. Mathematical semantics for two simple languages. Technical report (Aug.), Princeton Univ.

TROSTMANN, E., HOLM, H., PALSTRÖM, B., AND EDEMANN, J. 1986. *Robotteknik* : eng.: Robotics, Volume S 86.18 of *Lecture Notes*. Dept. of Control Techniques, Techn. Univ. of Denmark.

TSUJI, S. 1990. *AI-Based ROBOTICS*. Addison-Wesley Publishing Company.

VAN BENTHEM, J. 1983, 1991. *The Logic of Time* (Second ed.), Volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands.

VAN LEEUWEN, J. 1990. *Handbook of Theoretical Computer Science: vol.A: Algorithms and Complexity, vol.B: Formal Models and Semantics*. North-Holland Publ.Co., Amsterdam.

WILLEMS, J. 1991. Paradigms and puzzles in the theory of dynamical systems. *IEEE Trans. on Automatic Control 36*, 3 (March), 259–294.

WOODCOCK, J. AND LOOMES, M. 1988. *Software Engineering Mathematics*. Pitman, London.

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.

YAP, C. AND SCHWARTZ, J. Eds. 1987. *Algorithmic and Geometric Aspects in Robotics*, Volume 1 of *Advance in Robotics*. Lawrence Erlbaum Associates, Hillsdale, N.J., USA.

YOSHIKAWA, T. 1990. *Foundations of Robotics*. The MIT Press.