Towards a Meaning of 'M' in VDM

Specification Methodology Aspects of the Vienna Development Method

Invited Tutorial

Dines Bjørner
Department of Computer Science
Technical University of Denmark
DK-2800 Lyngby, Denmark

Abstract

A number of steps together characterising a method according to which one may employ the Vienna Development Method, VDM, are formulated and briefly illustrated. The paper is a summary of the methodological aspects of VDM as espoused in the author's recent books: *Software Architectures and Programming Systems Design,* vols. I-III incl. Only specification facets will be illustrated. A subsequent paper will similarly overview the implementation facets of function transformation and data reification.

The exposition is tentative: no claim is made as to the definitiveness of the method. The author believes this to be a "first" attempt to enumerate the methodological facets of VDM. It is felt that few, if any, so-called methods, can claim to be methods according to the definition given in the current paper. The author hopes that the present paper will spur more papers on the "true" nature of 'programming methodology'.

Contents

1 Introduction

1.1 The Vienna Development Method

VDM stands for a relatively well propagated method for developing deterministic systems software, like compilers, database management systems, application program generators, etc., as well as major parts of non-deterministic, concurrent and distributed software, such as operating systems, local area nets, office automation systems, etc.

VDM has developed over the years, from 1973 onwards.

Currently the British Standards Institute (BSI) is readying a draft proposal for a standard for a VDM Specification Languages (SL). It is not quite the one you will see exemplified in this paper, but one that unites and moderately extends a number of slightly differing schools of SL syntaxes, notably the English and the Danish schools. The English school goes considerably beyond the original VDM SL otherwise known as Meta IV, and is exemplified in the latter works of C.B.Jones and the industry groups at STC/STL (Standard Telephone and Cable, ST Labs.), ICL, Praxis and Adelard. The Danish school is exemplified in the works of the current author and the industry groups of Dansk Datamatik Center (DDC)[1] and DDC International[2].

The CEC (Commission of the European Communities) has created a VDM Europe group, some 25-35 VDM practitioners and researchers from industry and academia alike. This group meets thrice yearly to discuss (i) experience in the industrial, professional use of VDM, (ii) tool requirements and experience, (iii) education and training requirements and experience, (iv) mathematical foundations, and (v) possible standardisations of various VDM facets. VDM Europe also organizes VDM Symposia. Two have so far been held: March 1987 (in Bruxelles, Belgium) and September 1988 (in Dublin, Ireland). VDM'90 will be held in Kiel, Federal Republic of Germany, April 1990[3].

The Reference section lists a number of VDM references: [1], [2], [3], [4], [5]. [6] and [7] contain papers on VDM presented at the first 2 VDM Symposia.

1.2 Software Architectures and Programming Systems Design — A series of VDM Books

The current author is presently readying a series of books for publication. The series title is: *Software Architectures and Programming Systems Design — The VDM Approach.* Volume titles are: I: Foundations, II: Abstraction Principles, III: Formal Models of Program and Programming Concepts, IV: Implementation Principles, V: Programming Languages: Interpreters and Compilers, and *VI: Data Models and Data Base Management Systems.* Volumes IV-V-VI may appear as one, rather thick volume. Volumes I-II-III are expected out by the summer of 1989 — and together are expected to take up some 1400 pages ([8], [9], [10]).

The current tutorial reiterates, and summarizes, within the short span of 9 pages, the extent to which the Methodology facets of VDM have been enunciated. See further the discussion below, in subsection 1.4.

[1]DDC is now a part of CRI: Computer Resources International.

[2]DDC International is now a company, independent of Dansk Datamatik Center.

[3]Organisation Committee chairman is Prof. Hans Langmaack, Kiel; Programme Committee chairman and co-chairman are the current author and Prof. C.A.R.Hoare, Oxford. The theme of VDM'90 will be VDM and Z.

1.3 Model-Theoretic and Property-Oriented Specifications

VDM is based on a model-theoretic view of specification and development — and can be said to be based on a denotational semantics viewpoint, as contrasted to for example an algebraic semantics viewpoint. In a model-theoretic specification the formulas explicitly give a mathematical model, that is: denote a mathematical object. An algebraic (or an axiomatic) specification is property-oriented, that is: it "only" prescribes properties that a desired (mathematical object, or) program or software system is to satisfy.

In the DDC/ICL/NBB/STL RAISE project (ESPRIT 315), which spiritually began as a project to establish a firm foundation and methodology for VDM, that VDM has been "replaced" by a specification language (RSL, for RAISE Specification Language) and a development method, which differs rather radically from VDM. RSL embodies specification constructs that allow a "spectrum" of from model-theoretic to property-oriented specifications, of determinate, non-determinate and concurrent (parallel) programs and software.

The border-line between model-theoretic and property-oriented is not sharp, as will also be illustrated in this tutorial exposition.

1.4 What is a 'Method'?

In this subsection we discuss and define the concept of 'method'.

Although the 'M' in VDM stands for 'method', it is not quite clear whether VDM really qualifies to represent a full fledged method. Let us define what we mean by a method:

Definition 1 A method is a set of procedures for selecting and applying a number of principles, techniques and tools in order effectively to construct an effective mechanism (here: software).

The present author would like, here, to emphasize that the facets of the method espoused in the current paper are far from definitive.

The ultimate method would probably provide a calculus of design, much in the spirit of Leibniz — calculus which uniquely determines how to express an abstract model, what to emphasize, etc., and how to transform such an abstraction into a realisation.

The VDM is far from being such a method. At each step of abstraction and at each step of transformation the developer (that is: the user of VDM) is asked to inject abstraction and realisation decisions that could make the specification and the implementation go one way or the other.

A Discussion

So VDM is far from being a method as determinate as for example Michael Jackson's Systems Design (JSD). Both JSD and VDM rests on fairly firm mathematical foundations — although JSD carefully avoids burdening its users with having to be aware of those foundations.

A superficial answer to a question of why VDM is not as strict a method as for example JSD would stress that JSD is aimed at a very well-defined, narrow segment of software development, whereas VDM claims to be far more widely applicable. Thus for a given application area, like for example the devlopment of compilers for Algol/Pascal like languages, VDM can be embellished with strict methodological steps.

2 Overview of Procedures, Principles, Techniques, and "the" Method

In this section we overview the components that together make up one view of what the 'Method' aspect of VDM is.

2.1 Procedures

The procedures have to do with selecting among a variety of possible, or potential, principles, techniques and tools, and, having chosen such, with applying them according to their intentions. Thus selection pre-supposes that alternative principles, etc., at appropriate steps of development (specification and implementation), are available. We shall illustrate such possibilities.

2.2 Principles

We distinguish betwen general, concrete, methodological and philosophical principles.

2.2.1 General Principles

General principles are centered around the model-theoretic approach: that of constructing, using mathematics, couched in some specification and programming language notation, an abstract model (an abstract specification) of the software to be implemented, and that of transforming such a specification to a realisation.

Facets of general principles include: (i) analysis of existing artifacts and theories — from, or based, on which abstractions are constructed, (ii) combination of design and analysis, (iii) decomposition and reduction, (iv) abstraction, (v) limits of scope and scale, (vi) divide and conquer, (vii) the impossibility of capturing everything desired, etc.

We shall, in this introductory overview, of general principles, focus on abstraction.

2.2.2 Concrete Principles

Concrete sub-principles are those of for example: (i) the iterative nature of development, (ii) representational and operational abstraction, (iii) denotational and computational semantics, (iv) applicative and imperative function definitions, (v) hierarchical and configurational development and/or presentation, (vi) identification and discharge of proof obligations, etc.

We shall in this tutorial overview focus on items (ii)-(iv) only.

2.2.3 Methodological Principles

Facets here are: (i) the reduction principles: the whole = the sum of the parts, (ii) discreteness of development: separation of case analysis, induction, abstraction, etc., (iii) embedded nature of software: impact of context, environment and enclosing system (problem domain dependencies), etc.

In the present paper we shall not have space for illustrating these methodological principles in detail.

2.2.4 Philosophical Principles

Facets here are: (i) prevention is better than cure — which, to us, translates into: develop provably correct software, ie.: (ii) proof of properties rather than test for satisfaction, (iii) provision of a method that allows from strict formal, via rigorous, to relaxed systematic usage, all the while allowing for example systematic method uses later to be tightened ("repaired") to rigorous and formal uses, etc.

In the current overview we shall not illustrate items (i-ii), and, of (iii), only illustrate the rigorous and systematic uses of VDM.

2.3 Techniques

The techniques fall into three categories: specification, implementation, and proof techniques.

2.3.1 Specification Techniques

Within the specification techniques we can mention (i) explicit function definitions versus function definitions by a pair of pre-/post-conditions; (ii) definition of the semantics of GOTO programming language constructs by means of either a direct semantics, a continuation semantics, a resumption (style) semantics, or, as was quite common in VDM, by means of a so-called exit mechanism; (iii) the definition of composite (set, tuple or map) objects by means of (set, tuple, respectively map) comprehension — as contrasted to recursive object constructions; etc.

We shall illustrate a number of such techniques — and refer to the literature for illustrations of other techniques.

2.3.2 Implementation Techniques

The three major implementation techniques are: (i) data reification — in which abstract objects are injected into (represented by) more concrete objects; (ii) function transformation — in which, for example, recursive function definitions are transformed into imperative, iterative ones; and (iii) transformations of pre-/post-specified operations into concrete programs.

We shall very briefly illustrate data reification, and otherwise refer to the literature for illustrations of function and operation transformations.

2.3.3 Proof Techniques

Proof techniques apply either to proving properties of a specification (whether abstract, or of an implementation), or to proving properties of a pair of specifications (one being abstract, the other supposedly being a step of realisation, a transformation — of the former).

Subsidiary techniques relating to the latter involve the use of representation (injection) and abstraction (retrieval) relations, respectively functions.

2.4 Tools

The tools are either intellectual or material. The foremost intellectual tool is the specification language (viz.: Meta IV). Common material tools are: syntax directed editors, type checkers, etc.

2.5 "The" Method

The specific methodological instantiation of procedures, principles, techniques and tools (particular to VDM) normally prescribes the ordered establishment of a number of specification components — such as outlined in section 8, see contents listing above for a brief summary.

In creating definitions of semantic domains, and their invariants, syntactic domains, and their well-formedness, and of semantic and auxiliary functions, one then applies the principles, techniques and tools.

— Interlude

Our presentation of techniques, to follow, in the next sections, 4-7, is presented configurationally, bottom-up. Section 2, above, presented the 'problem' top-down, hierarchically.

3 Operational Abstraction

In order to usefully express manipulations of "real world" or "imagined world" objects we must choose representations of (primitive) operations and (defined) functions.

Definition 2 *By an* **operational abstraction** *we understand a [possibly formal] specification of operations and functions [on objects] — again — (1) irrespective of any possible external world manifestation; (2) irrespective of any computer realization of such functions; and (3) such that the function specification concentrates ('as far as possible') on properties, i.e. in* what *the function computes — rather than* how *it computes its result.*

That is: we either specify functions *in extension,* by defining the input/output relation; or we give an abstract *in intension* recipe, of algorithmic nature, of how the function might achieve its results.

Which are then the 'real', or 'imaginary', world 'things' to which we apply the principles of operational abstraction?

Again we answer by stating examples: operational abstraction is applied when specifying manipulations on objects like directories, inventories, etc. Such manipulation may e.g. be: inserting new entities into- or changing or deleting entities of, or from, directories and inventories, and looking up information contained in such inventories, etc.

Examples of operational abstractions are those of modelling 'the look-up in a directory of a phone number' by 'the application of a map to an argument of its domain', similarly for inventory search, and 'the listing of division department names' by 'taking the defining domain of the division map obtained by applying the organisation map to a division name argument'.
It is here being stressed that we have a two-by-two situation: (i) we (may) wish to abstract from how manipulations of 'real world' objects take place in 'reality', and (ii) from how we implement them inside the computer:, and (i') we wish sometimes to define functions purely in extension, by what they compute, as opposed to (ii') defining functions in intension, by how the achieve what they compute!

Also in the latter cases (i'-i") — in extension versus in intension — we still wish to be abstract!

3.1 Function Definitions

Previous sections and sub-sections gave examples of direct, or putative, function definitions. Other sections and sub-sections have presented function definitions in the style of axiomatic specifications and by means of pre-/post (predicate) specifications. Let us briefly and informally review these distinctions.

3.1.1 Direct Function Definitions

Assume *sq* to denote a square root function defined explicitly (say algorithmically, using the Newton-Raphsons, *NR*, method), that is:

1. $\underline{\textbf{type}} : sq\colon \textbf{RAT} \xrightarrow{\sim} \textbf{RAT}$
.1 $\underline{\textbf{pre}}\text{-}sq(r) \triangleq r \geq 0$
.2 $sq(r) \triangleq NR_{\mathrm{Expr}}(r)$

The form NR_{Expr}(r) is intended to be an expression (with *r* free) which somehow represents the Newton-Raphson method for taking squeare roots. The problem with the expression *NR* is that it most likely is a rather convolute (ie. "tricky" to understand) expression. Reading it might not reveal, very easily, that it indeed does perform the desired function. Thus using *NR* may not be a good abstract specification, but perhaps a good effecient, concrete coding of the problem.

As we shall see later in this book, one can indeed define quite a lot of functions explicitly, and abstractly.

Suffice the above for the time being as an illustration of the notion of direct (explicit, putative, constructive) function definitions.

3.1.2 Pre-/Post- Definition of Functions

In contrast, a pre-/post definition of *sq* could like like:

2. $\underline{\textbf{pre}}\text{-}sq(r) \triangleq r \geq 0$
.1 $\underline{\textbf{post}}\text{-}sq(r,r') \triangleq (r' \times r') = r$

- expressing very explicitly that *sq* is (to be) a square root function. That is: the above *specifies* a function — which then, later, has to be *implemented*. As a technicality, however [that is, as something that has less or nothing to do with the problem of specifying versus implementing, but rather with the specific problems of the square root function, ie. the speific problem at hand], we must remark that the above is probably too stringent a specification — one that can never be fully satisfactorily implemented. The technicality is that of approximating the result r'. Normally we would be content with an answer, r', satisfying:

3. $\underline{\textbf{post}}\text{-}sq(r,r') = \mid r - (r' \times r') \mid \leq \epsilon$

where epsilon: ϵ is some given (global) small value, or ϵ is given explicitly:

4. $\underline{\textbf{type}}: \; sq\colon \mathrm{RAT} \times \mathrm{RAT} \rightarrow \mathrm{RAT}$

.1 $\underline{\textbf{pre-}}sq(r,\epsilon) \triangleq (r \geq 0) \wedge (0 \leq \epsilon \ll 1)$

.2 $\underline{\textbf{post-}}sq((r,\epsilon),r') \triangleq ((1-\epsilon) \times r \leq r' \times r' \leq (1+\epsilon) \times r)$

where $\ll$ denotes the "much, much smaller than" relational operator.

In this chapter we shall only introduce the three function definition styles. We shall not systematically cover aspects of how, when and where they are used (selected for use). This will not be done extensively till we reach volume II. So we now go on to the axiomatic style of defining functions.

3.1.3 Axiomatic Function Definitions

Here we illustrate some axiomatic definitions.

The problem we wish to tackle has to do with finding substrings of strings of characters. We will not give a model of character strings other than saying (i) that $\underline{\textbf{null}}$ represents the empty (void) string, (ii) that if s and r are strings (ie.: $s, r \in String$), then $s\frown r$ stands for the string whose first part is the string s and whose remaining part is the string r.

We wish to define two functions: B (for 'before'), and A (for 'after'):

5. $\underline{\textbf{type}}: \; B: String \times String \rightarrow String$

.1 $\underline{\textbf{type}}: \; A: String \times String \rightarrow String$

such that $B(r,s)$ yields the substring (or part) of s before the first left-to-right occurrence of r, and $A(r,s)$ yields the substring (or part) of s after the first left-to-right occurrence of r.

To define B and A we introduce an auxiliary function I (for initial substring):

6. $\underline{\textbf{type}}: \; I\colon String \times String \rightarrow \mathrm{BOOL}$

.1 $\underline{\textbf{axiom}}: \; (\forall r, s \in String)(I(r,s) \equiv (\exists t \in String)(r\frown t = s))$

that is: $I(r,s)$ is true if r is an initial substring of s. The axioms for B and A are now:

7. $\underline{\textbf{axiom}}: \; (\forall r, s \in String) \; (B(r,s)\frown r\frown A(r,s) = s)$

.1 $\underline{\textbf{axiom}}: \; (\forall r, s, t', t'' \in String) \; ((t'\frown r\frown t'' = s) \supset I(B(r,s),t'))$

The first axiom says that any string s containing at least one substring r, consists of a part before r, r itself, and a part after r. The second axiom says that B and A work with respect to the first occurrence of their first argument. If there is another way of decomposing s ($s = t'\frown r\frown t''$) then $B(r,s)$ must be an initial part of t'.

The specification of B and A is such that $B(r,s)$ and $A(r,s)$ are not specified to produce results if r is not a part of s.

Note that the two axioms "simultaneously", ie. in an "intertwined" way, define B and A. There is not (here) a single axiom - or a set of axioms - which only define all of B, respectively all of A. Even though the second axiom contains no explicit use of A, it constrains the implementation of A because of the way in which B and A are related by the first axiom.

A <u>pre-/post</u> definition of B, which specifies that r must be a substring of s, could look like:

8. $\underline{\textbf{pre-}}B(r,s) \triangleq (\exists t_1, t_2 \in String)(s = t_1\frown r\frown t_2)$

.1 $\underline{\textbf{post-}}B((r,s),t) \triangleq$

.2 $\quad (\exists t' \in String)$

.3 $\quad ((s = t\frown r\frown t')$

.4 $\quad \wedge(\neg \exists t_1, t_2 \in String)$

.5 $\quad ((t_1 \neq t) \wedge (t\frown r = t_1\frown r\frown t_2)))$

A similar <u>pre-/post</u> definition of A is left as an exercise.

3.1.4 Loose [Function] Definitions

The pre-condition (pre-B) on application of B (ie.$B(r,s)$) is more constraining than the previous axioms on B. And the post-condition on the square-root function, sq, permits a range (an infinity) of implementations.

It is thus we see that descriptive, ie. pre-/post and axiomatic (axiom:), definitions are *vague*.

But this vaguesness, in the cases shown, was on purpose. Any implementation which satisfies such purposefully vague definitions is acceptable. It is, however, not only descriptive function (and, in general, object) definitions that can be made purposefully vague. Also prescriptive (explicit, putative, or direct) definitions can be loosened up to allow for a range of implementations. One way of doing so is through operational and representational abstraction – to be treated systematically in volume II. Another way is through the use of non-deterministic "function" definitions – ie. in reality: relation definitions. A specification construct is non-deterministic (or non-determinate) if its use in some expression form may denote either one from a set of values.

Definitions, like the above, which leave certain aspects unspecified are neither imprecise nor ambiguous. They specify those aspects in which we are interested and have left open those things in which we are not, at first, interested. Thus such definitions leave choices open for later design stages.

We call such (purposefully vague) definitions for **loose specifications**. Space does not here permit us to single out two sub-classes of loose specifications: non-determinate and under-specified.

3.1.5 Concluding Remarks

We have given three seemingly distinct ways of defining functions. In the previous sub-section we have indicated, and we shall throughout the book illustrate, that these three definition styles are but points on a "continuous" spectrum of definition styles. That is: a function basically defined (or planned expressed) in the prescriptive style may contain relational or non-deterministic constructs such that the function definition, as a whole, is non-determinate by containing descriptive parts. Etcetera.

One could argue that descriptive definition styles are more abstract, and are less bound to implementational bias than is the presecriptive style. In fact one could successfully argue that the descriptive style "**specify**", whereas the prescriptive style **implements**. That is: one could think of mechanizing the interpretation of prescriptively defined functions, but one must abandon such thoughts of evaluating descriptively defined functions. Mechanizing the latter would entail, in its general form, a full theorem proving capability of the interpreter – and this would in general not be possible. Since even the basically prescriptively defined functions may contain, or rely on, descriptive parts, respectively defined auxiliary functions, we also abandon that thought even for direct function definitions.

So, why do we also provide a direct (prescriptive) function definition since, as we have claimed, the descriptive styles are more specification-oriented? The answer basically is that it turns out, when specifying, that there are a large majority of definition situations where it either becomes very difficult to ascertain whether an axiomatic definition is consistent and complete, or it becomes exceedingly tedious, clumsy and voluminous to define functions descriptively, or both. In a sense it therefor becomes well nigh "impossible", and certainly we loose transparency and thus comprehensibility. In other words: the prescriptive definition style in reality is often more specification-friendly!

One could now turn the question around: since, as claimed, we shall be using the prescriptive, ie. the model-oriented approach more often, why do we still allow descriptive definition components? The answer here reflects a pragmatic attitude, and goes as follows: we do so because there are specification situations, especially in defining auxiliary functions, and increasingly a larger need for such auxiliary functions arise in design stages, where the definition of such functions is best, ie. most succinctly, done in the descriptive style.

The above concluding remarks may seem abstract – as they are presented here, early on, without much evidence, ie. with only very few examples. We beg the reader's patience – our points will be amply illustrated in these volumes, and we shall then have ample opportunity to refer back to the discussion of the present sub-section.

3.2 Proof Obligations

Prescriptive function (and other object) definitions are *model-oriented*. That is: the functions and hence all the objects it manipulates do exist, at least in some constructive, mathematical universe. This is not necessarily the case for the descriptive definition style, which, in analogy, we also term: *property-oriented*. In the latter the objects (incl. functions) are postulated. Since they are basically described through the use of predicates (axioms, pre-/post conditions), one is faced with the burden of showing that there is at least one interesting, ie. non-trivial model for those axioms (etc.). We need to show so since we aim at implementing the specified thing within the computer.

We say that specifying objects (incl. functions) axiomatically (ie. in general, descriptively, through their properties) **generates a proof obligation**: we are obliged to show existence of an object (of a function) which satisfies the description. Since we are concerned with computation we have, in addition, to show that the object (the function) is computable!

So for every pair of **pre-/post** conditions, purportedly defining a function, an **implementability** proof obligation arises. And similar for a set of axioms together characterising a set of functions; also here implementability proof obligations arise.

The form of the **implementability theorem** for **pre-/post** specified functions is as follows. In general we have:

9. $\underline{\textbf{type}} : f : D \rightarrow R$
.1 $\underline{\textbf{pre-}}f : D \rightarrow \texttt{BOOL}$
.2 $\underline{\textbf{post-}}f : D \times R \rightarrow \texttt{BOOL}$

The $\underline{\textbf{pre-}}$condition $\underline{\textbf{pre-}}f(d)$ says:

10. $\underline{\textbf{pre-}}f(d) \supset (\exists r \in R)(f(d) = r)$

The $\underline{\textbf{post-}}$condition $\underline{\textbf{post-}}f(d,r)$ says:

11. $\underline{\textbf{post-}}f(d,r) \supset \underline{\textbf{pre-}}f(d)) \wedge (\exists f \in (D \rightarrow R))(f(d) = r)$

Hence the implementability condition is:

12. $(\forall d \in D)((\underline{\textbf{pre-}}f(d) \supset (\exists r \in R)(\underline{\textbf{post-}}f(d,r))))$

4 Representational Abstraction

Definition 3 *By a* **representational abstraction** *we understand a [possibly formal] specification of domains and instances of objects (1) irrespective of a number of 'real world' properties of the modelled phenomena, and (2) irrespective of, or, more apprpriately: with no bias towards, any possible realization of such objects.*

To paraphrase the latter: we do not take into consideration, when specifying objects and their Domains abstractly, how we may wish, or be able, to implement these objects. Furthermore the representational abstraction attempts to model 'as closely as possible' only relevant and intrinsic properties.

Which are the 'real', or 'imaginary', "worldly" 'things' to which we then apply the principles of representational abstraction? In an attempt to stay clear of philosophical issues (of e.g. epistemological nature) we give an operational answer to the above question.

Examples of 'things' subjected to representational abstractions are: (the concept of) telephone directories, (...) inventory lists, and (...) company organizations (viz.: organisation charts).

Examples of representational abstractions are: directories are seen as maps from names (and addresses) of telephone subscribers to telephone numbers; inventory lists are seen as maps from part numbers to quantity on hand and where stored; and company organisations are seen as maps from division

names to divisions, where divisions are seen as maps from department names to departments, where departments are seen as maps from ... etc.
It is here being stressed that we deal with two abstraction concerns: (1) abstracting away what is considered irrelevant properties of 'real world' object phenomena, and (2) abstracting from *how* to implement these inside the computer.

We now illustrate the notion of representational abstraction while at the same time illustrating four basic composite data type abstractional facilities of VDM.

4.1 Set Abstractions

Sets, as an abstract data type, is an auxiliary "work horse", used in many contexts, as will be seen below, but not itself a prime vehicle. That is: few every day notions, such which we are to model abstractly, directly "asks" for one in terms of sets. But let us try anyway.

4.1.1 A Demographic Database — an Example

In this example the following 8 points (i-viii) characterise the problem:

(i) a State consists of a set of Counties;

(ii) a County consists of a set of distinct communities (hamlets, villages, towns, and cities – all considered on par);

(iii) each Community consists of a distinct set of Households;

(iv) and each Household consists of a set of Persons.

(v) Each Person is uniquely identified - by some county-wide unique numbering system.

(vi) No two counties have identical households.

(vi) No two otherwise distinct Households of a State have Persons in common;

(viii) and a Household (a core-family) may live in several counties, but not in several communities within the same county (don't ask why!).

Any of the descriptions (i-vi) give rise to respective Domain definitions; descriptions (vii-viii) give rise to invariance definitions:

$S = C$-**set**	(1)	State
$C = V$-**set**	(2)	County
$V = H$-**set**	(3)	Community (...,Village,...)
$H = P$-**set**	(4)	Household
$P =$ TOKEN	(5)	Person

Characterisation (vi) justify Domain definitions (1-5) since it implies no two Communities with two or more identical households, etc.

Now to the invariance:

13. $\underline{\mathbf{inv}}\text{-}S(s) \triangleq No2HiSC(s) \wedge NotSH2iC(s)$

The first predicate *No2HiSC* (short for: no two otherwise distinct households in the same state have persons in common) models (vii), and *NotSHi2C* (for: not same household twice in same county) models (viii):

14. $No2HiSC(s) \triangleq (\forall h,h' \in HiS(s))\ ((h \neq h') \supset ((h \cap h') = \{\}))$

14.1 $\underline{\mathbf{type}}$: $No2HiSC\colon S \rightarrow$ BOOL

where:

15. $HiS(s) \triangleq$ **union** **union** s
15.1 **type** : $HiS\colon S \rightarrow H$-**set**

computes all households of a state.

16. $NotSH2iC(s) \triangleq (\forall c{\in}s)(\forall v,v'{\in}c)\ ((v{\neq}v'){\supset}((v{\cap}v') = \{\}))$
16.1 **type** : $NotSH2iC\colon S \rightarrow$ BOOL

If there is more than one community in a county then they must not have households in common:

17. $NotSH2iC(s) \triangleq (\forall c{\in}s)\ (($**card**$c{\geq}2){\supset}($**intersect**$c = \{\}))$

is another way of expressing (viii). That is: functions (16.) and (17.) are identical in extension (but not in intension).

4.1.2 The Set Modelling Principle

We are ready to summarize the essence of the above example.

The question to be answered is this: when should you use the set data type abstraction in your abstractions? An informal answer is for example:

- When the object being subject to abstract modelling posseses a composite property such that it can be regarded as a finite collection of unordered, un-distinguished, but distinct elements, then a set abstraction seems reasonable!
- If, furthermore, manipulations of the object may involve arbitrary selection of component (sub-)objects, removal or addition of distinct objects, etc., then a set abstraction seems further motivated.

The above modelling principle is just a rule-of-thumb. It is vaguely formulated. It cannot be more precisely stated! Once you have digested the contents and the similar modelling principles of the next three subsection you will better appreciate why the principles must necessarily be approximate.

The above rules, in actual modelling situations "translates" as follows, in two ways: (i) if you are 'told': *Some facility consists of an unordered collection of distinct, further un-distinguished things* etc., then you should consider whether a model based on a set abstraction is otherwise appropriate; and (ii) vice-versa: in deciphering somebody else's unstructured, informal, ad-hoc, incomplete and possibly even inconsistent "specification", you should analyze that description with spectacles viewing "it" (the thing spoken about by the "specification") from the point of view of: is a set abstraction an appropriate choice? (You may find, in the latter case (ii) that it either fits, or does not; if not, then perhaps any of the other composite data types [tuples, maps, trees] may be used.)

4.2 Tuple Abstraction

4.2.1 KeyWord-In-Context, KWIC, Program — an Example

This example sub-section has several sub-parts, and otherwise presents the problem in a more pedantic style than were the examples above. First we are given a problem formulation. We then, very briefly, analyze this given formulation. From the informal formulation and, as a result of the analysis, we (informally, yet somehow) systematically 'derive' our formal model. Finally we discuss our particular model and variants thereof. The purpose of this example illustration is then to show some of the aspect of going from fixed, by others given problem formulations to models, and the problems posed by such oftentimes incomplete (or, but not in this case, inconsistent) informal formulations.

4.2.2 The Given Problem

We are given the following informal, english language program specification:

"Consider a Program which generates a KWIC Index (KeyWord-In-Context). A *title* is a list of words which are either *significant* or *non-significant*. A *rotation of* a list is a cyclic shift of words in the list, and a *significant rotation* is a rotation in which the first word is significant. Given a set of titles and a set of non-significant words, the program should produce an alphabetically sorted list of the significant rotations of titles"

An example of input and output is then given:

"Titles :

THE THREE LITTLE PIGS.
SNOW WHITE AND THE SEVEN DWARWES.

Non-significant Words :

THE, THREE, AND, SEVEN

Output :

DWARFS. SNOW WHITE AND THE SEVEN
LITTLE PIGS. THE THREE
PIGS. THE THREE LITTLE
SNOW WITHE AND THE SEVEN DWARFS.
WHITE AND THE SEVEN DWARFS. SNOW"

4.2.3 Discussion of Informal Problem Formulation

We now analyze the problem statement. The point of our analysis is to isolate concepts, discover incompleteness and/or inconsistencies, etc.

(1) The informal problem formulator already isolated some concepts; these appear italizised in the text. Other concepts potentially useful in, or for, our further work are: *List, word, cyclic shift, first, set,* and *alphabetically sorted.*

(2) Some concepts are problem-oriented: *Title, words, significant,* and *non-significant.* Other concepts are more abstract, explication-oriented: *list, rotation,* (equal to) *cyclic shift, first, set,* and [*alphabetically*] *sorted.* (Our modelling will basically center around, or express, but not necessarily all of, these concepts.)

(3) The descriptive paragraph does not deal with *punctuation marks;* period (".") is not isolated as a concept, but it occurs, as a marker, in the rotations. Also: *words* are not further explained. We take these to consist of letters. And we assume some given *alphabetical order* of, or among, both upper- and lowercase letters. *Blanks* appear, but noting is said about their relation to the ordering of titles.

(4) Nothing is said about duplicate occurrences in the input or output. The input title "XXX XXX" might thus give rise to e.g. two output rotations!

(5) Finally nothing is said about the concrete input and output presentation: carriage returns, new lines; respectively single or multiple column printing and display and the ordering within multiple columns: whether by row or by column. Etc.

4.2.4 Assumptions and Decisions

4.2.5 — Program Assumptions:

In order to proceed into a modelling phase we make the following assumptions:

(1) We ignore punctuation marks.

(2) We assume 'alphabetic sorting' to apply to all of the text of a title.

(3) We omit multiple (duplicate) occurrences of [rotated] titles in the output, i.e. we list (generate) only one copy.

4.2.6 — Model Decisions:

Our modelling will be based on the following decisions:

(4) We assume an ordering relation:

18. $\underline{\textbf{type}}$: *WOrder: Word* $\times$ *Word* $\rightarrow$ BOOL

(5) That is, we assume a Domain of words:

19. *Word*

- not further specified.

(6) We do not abstract away blanks — since blanks (in general punctuation marks) are needed to delineate words.

(7) We abstract, as suggested by the informal formulation, both the presentation of input and output. (This issue will be a pressing one the 'closer' we get to a realisetion — and should, we seriously believe, be specified, in detail, before implementation is properly begun.)

Since we ((1)) ignore punctuation marks, including end-of-title marker, such marks will not be modelled either.

The major model decision is that of giving a model, in particular one in the style that this book advances.

4.2.7 Model

The presentation of the model will follow, in sequence, the way in which it was derived. That is: we decide, in a first, successfull, attempt to model first some of the individual concepts outlined or italisized above. Then we bring all aspects together in the specification of the input/output Domains and the one, major program function (i.e. the specification of the program itself). Finally we specify the auxilliary functions introduced by the major program specification.

In this example the modelling of the auxiliary concepts turned out to be of direct use in the subsequent [main] model.

4.2.8 — Auxiliary Notions

"A *title* is a list of *words*" leads to the following (main) Domain:

20. $Title = Word^{+}$

"A *rotation* of a *list* is a *cyclic shift* of the words in the list":

21. $\underline{\textbf{type}}$: $Rotations: Title \rightarrow Title\underline{\textbf{-set}}$
21.1 $Rotations(t) \triangleq \{rot(t,i) \mid i \in \underline{\textbf{ind}}\, t\}$

22. $\underline{\textbf{type}}$: $rot: Title \times \mathbf{N}_1 \rightarrow Title$
.1 $rot(t,i) \triangleq \langle t[j] \mid i \leq j \leq \underline{\textbf{len}}\, t \rangle \frown \langle t[k] \mid 1 \leq k < i \rangle$
22.2 $\underline{\textbf{pre-}}rot(t,i) \triangleq (i \in \underline{\textbf{ind}}\, t)$

"first word":

23. $\underline{\textbf{type}}$: $First: Title \rightarrow Word$
23.1 $First(t) \triangleq \underline{\textbf{hd}}\, t$

"is significant" (w.r.t. a set of non-significant words):

24. $\underline{\textbf{type}}$: *Is-Significant: Title* $\times$ *Word*$\underline{\textbf{-set}}\rightarrow$ BOOL
24.1 $\textit{Is-Significant}(t,ws) \triangleq \neg(\textit{First}(t) \in ws)$

We choose to model "alphabetical sort", rather than "is alphabetically sorted" — leaving the latter as a variant exercise:

25. $\underline{\textbf{type}}$: *A-Sort: Title*$\underline{\textbf{-set}}\rightarrow$ $\textit{Title}^{+}$
.1 $\underline{\textbf{pre-}}\textit{A-Sort}(ts) \triangleq \underline{\textbf{true}}$
.2 $\underline{\textbf{post-}}\textit{A-Sort}(ts,tl) \triangleq$
.3 $\quad(\underline{\textbf{elems}}\,tl = ts)$
.4 $\quad\wedge(\underline{\textbf{len}}\,tl = \underline{\textbf{card}}\ \underline{\textbf{elems}}\ tl)$
25.5 $\quad\wedge\textit{Ordered}(tl)$

Line 25.3 secures that all (rotated) titles in the set, and only such, appear in the title output list; and line 25.4 secures that there are no duplicates.

26. $\underline{\textbf{type}}$: *Ordered:* $\textit{Title}^{+}\rightarrow$ BOOL
.1 $\textit{Ordered}(tl) \triangleq$
26.2 $\quad(\forall i,j \in \underline{\textbf{ind}}\,tl)(i<j \supset \textit{T-Order}(tl[i],tl[j]))$

27. $\underline{\textbf{type}}$: *T-Order: Title* $\times$ *Title* $\rightarrow$ BOOL

Let there be given two titles t_1 and t_2. Assume $t_1 \neq t_2$. For $\textit{T-Order}(t_1,t_2)$ to hold either (i) $\textit{W-Order}(\underline{\textbf{hd}}\,t_1,\underline{\textbf{hd}}\,t_2)$ or (ii) there is a proper prefix, t, of both t_1 and t_2 such that $t_1 = t \frown t'_1$, and $t_2 = t \frown t_2$ such that either $t'_1 = <>$ and $t'_2 \neq <>$, or both $t'_1 \neq <> \neq t'_2$ and $\textit{W-Order}\ (\underline{\textbf{hd}}\,t'_1,\underline{\textbf{hd}}\,t'_2)$:

28. $\underline{\textbf{pre-}}\textit{T-Order}(t_1,t_2) \triangleq (<> \neq t_1 \neq t_2 \neq <>)$
.1 $\textit{T-Order}(t_1,t_2) \triangleq$
.2 $\quad(\textit{W-Order}(\underline{\textbf{hd}}\,t_1,\underline{\textbf{hd}}\,t_2)$
.3 $\quad\vee(\exists t,t'_1,t'_2 \in \textit{Title})$
.4 $\qquad((t_1 = t \frown t'_1) \wedge (t_2 = t \frown t'_2)$
.5 $\qquad\wedge(((t'_1=<>)\wedge(t'_2\neq<>))$
.6 $\qquad\vee((t'_1\neq<>\neq t'_2) \wedge \textit{W-Order}(\underline{\textbf{hd}}\,t'_1,\underline{\textbf{hd}}\,t'_2)))))$

4.2.9 Domains

"Given a set of titles and a set of non-significant words":

29. *Input* $=$ *Title*$\underline{\textbf{-set}}$ $\times$ *Word*$\underline{\textbf{-set}}$

"the program should produce a ... list ... of titles":

30. *Output* $= \textit{Title}^{+}$

4.2.10 The Main Function

The main function is expressed as: "Produce an alphabetically sorted list of the significant rotations of titles":

31. $\underline{\textbf{type}}$: *KWIC: Input* $\rightarrow$ *Output*

Again we choose to express the definition of *KWIC* in terms of a pair of **pre-/post** conditions:

```
32.    pre-KWIC(in) ≜ true
  .1   post-KWIC(in,out) ≜
  .2       Signif-Rots(in,out)
  .3       ∧ Ordered(out)
32.4       ∧ No-Duplicates(out)
```

4.2.11 — Auxiliary Functions

```
33.    type : Signif-Rots: Input × Output → BOOL
  .1   Signif-Rots(in,out) ≜
  .2       All-Rots(in,out)
33.3       ∧Only-Rots(in,out)

34.    type : All-Rots: Input × Output → BOOL
34.1   type : Only-Rots: Input × Output → BOOL
```

The *All-Rots* predicate checks that the output contains all significant rotations inplied by input. The *Only-Rots* predicate checks that the output does not contain other such rotations:

```
35.    All-Rots((ts,ns),tl) ≜
  .1       (∀t ∈ ts)
  .2           (∀t'∈ Rotations(t)
35.3               (Significant(t',ns) ⊃ (t'∈ elems tl))

36.    Only-Rots((ts,ns),tl) ≜
  .1       (∀t'∈ elems tl)
36.2           (∃!t ∈ ts)(t'∈ Rotations(t)) ∧IsSignificant(t',ns))

37.    type : No-Duplicates: Title⁺ → BOOL
  .1   No-Duplicates(tl) ≜
  .2       EITHER: card elemstl = lentl
37.3       OR:      (∀i,j∈indtl)(i≠j ⊃ tl[i]≠tl[j])
```

Observe that although we defined it, we never actually found a need for deploying the *A-Sort* function. Such "things" happen when modelling bottom-up, configurationally!

4.2.12 The Tuple Modelling Principle

The question to be answered is this: when should we apply the tuple data type in our abstractions? The answer goes somewhat like this:

> When the object being subject to abstraction possesses a composite property such that its components can best be thought of as being ordered (rather than un-ordered) and such that it is natural to speak of a first, a second, etc., element, then a tuple abstraction seems reasonable.
>
> If, furthermore, manipulations of the object may involve composing pairs (or sequences) of such objects, as in infix (or distributed) concatenation, or involve inquiring about its length, or about the set of its elements, etc., then a tuple abstraction seems further motivated.

The above modelling principle is a guide-rule. There is nothing absolute about it. It is really not a law cast in concrete. To model abstractly is an art. The discussion at the end of subsection 4.1.2 apply equally well here.

The above rules, in actual modelling situations "translates" as follows, in two ways: (i) if you are 'told': *Some facility consists of an ordered collection of not necessarily distinct, further un-distinguished things* etc., then you should consider whether a model based on a tuple abstraction is otherwise appropriate; and (ii) vice-versa: in deciphering somebody else's unstructured, informal, ad-hoc, incomplete and possibly even inconsistent "specification", you should analyze that description with spectacles viewing "it" (the thing spoken about by the "specification") from the point of view of: is a tuple abstraction an appropriate choice? (You may find, in the latter case (ii) that it either fits, or does not; if not, then perhaps any of the other composite data types [sets, maps, trees] may be used.)

4.3 Map Abstractions

4.3.1 Direct/Random Access Files — an Example

In this section we illustrate abstractions of rather conventional file systems: their objects (files, records, etc.) and operations (read, write, etc.).

4.3.2 Semantic Domains and Semantic Objects

— File systems

The files of our system are uniquely identified, that is two or more otherwise identical files must be distinctly named. Let *FILE* and *Fnm* denote the Domains of further un-explained files, respectively file names. Then:

38. $fs\colon FS = Fnm \underset{m}{\to} FILE$

is a Domain equation. The identifier *FS* (by the use of the equality sign, =) denotes the same thing as does the right-hand-side Domain expression, namely a Domain of maps from file names to files. Thus: any file system, an object (let us call it *fs*) in *FS*, consists of an otherwise unordered collection of uniquely named files.

Let suitably decorated *f*'s (be identifiers which) denote distinct file names, and let suitably decorated *file*'s denote (not necessarily distinct) files, then the expression:

39. $[\, f_1 \mapsto file_1, f_2 \mapsto file_2, \ldots, f_n \mapsto file_n \,]$

denotes a file system. [] denote the empty file system.

— Files, Keys and Records

We choose to illustrate so-called random access files, ie. files whose components (which we could call "records") can be retrieved ("read") on the basis only of a so-called "key". Thus there is a notion of files consisting of records, and of these records being (uniquely) retrievable on the basis only of a key, which is often considered part of the record, and which is otherwise unique to each record of a file. We choose, here, to call that part of a record which is not the key (ie. the record exclusive of its key) for the data part of the record. A record hence consists of two parts: a key and a data part. To sum up: a file is an unordered collection of records. Since these are uniquely identified by their key part, we take a file to be a collection of uniquely keyed data parts. Let the identifiers *Key* and *Data* denote the Domains of respectively keys and data parts, then, on one hand:

40. $file\colon FILE = Key \underset{m}{\to} Data$

defines files to be maps from keys to data. On the other hand, and maybe not so useful here:

41. $r\colon Record = (Key \times Data)$

defines a record to be a pair consisting of a key and a data part. Let suitably decorated k's and d's (be identifiers which) denote keys, respectively data — the former assumed distinct, the latter not. Then:

$$[k_1 \mapsto d_1, k_2 \mapsto d_2, \ldots, k_m \mapsto d_m]$$

denotes a file, with eg.:

42. r: (k,d)

denoting a record (r). We shall presently leave the *Data* Domain further unspecified.

— Primitive File and File System Operations

A number of operations will now be defined on files and file systems. First "informally" formalized, subsequently "closed-form" (function definition) formalized. Let potentially decorated *file*'s, k's, d's, *fnm*'s and *fs*'s be identifiers which which denote files, keys, data, file names and file systems, ie. let:

43. $file \in FILE,\ k \in Key,\ d \in Data,\ fnm \in Fnm,\ fs \in FS$

Then if k is not the key of any record in *file*, ie. if $k \notin \underline{\mathbf{dom}}\ file$, then:

44. $file \cup [k \mapsto d\ ]$

denotes a file which is like *file* is except that it now also contains the record (k,d), that is: we can interpret (ie. understand, or take) the above expression as describing the essential aspect of *writing* a record to a file.

If, instead, k is already a key of some record in *file* (namely record: $(k,\ file(k)))$, then:

45. $file + [\ k \mapsto d\ ]$

could be used for expressing the *update* of a file, *file*, record with key k to a new data part, d. The wording above is a bit "dangerous". Nothing "happens" to file *file*. All we are expressing is some other file which is like *file* is, except that whatever the record with key k had as data part in *file*, in this other file the record with key k ("now") has data part d.

If k is the key of some record in *file* then the data part of that record can be *read*:

46. $file(k)$

To express *deletion* of the record with key k from a file *file* we write:

47. $file \backslash \{k\}$

Expressions (44.-47.) were "informal". More "formal", "closed-form" descriptions of these operations could be:

48. $\underline{\mathbf{type}}$: $Write: Data \times FILE \rightarrow FILE \times Key$
.1 $\underline{\mathbf{type}}$: $Update: Record \times FILE \xrightarrow{\sim} FILE$
.2 $\underline{\mathbf{type}}$: $Read: Key \times FILE \xrightarrow{\sim} Data$
48.3 $\underline{\mathbf{type}}$: $Delete: Key \times FILE \rightarrow FILE$

where we assume (ie. edict!) that the write operation itself shall generate, use and return a suitable key:

49. $\mathit{Write}(d,\mathit{file}) \triangleq$
.1 $\quad (\underline{\mathbf{let}}\ k \in \mathit{Key} \backslash \underline{\mathbf{dom}}\ \mathit{file}\ \underline{\mathbf{in}}$
49.2 $\quad (\mathit{file} \cup [k \mapsto d], k))$

50. $\mathit{Update}((k,d),\mathit{file}) \triangleq$
.1 $\quad \underline{\mathbf{if}}\ k \in \underline{\mathbf{dom}}\ \mathit{file}$
.2 $\quad \underline{\mathbf{then}}\ \mathit{file} + [\, k \mapsto d \,]$
50.3 $\quad \underline{\mathbf{else}}\ \ \underline{\mathbf{undefined}}$

We could have defined update unconditionally — to just contain (45.) as the function definition body. Doing so would, however, lead to "update" usable also for "write" purposes — as the map override operation, +, does not require, in this case, *k*, to be already in the domain of, in this case *file*.

51. $\mathit{Read}(k,\mathit{file}) \triangleq$
51.1 $\quad \underline{\mathbf{if}}\ k \in \underline{\mathbf{dom}}\ \mathit{file}\ \underline{\mathbf{then}}\ \mathit{file}(k)\ \underline{\mathbf{else}}\ \underline{\mathbf{undefined}}$

52. $\mathit{Delete}(k,\mathit{file}) \triangleq$
52.1 $\quad \underline{\mathbf{if}}\ k \in \underline{\mathbf{dom}}\ \mathit{file}\ \underline{\mathbf{then}}\ \mathit{file} \backslash \{k\}\ \underline{\mathbf{else}}\ \underline{\mathbf{undefined}}$

Similar remarks, as for update, apply to read and delete. Applying a map to an argument not in its domain "automatically" yields undefined — but we express ourselves "defensively". And: deleting a non-existing record doesn't change anything: however we prefer to be told of attempts to delete non-existing records, and use the undefined clause as a future reference point for inserting useful diagnostics when actually implementing eg. this file system!

The expressions:

53. $\mathit{file}_1 \cup \mathit{file}_2$
.1 $\mathit{file}_1 + \mathit{file}_2$
.2 $\mathit{file}_1 \setminus \underline{\mathbf{dom}}\ \mathit{file}_2$
53.3 $\mathit{file}_1 \mid \underline{\mathbf{dom}}\ \mathit{file}_2$

can, as a suggestion, be understood as modelling the following transactions: (53.) The merging of two files of distinctly keyed records. (53.1) The update of a master file, file_1, with a (daily) transaction file, file_2 — the latter permitted, now, to contain records with keys not in file_1, ie. "new" records "to be written" onto the new master file! (53.2) expresses the deletion of all those records from file_1 whose keys are keys of records in file_2 — of course nothing is physically, or actually, "deleted" — as before (53.2), and for that matter (53.-53.3 incl.). just expresses ("new") files. (53.2) denotes a file which is like file_1 is, except that it does not "contain" those records of file_1 which have keys in common with records of file_2. Finally (53.3) expresses a file which is like file_1 is, except it only has those records whose keys are in common with records of file_2.

As we did with set- and tuple-oriented abstractions of file systems (section 2.4, respectively 3.4.3), we now show imperative versions of some of the above operations:

54. $\underline{\mathbf{dcl}}\ \mathrm{file} := [\,]\ \underline{\mathbf{type}} : \mathit{FILE}$
54.1 $\Sigma = \mathrm{file} \underset{m}{\rightarrow} \mathit{FILE}$

55. $\underline{\mathbf{type}} : \mathit{write}\!: \mathit{Data} \rightarrow (\Sigma \rightarrow \Sigma \times \mathit{Key})$
.1 $\underline{\mathbf{type}} : \mathit{update}\!: \mathit{Record} \rightarrow (\Sigma \xrightarrow{\sim} \Sigma)$
.2 $\underline{\mathbf{type}} : \mathit{read}\!: \mathit{Key} \rightarrow (\Sigma \xrightarrow{\sim} \mathit{Data})$
55.3 $\underline{\mathbf{type}} : \mathit{delete}\!: \mathit{Key} \rightarrow (\Sigma \xrightarrow{\sim} \Sigma)$

```
56.    write(d) ≜
  .1     (def k ∈Key\dom c file;
  .2     file := c file ∪ [k↦d];
56.3     return  k)

57.    update(k,d) ≜
  .1     if k ∈dom c file
  .2     then file := c file +[k↦d]
57.3     else error

58.    read(k) ≜
  .1     if k ∈dom c file
  .2     then (c file)(k)
58.3     else error

59.    delete(k) ≜
  .1     if k ∈dom c file
  .2     then file := (c file)\{k}
  .3     else error
```

Given:

```
60.    type : F: FILE → FILE
60.1   type : R: Data → Data
```

we can define file system and file (ie. "record system") processing functions:

```
61.    type : afP: (FILE → FILE) × FS → FS
61.1   afP(F,fs) ≜ [f ↦ F(fs(f)) | f ∈dom fs]

62.    dcl fs     := [] type : FS
63.    Σ          = fs ⇸ FS

64.    type : ifP: (FILE → FILE) → (Σ → Σ)
  .1   ifP(F) ≜
  .2     (def fns: dom c fs;
  .3     for all f ∈fns do fs := c fs + [f ↦F((c fs)(f))])

65.    type : arP: (Data → Data) × FS → FS
  .1   arP(R,fs) ≜
65.2     [f ↦ [k↦R((fs(f))(k)) | k∈dom (fs(f))]| f∈dom fs]

66.    dcl fs     := [] type : FS;
67.    dcl file   := [] type : FILE;
68.    Σ          = (fs → FS) ⊎ (file ⇸ FILE)
```

```
69.   $\underline{\text{type}}$ : $irP$: $(Data \rightarrow Data) \rightarrow (\Sigma \rightarrow \Sigma)$
 .1   $irP(R) \triangleq$
 .2       $(\underline{\text{def}}\ fns$: $\underline{\text{dom}}\ \underline{c}$ fs;
 .3       $\underline{\text{for}}\ \underline{\text{all}}\ f \in fns\ \underline{\text{do}}$
 .4           (file := [];
 .5           $\underline{\text{def}}\ ks$: $\underline{\text{dom}}\ ((\underline{c}$ fs$)(f))$;
 .6           $\underline{\text{for}}\ \underline{\text{all}}\ k \in ks\ \underline{\text{do}}$
 .7               file := $\underline{c}$ file $\cup\ [k \mapsto R(((\underline{c}$ fs$)(f))(k))]$;
69.8          fs := $\underline{c}$ fs + $[f \mapsto \underline{c}$ file$]))$
```

4.3.3 The Map Modelling Principle

The question to be answered is this: when should we use the map data type in our abstractions? The answer goes somewhat like this:

> When the object being subject to abstraction possesses a composite property such that it can be regarded as a finite collection of uniquely distinguished elements then a map abstraction seems reasonable.
>
> If, furthermore, manipulations of the object may involve searching for a distinguished element, or extending the object with yet another such new, uniquely distinguishable element, etc., then a map abstraction seems further motivated.

As was discussed earlier, the above modelling principle is a guide-rule, etc. The discussion at the end of subsection 4.1.2 applies equally well here!

The above rules, in actual modelling situations "translates" as follows, in two ways: (i) if you are 'told': *Some facility consists of an unordered collection of distinct, uniquely distinguished things* etc., then you should consider whether a model based on a map abstraction is otherwise appropriate; and (ii) vice-versa: in deciphering somebody else's unstructured, informal, ad-hoc, incomplete and possibly even inconsistent "specification", you should analyze that description with spectacles viewing "it" (the thing spoken about by the "specification") from the point of view of: is a map abstraction an appropriate choice? (You may find, in the latter case (ii) that it either fits, or does not; if not, then perhaps any of the other composite data types [sets, tuples, trees] may be used.)

4.4 Tree Abstractions

4.4.1 Programming Language Constructs — an Example

The basic idea is to abstract from any concretely written form. How eg. statements are written:

```
70.   var := expression
 .1   $\underline{\text{let}}$ var $\underline{\text{be}}$ expression
 .2   $\underline{\text{assign}}$ expression $\underline{\text{to}}$ var
 .3   expression $\rightarrow$ var
70.4  $\underline{\text{compute}}$ expression $\underline{\text{in}}$ var
```

or some such way, cannot be important. At least not when the "real" issue is "what does assignment mean?". Common to all of the above (70.-.1-.2-.3-.4), ie. the case of the assignment statement, is that it consists of two parts: one being the variable reference (denoting the location to which the assignment update shall occur), the other being an expression (denoting ... etc.). Thus, instead of writing some BNF grammer, like:

71. $<$ Assignment $>$ ::= $<$ Variable $>$:= $<$ Expression $>$

which denotes text-string generation or analysis for the first (70.) of the above concrete forms, we write:

72. $Asgn\ ::\ Vid \times Expr$

Either of the above four concrete text string representations of assignment statements are now abstracted by the one abstract tree expression:

73. $\underline{\textbf{mk-}}Asgn(var, expression)$

where *var* is the abstraction of 'var', and *expression* the abstraction of 'expression'.

We have just illustrated the representational abstraction of assignment statements. We now go on to illustrate the representational abstraction of other, typical, source language statements:

74. $If \quad :: Expr \times \underline{\textbf{s-}}cons{:}Stmt \times \underline{\textbf{s-}}alt{:}Stmt$

abstracts the Domain of if-then-else statements, which syntatically consists of an expression and two (the consequence, and the alternative) statements.

The 'while loop' statement Domain is (eg.) abstracted as:

75. $Wh \quad :: Expr \times Stmt^{+}$

That is: a while-loop apparently consists of an expression and a statement list — concretely one such while loop statement could look like:

76. '$\underline{\text{while}}$ e $\underline{\text{do}}$ s_1; s_2; ...; s_n $\underline{\text{od}}$'

or like:

77. " $\underline{\text{DO}}$ $\underline{\text{WHILE}}$ (e); s_1; s_2; ...; s_n; $\underline{\text{END}}$ "

Observe that although we have written the *Expr* before the $Stmt^{+}$ that does not always mean that in a(ny or some) concrete representation the corresponding concrete text for *Expr* precede text for $Stmt^{+}$. The example of the contrary is the 3rd, 4th, and 5th example (70.2-.3-.4) of concrete assignments versus the abstract Domain of *Asgn*.

Observe also that whereas a BNF grammar generally specifies text strings (strings of characters), as opposed to eg. phrase-tree structures (ie. text strings annotated with their underlying phrase-structure), our tree Domain equations specify structured, composite, objects, ie. objects not subject to any "parsing" or analysis with respect to which structure they (might) have. This last point is often overlooked, or missed. Tree Domain specifications of the syntactic constructs of an(y) object language is a specification of already analyzed (parsed) objects, ie. a specification of parse-trees rather than text strings. As we shall later see, we also use the tree data type for other than specifying (and manipulating) syntactic objects.

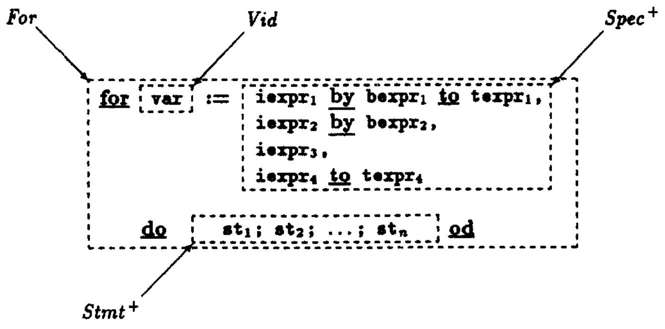
To round up our example of illustrating the statement constructs of an ALGOL-like language we throw in some further examples, including some concerned with expressions:

78. $For \quad :: Vid \times Spec^{+} \times Stmt^{+}$

is intended to define the Domain of abstract, ALGOL-60—like "for loops", a concrete, schematic example of which is shown in figure 1.

The dashed boxes enclose various, so designated phrase type components. (We shall later, in volume III chapter 4, explain and formalize the semantics of Algol 60–like for loops.) (The above dashed boxes and italicized words (at the root of arrows) are extraneous to the concrete example, but should illustrate the parts corresponding to the abstract tree Domain *For*.) Thus:

Figure 1: A Schematic, General For-Loop



```
for var := iexpr1 by bexpr1 to texpr1,
           iexpr2 by bexpr2,
           iexpr3,
           iexpr4 to texpr4
   do st1; st2; ...; stn od
```

79. $\mathit{Spec} = \mathit{BT\text{-}Spec} \mid \mathit{B\text{-}Spec} \mid \mathit{Expr} \mid \mathit{T\text{-}Spec}$
80. $\mathit{BT\text{-}Spec} :: \mathit{Expr} \times \mathit{Expr} \times \mathit{Expr}$
81. $\mathit{B\text{-}Spec} :: \mathit{Expr} \times \mathit{Expr}$
82. $\mathit{T\text{-}Spec} :: \mathit{Expr} \times \mathit{Expr}$
83. $\mathit{Expr} = \ldots$

Since we apparently assume that *B-Spec* (only by) and *T-Spec* (only to) specifications imply distinct semantics we must enable such a distinction syntactically. This distinction is afforded by the axiom on tree Domains: even though we use the same expression e_1 and e_2 in both by and to specifications:

84. $\underline{\mathbf{mk\text{-}}}\mathit{B\text{-}Spec}(e_1,e_2)$,
84.1 $\underline{\mathbf{mk\text{-}}}\mathit{T\text{-}Spec}(e_1,e_2)$.

By the mere distinctness of the identifiers *B-Spec* and *T-Spec* the above two tree objects are distinct, and hence distinguishable.

The Domain of all statements is referred to above as *Stmt*, its proper definition is:

85. $\mathit{Stmt} = \mathit{Asgn} \mid \mathit{If} \mid \mathit{Wh} \mid \mathit{For} \mid \ldots$
86. $\mathit{Asgn} :: \mathit{Vid} \times \mathit{Expr}$
87. $\mathit{If} :: \mathit{Expr} \times \mathit{Stmt} \times \mathit{Stmt}$
88. $\mathit{Wh} :: \mathit{Expr} \times \mathit{Stmt}^+$
89. $\mathit{For} :: \mathit{Vid} \times \mathit{Spec}^+ \times \mathit{Stmt}^+$

etcetera. Among expressions we have simple variables, constants, pre-, in- and suffix-expressions, conditional expressions, etcetera:

90. $Expr = Var \mid Const \mid Pre \mid Inf \mid Suf \mid Cond \mid \ldots$
91. $Var :: Vid$
92. $Const = Intg \mid Bool \mid \ldots$
93. $Pre :: Pop \times Expr$
94. $Inf :: Expr \times Iop \times Expr$
95. $Suf :: Expr \times Sop$
96. $Cond :: \underline{s}\text{-}tst{:}Expr \times \underline{s}\text{-}cons{:}Expr \times \underline{s}\text{-}alt{:}Expr$
97. $Pop = \underline{\text{MINUS}} \mid \underline{\text{NOT}} \mid \ldots$
98. $Iop = \underline{\text{ADD}} \mid \underline{\text{SUB}} \mid \underline{\text{MPY}} \mid \underline{\text{DIV}} \mid \underline{\text{AND}} \mid \underline{\text{OR}} \mid \ldots$
99. $Sop = \underline{\text{FAC}} \mid \ldots$
100. $Intg :: \mathtt{INTG}$
101. $Bool :: \mathtt{BOOL}$

Some comments are in order: instead of defining syntactic designators for integers, ie. instead of defining numerals, and instead of defining similar designators for truth values, we prescribe the denoted objects directly! That is we abstract numerals by their denoted values: integers (or rational numbers, etc.). And we abstract the syntactic markers designating truth values by their denoted values.

Note also that we have just used the meta-language quotation data type: the underlined words, or identifiers, listed in the *Pop, Iop* and *Sop* Domain definitions, are intended to abstract the operator symbols which in some source language might be represented by $-, \neg, \ldots, {}^{+}, -, {}^{*}, /$, and, or, |, ... , !. We refer to section 3.6 volume I chapter 3, for a concise treatment of this so-called QUOT data type. Suffice it here to repeat that QUOT objects stand for themselves.

Finally we note an "extreme" case of a (cartesian product, or tree) Domain expression involving three occurrences of the same Domain identifier: $Expr \times Expr \times Expr$. For ease of (future) reference, ie. as an aid in documentation, hinting at the various rôles the individual *Expr*essions of conditional *Expr*essions serve, we have "annotated" the Domain definition by suitably chosen mnemonics for the sub-component functions which select the: "$\underline{\text{t}}$e$\underline{\text{st}}$", "$\underline{\text{cons}}$equence" and "$\underline{\text{alt}}$ernative" expressions.

Our final syntactic Domain definition is intended to bring the whole apparatus of set, tuple, map and tree data type abstractions together, into one single Domain definition. The point is to illustrate how abstract we may wish to go when defining even syntactic objects, objects for which we are used to a rather pedantic, concrete representation. The case in point is the ALGOL-like language construct "blocks". To carry our message as forcefully and clearly as possible, we think of a block as consisting of three things: declaration of variables, definition of procedures and a statementlist body.

102. $Block :: Vars \times Procs \times Body$

We think, in this, very simplifying, case, of variables being declared by just listing their identifiers (no type or other information), and we think of the order of listing of variable identifiers to be (semantically) immaterial:

103. $Vars = Vid\text{-}\underline{\text{set}}$

We think of procedure definitions as consisting of two parts: a procedure identifier (the definiendum) and the rest: formal parameter specification and a procedure body (which is a block), and we call, ie. name the Domain of these rest's, *Prc*. Since we think of no two procedures of a block to have the same identifier we abstract the procedure definitions as a map from identifiers to "rests":

104. $Procs = Pid \underset{m}{\rightarrow} Prc$

Finally:

105. $Body = Stmt^{+}$

By substituting the last three definitions (back) into that of *Block* we get

106. *Block* $::$ *Vid*-set $\times$ *(Pid* $\overrightarrow{m}$ *Prc)* $\times$ *Stmt*$^+$

where we "smuggled" in some (precedence-breaking-, or at least "text"-grouping-) parentheses around *Procs*. Here they cause no change in what is being defined. The above, last, *Block* definition wraps up all four abstract data types of the meta-language in one definition: trees, sets, maps and tuples. Although actual, ie. concrete representation of blocks syntactically must be linear, ie. ultimately ordered (in extreme: tuples of characters), we have here, in our abstraction, not only abstracted away concrete syntactic markers such as keywords and other delimiters, and ordering of sub-phrases, but two additional, similar, things have been obtained: the fact that no two variable declarations are (usually) allowed to introduce the same identifier (twice), and the fact that no two (or more) procedure definitions are (usually) allowed to use, ie. define the same procedure identifier (twice or more). We say that some of the *context sensitive conditions* of eg. a BNF specification have been solved, ie. done away with, in our, more abstract Domain specifications. Not all such context conditions can, however, be solved merely by using abstraction.

To wrap up some, but not all loose ends of the *Block* Domain definition we partially complete:

107. *Prc* $::$ *Fid*$^*\times$ *Block*
108. *Stmt* $= \ldots |$ *Call*
109. *Call* $::$ *Pid* $\times$ *Expr**

4.4.2 The Tree Modelling Principle

The question now to be answered is this: when, in specifying software abstractly, do we use the tree data types? The answer goes something like this:

> When the object to be modelled — of some external, "real" world, or of some programming world, possesses a composite structure, and when that structure is fixed, ie. consists of a fixed number of components (of arbitrary composite or atomic nature), then a tree abstraction seems possible.
>
> If, further, manipulation of the object being modelled consists basically in taking it apart, into its constituent components, and comparing two structures (for equality, for example), then the tree abstraction seems justified.

Etcetera.

5 Applicative and Imperative Definitions

A model, a specification, is applicative iff it is expressed solely in the applicative style, ie. based only on applicative constructs. A model is imperative if it contains at least one imperative construct.

Several examples have been give above using either style of definition. Hence:

What determines our choosing either the applicative or the imperative style? The question to be answered here is: when do we choose to introduce global state variables?

The answer, is based on pragmatics, has several parts, and covers several facets, and goes somewhat like this:

1. If the concept modelled (i) exhibits sequentialism, ie. that certain object manipulations are done in certain orders, and (ii) if past creation of object values. once consumed, ie. once used in the subsequent (ordered, sequential) creation of new values, are never again used, then a meta state may be a proper thing to introduce. We shall illustrate this rule in volume IV chapter 3 on sequentialism!

   There are actually two notions involved here: (i) sequentialism and (ii) states. They obviously intertwine. Sequentialism cannot go without a state.

2. The balance between having few versus many global variables is a choice determined by stylistic concerns: many variables lead to a need for few parameters to functions, and to few components of returned values. Few variables lead to many parameters and many result components. The more

global variables that are used in any one function definition, the more side-effects are "potentially" hidden.

6 Denotational and Computational Definitions

Definition 4 *A denotational semantics definition of, say a programming language, assigns to each primtive construct of the language (viz.: identifiers of variables, labels, procedures, etc.) a mathematical function (the denotation of the identifier), and otherwise expresses the semantics of composite constructs (homomorphically) as a function of the semantics of each of the components of such composite constructs.*

Thus a denotational semantics ascribes functions, usually input/output functions, that describe the i/i function of constructs.

Definition 5 *A computational semantics, in contrast, describes the execution behaviour of programming language constructs in terms of state sequences undergone while computing according to program (construct) prescription.*

We illustrate the important notions of Denotational and Computational Semantics by giving semantics to a common language of expressions.

6.1 Syntactic Domains

Our example source language consists, syntactically, of expressions. Expressions are either constants, identifiers or pre- or infix operator/operand expressions. Constants are (for simplicity) integers. Identifiers are just that. Prefix expressions has two parts: a monadic operator and an expression. Infix expressions has three parts: a dyadic operator and two expressions. Monadic (dyadic) operators are "plus", "minus", "factorial", etc. (and "add", "subtract", "multiply", etc.):

110. $Expr \;=\; Const \mid Id \mid Pre \mid Inf$
111. $Const \;::\; \texttt{INTG}$
112. $ID \;::\; \texttt{TOKEN}$
113. $Pre \;::\; MOp \times Expr$
114. $Inf \;::\; Expr \times DOp \times Expr$
115. $MOp \;=\; \underline{\text{PLUS}} \mid \underline{\text{MINUS}} \mid \underline{\text{FACT}} \mid \ldots$
116. $DOp \;=\; \underline{\text{ADD}} \mid \underline{\text{SUB}} \mid \underline{\text{MPY}} \mid \ldots$

(The above equations display, or exhibit, almost negligeable representational abstraction: little "room" is given in this example for doing abstraction!)

We observe how expressions have been recursively defined — just as would be expected in a standard, concrete BNF grammar definition.

6.2 Semantic Domains

Only constants have been representationally abstracted: instead of specifying numerals, we (directly) specify the integer numbers denoted.

Identifiers occurring in expressions are bound to integer values, in something we shall call an environment:

117. $\rho\colon ENV = Id \underset{m}{\rightarrow} \texttt{INTG}$

The primitives of the language are: constants, identifiers and operators. Constants denote themselves. Identifiers denote integers — with their denotation being recorded in the environment.

6.3 The Denotational Semantics

6.3.1 Auxiliary Denotation Functions

Operators denote certain arithmetic functions.

```
118.    DenOp(op) ≜
  .1      cases op :
  .2        PLUS   → $\lambda x.x$
  .3        MINUS  → $\lambda x.\text{-}x$
  .4        FACT   → $\lambda x.x!$
  .5        ...    → ...
  .6        ADD    → $\lambda x.\lambda y.x+y$
  .7        SUB    → $\lambda x.xy.x\text{-}y$
  .8        MPY    → $\lambda x.\lambda y.x\times y$
  .9        ...    → ...
  .10   type : (MOp → (INTG→ INTG)) |
118.11    (DOp → (INTG× INTG→ INTG))
```

In order that the semantic function can find the meaning (i.e. value) of an identifier it must refer to an environment which is therefore an argument to the semantic function.

6.3.2 The Semantic Elaboration Functions

Without much ado we present the semantic function which, since expressions were recursively defined, itself is recursively defined.

```
119.    Val-Expr(e)$\rho$ ≜
  .1      cases e :
  .2        mk-Const(i) → i,
  .3        mk-Id(t)    → $\rho$(e),
  .4        mk-Pre(m,e')→ DenOp(m)(Val-Expr(e')$\rho$),
  .5        mk-Inf(l,d,r) → DenOp(d)(Val-Expr(l)$\rho$, Val-Expr(r)$\rho$)
119.6   type : Expr → (ENV $\xrightarrow{\sim}$ INTG)
```

The functions *M* and *F* alluded to in the introduction (section 10.1) can now be stated: *M* is *Val-Expr* when the syntactic construct is an expression, and is *DenOp* when it is an operator. *F* is functional composition for the case of prefix expressions:

120. $F(DenOp(m), Val\text{-}Expr(e)\rho) =$

120.1 $DenOp(m \underbrace{)(}_{\text{function composition}} Val\text{-}Expr(e)\rho)$

F is the composite of the "pairing" function with functional composition when the composite is an infix expression:

121. $F(Val\text{-}Expr(l)\rho, DenOp(d), Val\text{-}Expr(r)\rho) =$

121.1 $DenOp(d \underbrace{)(}_{\text{function composition}} Val\text{-}Expr(l)\rho \underbrace{,}_{\text{pairing}} Val\text{-}Expr(r)\rho)$

That is: we view the prefixing of an expression with a monadic operator, respectively the infixing of two expressions with a dyadic operator as (syntactic) operators — not explicitly written. And we then assign the meaning:

122. $\lambda f.\lambda x.f(x)$

to the (invisible) prefixing operator, and:

123. $\lambda x.\lambda f.\lambda y.f(x,y)$

as the meaning of the (invisible) infixing operator.

Instead of "juggling" around with the *DenOp* function and with what to us are rather convolute formulae of *Val-Expr* we syntactically sugar *Val-Expr* while factoring *DenOp* into the new *V-Expr*:

```
124.    V-Expr(e)ρ ≜
  .1      cases e :
  .2        mk-Const(i)     → i,
  .3        mk-Id(t)        → ρ(e),
  .4        mk-Pre(m,e'     → (let v = V-Expr(e')ρ in
  .5                           cases m :
  .6                             PLUS   → v,
  .7                             MINUS  → -v,
  .8                             FACT   → v!),
  .9        mk-Inf(l,d,r)   → (let lv = V-Expr(l)ρ,
  .10                              rv = V-Expr(r)ρ in
  .11                          cases d :
  .12                            ADD   → lv+rv,
  .13                            SUB   → lv-rv,
  .14                            MPY   → lv×rv,
  .15                            ...   → ...)
124.16  type : Expr → (ENV ⥲ INTG)
```

We are finally ready to summarize the type of the denotation of expressions, whether constants, identifiers or operator/operand expressions. That (general) type can be read directly from the type of the semantic function (119 or 124) above. The type of the meaning of an expression, i.e. its semantic type, is that of a function from environments to integers:

125. *Expr*: $ENV \xrightarrow{\sim}$ INTG

The function is partial in that expression identifiers not in the domain of the environment lead to undefinedness. For a constant, **mk-***Const(i)*, expression the function is the constant function which "maps" any environment, ρ, into i. For an identifier, **mk-***Id(t)*, expression, e, the function maps any environment, ρ, into the integer, $\rho(e)$, which that identifier is associated with in those environments. If the identifier is not in the environment **undefined**is yielded. For the remaining expressions we refer the reader to the formulae of e.g. (124.), from which we also "read" the meaning functions of the two previous sentences.

6.3.3 An Extension

For the sake of making the computational semantics example a bit more interesting than it would otherwise be with the present source language of expressions, we extend this language. The extension amounts to the introduction of conditional expressions:

126.	*Expr*	$= \ldots \mid Cond$
127.	*Cond*	$:: Expr \times Expr \times Expr$

where we think of the semantics of "**if** e_t **then** e_c **else** e_a" as really specifying: "**if** $e_t=0$ **then** e_c **else** e_a"! Thus:

```
128.   V-Expr(e)ρ≜
  .1     cases e :
  .2       mk-Cond(t,c,a)  → (let b = V-Expr(t)ρ in
  .3                          if b=0
  .4                          then V-Expr(c)ρ
  .5                          else  V-Expr(a)ρ),
  .6       ...
```

Thus F of a conditional expressions' semantic is that of "delaying" the evaluation of either the consequence- or the alternative expression till the value of the test expression has been obtained. More precisely:

129. $M(t,c,a)$

.1 $= F(M(t),M(c),M(a))$

129.2 $= \lambda\rho.(\underline{\textbf{if}}\ M(t)\rho=0\ \underline{\textbf{then}}\ M(c)\rho\ \underline{\textbf{else}}\ M(a)\rho)$

whereby F is expressible as:

130. $\lambda\rho.\lambda m_t.\lambda m_c.\lambda m_a.\underline{\textbf{if}}\ m_t(\rho) = 0\ \underline{\textbf{then}}\ m_c(\rho)\ \underline{\textbf{else}}\ m_a(\rho)$

where m_t, m_c and m_a now are the "meanings" of the "correspondingly" named syntactic objects: t, c and a. Observe how the "delay" is afforded by the "encapsulation" of final evaluations of c and a.

6.4 A Computational Semantics

6.4.1 Introduction

The basic idea of the example of the next 2 sections is that of realizing the recursion of *V-Expr* of sections 10.3-4 by means of *stacks* . Many realizations of the recursion of *V-Expr* are possible. We will, rather arbitrarily, select one. Volumes IV-V-VI will explore the unfolding of recursion onto stacks in a more systematic fashion.

Before proceeding into a description of which stacks to create and how they are used we note that our stacks are not to be used for sorting out precedence of operators. Since we work only on abstract syntactic objects, all such precedence has already been resolved, and is "hidden" in the (invisibly) parenthesized sub-expressions.

Thus we remove recursion in the function definition (of *V-Expr*) by introducing (one or more) stacks. At the same time we change our definitional style from applicative to imperative. This is not an intrinsic consequence of choosing stacks, but a pragmatic one. In doing so we can, at the same time simply change the recursive function definitions into iterative. The imperative/iterative nature of the resulting definition further gives it an air of being "mechanical".

6.4.2 The Computational State

One stack is the *value stack* . It is motivated by the "stacking" of temporaries (cf. (124.4), (124.8-124.9)) due to recursion in *V-Expr*.

Another stack is a *control* , or *operator/operand-expression* stack. It is motivated by recursion over syntactical expression objects.

Thus we make two decisions: first to state the model imperatively, in terms of some globally declared variables. Then to express the computational semantics in terms of two stack variables and a constant environment.

```
131.    dcl opestk := < > type : (MOp | DOp | Expr | ITE )*,
  .1        valstk := < > type : INTG*;
131.2   let env = [...] in...
```

Why we made those two, and not other, among quite a few other possible, decisions will not be explained much further! We reserve such discussions to volumes IV and V.

In our computational semantics, as imperatively stated, we must necessarily choose an elaboration order for operand expressions of infix expressions. This order was left "unspecified" by *V-Expr* of section 10.3.

6.4.3 Motivating the Control Stack

The idea of the operator/operand stack is now that the topmost element is either an expression, to be evaluated, or an operator to be applied to either the operator/operand or to the value stacks.

If the top of the operator/operand stack is an expression then it is either elementary or composite. If it is elementary, i.e. a constant or an identifier then the associated value is pushed onto the value stack, while the expression is being popped off the operator/operand stack. If it is composite, i.e. a prefix, infix or conditional expression, then those expressions are decomposed, with the decomposition replacing it on the operator/operand stack. Hence the control stack will consist of a sequence of operators and their operands, in what turns out to be some variant of a so-called post-fix polish "notation".

1: A *prefix expression* is replaced by two elements on this stack: the monadic operator and the (sub-) expression (on top).

2: An *infix expression* is replaced by three elements: the dyadic operator and the two (sub-) expressions (in some order, on top).

3: A *conditional* expression is replaced by four elements, in order from top towards bottom: the test expression, a "meta-"operator (ITE), and the consequence and alternative expressions — the latter two in arbitrary, but fixed, order. The idea of the ITE operator will be explained presently.

4: If the top of the operator/operand stack is a *monadic operator*, then the denoted operation is applied to the top of the value stack. (Thus if the operator is MINUS the top of the value stack is replaced by its complemented ("negative") value.) [It follows from the operator/operand stack manipulations that the value stack top is the value of the expression to which the monadic operator was once prefixed.]

5: If the top of the operator/ operand stack is a *dyadic operator*, then the denoted operation is applied, in an appropriate way, to the two topmost values of the value stack — with the result replacing these values.

6: Finally if the operator/operand stack top element is ITE then it means that the value of the test expression of the conditional expression, whose manipulation gave rise to this ITE operator, is on the top of the value stack. If it, the latter, is 0 then we compute only the consequence expression, otherwise we compute only the alternative expression. These are the next two elements on the operator/operand stack. The appropriate one is thrown away together with the value stack top.

6.4.4 The Elaboration Functions

Computation proceeds based, as always, on the top element of the operator/operand stack. And computation proceeds as long as there are elements on the operator/operand stack. When it becomes empty the computed value is the top value of the value stack. The function informally described in this paragraph is called *Compute*, it is defined formally below.

Let us call the function which transforms the system state dependent on the top of the operator/operand stack for *Transform*, then:

```
132.    type : Compute: Expr → (Σ ⥲ Σ× INTG)
  .1    type : Transform: Σ ⥲ Σ

133.    Σ  = opestk →m (MOp | DOp | Expr | ITE )*
134.         ∪ valstk →m INTG*
```

```
135.    Compute(e) ≜
  .1      ( opestk := <e>;
  .2        while c opestk ≠ < > do Transform();
  .3        c hd valstk )
```

To facilitate the statement of *Transform* we define four *auxiliary* stack *functions* :

```
136.    PopO() ≜ (def oe : hd c opestk;
  .1          opestk := tl c opestk;
  .2          return oe)
136.3   type : Σ→ (Σ × (MOp | DOp | Expr | ITE ))

137.    PopV() ≜ (def v : hd c valstk;
  .1          valstk := tl c valstk;
  .2          return v)
137.3   type : Σ→ Σ × INTG

138.    PushO(oel) ≜ opestk := oel ⌢(c opestk)
138.1   type : (MOp | DOp | Expr | ITE )*→ (Σ→ Σ)

139.    PushV(v) ≜ valstk := <v>⌢c valstk
139.1   type : INTG→ (Σ→ Σ)
```

Now to the main function:

```
140.    Transform() ≜
  .1      def oe : PopO();
  .2      cases oe :
  .3        mk-Const(i)        → PushV(i),
  .4        mk-Id(t)           → PushV(env(oe)),
  .5        mk-Pre(m,e')       → PushO(<e',m>),
  .6        mk-Inf(l,d,r,)     → PushO(<r,l,d>),
  .7        mk-Cond(t,c,a,)    → PushO(<t,ITE ,c,a,>),
  .8        MINUS              → (def v : PopV();
  .9                              PushV(-v)),
  .10       ...                → ...
  .11       ADD                → (def lv : PopV();
  .12                             def rv : PopV();
  .13                             PushV(lv+rv)),
  .14       ...                → ...
  .15       ITE                → (def b : PopV();
  .16                              def c : PopO();
  .17                              def a : PopO();
  .18                              PushO(if b=0 then c else a))
```

6.4.5 A Discussion

We observe that the above definition does not satisfy the denotational principle. Instead we should get a rather operational "feeling" for how one might mechanically pursue an interpretation of expressions — resulting, after some iterations, rather than recursions, in its value.

7 Hierarchical and Configurational Developments and Presentations

7.1 Definitions

Definition 6 *By 'construction' we here mean the process of developing a specification, or, in general, the process of developing software.*

Definition 7 *By 'presentation' we here mean the documentation resulting from construction, and presented to the readers.*

Definition 8 *'Hierarchical' (or 'hierarchal') is basically an intellectual concept, and conjures that something should be conceived from the top-down.*

Definition 9 *'Configurational' is basically a a mechanical concept, and conjures that something should be conceived from the bottom-up.*

Well-known, familiar artifacts, such as would be yet another `Pascal/Algol 60` like programming language, or a similarly classically conceived relational data base system, — such "well-known" notions — can be both hierarchically developed and presented. Rather "newish" concepts, as might for example be a so-called `Petri-Net` based office automation system architecture based on some form (and document) flow concept, might be both developed and presented in a configurational manner. Finally there may be software architectures that are configurationally developed, but once developed, and hence well-understood, they might be hierarchically presented.

The example of the Tuple Abstraction section was configurationally developed and presented. The example Denotational Semantics was partly hierarchically, partly configurationally presented.

8 Specification Components

The basic components of system models are:

1. semantic Domain equations,
2. invariant predicate definitions (over semantic Domains),
3. syntactic Domain specifications,
4. well-formedness predicate definitions (over syntactic Domains),
5. semantic elaboration function type definitions,
6. semantic function (body) definitions.
7. and usually a number of auxiliary functions

The above examples abundantly illustrates this decomposition of a specification.

8.1 Semantic Domains

We give a hierarchical presentation.

8.1.1 A Data Management System State — an Example

The state of a simple Data Management System consists of a Dictionary and a File System. The Dictionary maps File Names to File Types, and the File System maps File Names to Files. Files are sequences of groups, each group being a sequence of either Boolean, Integer or Character Values. The File Type of a File describe, for each group its Type, whether BOOLEAN, INTEGER, or CHARACTER, and its maximum sequence length. See figure 2.

Figure 2: Abstract Syntax for a Data Management System State

141. $DMS \;::\; DICT \times DATA$
142. $DICT \;=\; Fn \xrightarrow{m} FTyp$
143. $FTyp \;=\; (DTyp \times Length)^{+}$
144. $DTyp \;=\; \underline{\text{BOOLEAN}} \mid \underline{\text{INTEGER}} \mid \underline{\text{CHARACTER}}$
145. $Length \;=\; \mathbf{N}_1$
146. $DATA \;=\; Fn \xrightarrow{m} FILE$
147. $FILE \;=\; Data^{+}$
148. $Data \;=\; Bool \mid Intg \mid Char$
149. $Bool \;=\; \mathtt{BOOL}^{+}$
150. $Intg \;=\; \mathtt{INTG}^{+}$
151. $Char \;=\; \mathtt{TOKEN}^{+}$

Figure 3: Data Management System State Invariant

```
152.    inv-DMS(mk-DMS(d,fs)) ≜
  .1        (dom fs ⊆ dom d)                                   All files are defined.
  .2        ∧(∀fn ∈ dom fs)                                              For each file,
  .3              (let ft = d(fn),
  .4                   file = fs(fn) in
  .5                   (len ft = len file)                its number of groups is correct,
  .6                   ∧(∀i ∈ ind ft)                                   For each group,
  .7                       (let (t,l) = ft[i],
  .8                            grp = file[i] in
  .9                            (len grp ≤ l)                       it is within length,
152.10                          ∧SameTyp(t,hd grp)                and of right value type

153.    SameTyp(t,v) ≜
  .1        ((t=BOOLEAN)∧is-BOOL(v)) ∨                         BOOL value iff BOOL type
  .2        ((t=INTEGER)∧is-INTG(v)) ∨                         INTG value iff INTG type
  .3        ((t=CHARACTER)∧is-TOKEN(v))                 TOKEN value iff Character type
153.4       type : SameTyp: DTyp × (BOOL|INTG|TOKEN) → BOOL
```

8.2 Semantic Invariance

The "connection" between the two related system components: the dictionary and the file system, namely that the former describes the latter, leads to an invariant predicate. It is given in figure 3.

We shall often see the need for relating, through an inv-ariant predicate, the context-sensitive information needed between otherwise context-free specified components of for example abstract trees.

8.3 Syntactic Domains

Let us model a very simple-minded concept of programming language blocks. Blocks consists of a set of variables, defined by their *v*ariable *id*entifiers, and a list of assignment statement, which consists of a lhs variable (identifier), and a rhs expression:

154. $Block \;::\; Vid\text{-}\underline{\text{set}} \times Asgn^{+}$
155. $Asgn \;::\; Vid \times Expr$

Let us (before we turn to a technique whereby we solve the expression of the constraint problem) further compound the last example. Assume that expressions are either just variables or infix expressions:

```
156.   Expr   = Var | Infix
157.   Var    :: Vid
158.   Infix  :: Expr × (AND | MPY | ...) × Expr
```

8.4 Syntactic Well-formedness

The constraint on blocks is now that all variables of assignment statements (presently their **lhs** 's) must be defined, ie. must be those mentioned in the block. This constraint is not captured, and cannot be expressed by context free Domain equations.

The constraint on expressions (of blocks) is that they mention only defined variables. This mutual inter-dependency between the two parts of a block cannot be formulated within the technique of Domain equations.

Whenever we define a function type to take, say Domain A arguments, or yield A results, where A has some **inv-** or **is-wf-** constraint "attached" to it (ie. **is-wf-**A or **inv-**A has been defined), then we mean, not the entire Domain A, but the constrained subset Domain A'.

Thus defining a function involving A gives rise to one or more *proof obligations:* namely to show that the defined function indeed is total over A', respectively only yields A' objects.

```
159.    is-wf-Block [mk-Block(vs,al)] ≜
  .1       (∀mk-Asgn(v,e) ∈elems al)
  .2          ((v ∈vs)
159.3         ∧is-wf-Expr [e](vs))

160.    type : is-wf-Expr: Expr → (Vid-set → BOOL)
  .1    is-wf-Expr [e](vs) ≜
  .2       cases e :
  .3          (mk-Var(v)      →
  .4             v ∈vs,
  .5           mk-Infix(l,,r) →
160.6          (is-wf-Expr [l](vs) ∧ is-wf-Expr [r](vs)))
```

Lines (159.1-3) could be rephrased in terms of an **is-wf-**_Asgn_ function (which, in turn appeals to **is-wf-**_Expr_):

```
161.    is-wf-Block [mk-Block(vs,al)] ≜
161.1      (∀a ∈ elems al)(is-wf-Asgn [a](vs))

162.    type : is-wf-Asgn: Asgn → (Vid-set → BOOL)
  .1    is-wf-Asgn [mk-Asgn(v,e)](vs) ≜
  .2       ((v ∈ vs)
162.3      ∧is-wf-Expr [e](vs))

163.    is-wf-Expr [e](vs) ≜
  .1       (is-Var(e) → is-wf-Var [e](vs),
163.2      is-Infix(e) → is-wf-Infix [e](vs))

164.    is-wf-Var [mk-Var(v)](vs) ≜ (v ∈ vs)

165.    is-wf-Infix [mk-Infix(l,,r)](vs) ≜
165.1      (is-wf-Expr [l](vs) ∧is-wf-Expr [r](vs))
```

where the types of the latter three functions are:

166. $\underline{\textbf{type}}$: $\underline{\textbf{is-wf-}}$*Expr: Expr* → *(Vid*-$\underline{\textbf{set}}$ → BOOL*)*
.1 $\underline{\textbf{type}}$: $\underline{\textbf{is-wf-}}$*Var: Var* → *(Vid*-$\underline{\textbf{set}}$ → BOOL*)*
.2 $\underline{\textbf{type}}$: $\underline{\textbf{is-wf-}}$*Infix: Infix* → *(Vid*-$\underline{\textbf{set}}$ → BOOL*)*
and:
166.3 $\underline{\textbf{type}}$: $\underline{\textbf{is-wf-}}$*Block: Block* → BOOL

Formulation (161.-165.) correspond directly to our "requirement" of associating with each defined Domain name a constraint function. Formulation (159.-160.) is a short-cut expressing the same.

We observe two things: (i) the constraint functions are always *total*; and (ii) they sometimes "act" on some *context*.

8.5 Semantic Functions

The sub-section on Denotational Semantics amply illustrated some Semantic Functions.

8.6 Auxiliary Functions

The sub-section on Computational Semantics amply illustrated some Auxiliary Functions.

9 Conclusion

We have illustrate but a few of the principles, techniques, and tools characterizing denotational semantics based, model oriented, in particular VDM style specifications.

Many facets have not been shown, nor have we had the space to enunciate when to choose for example denotational over computational specifications, etc.

Instead we refer to our forthcoming books: *Software Architectures and Programming Systems Design, vols. I-II-III* for a more complete story.

Also we have totally omitted any reference to the developmental aspects: transformation and reification of functions and operations, respectively data (structures) from abstract specifications towards more concrete realisations.

Devcelopmental facets are covered in *vols. IV-V-VI* of the above referenced book.

References

[1] D.Bjørner & C.B.Jones (eds.): *The Vienna Development Method: The Meta Language* Springer Verlag, LNCS61, 1978.

[2] C.B.Jones: *Software Development: A Rigorous Approach* Prentice Hall International, 1980.

[3] *Towards a Formal Description of Ada* ed. D.Bjørner & Ole N. Oest, Springer Verlag, LNCS98, 1980.

[4] D.Bjørner & C.B. Jones: *Formal Specification & Software Development* Prentice Hall International, 1982.

[5] C.B.Jones: *Systematic Development using VDM* Prentice Hall International, 1986

[6] *VDM'87: A Formal Method at Work* Proceedings (eds. D.Bjørner et al.), VDM Europe Symposium, Brussels, Springer Verlag, LNCS252, 1987.

[7] *VDM'88: VDM — The Way Ahead* Proceedings (eds. R.Bloomfield et al.), VDM Europe Symposium, Dublin, Springer Verlag, LNCS328, 1988.

[8] D.Bjørner: *Software Architectures and Programming Systems Design, vol.I: Foundations* To be published 1989.

[9] D.Bjørner: *Software Architectures and Programming Systems Design, vol.II: Basic Abstraction Principles* To be published 1989.

[10] D.Bjørner: *Software Architectures and Programming Systems Design, vol.III: Formal Description of Program and Programming* Concepts To be published 1989.