

Scientific Decisions which Characterize VDM

Cliff B Jones

Department of Computer Science
Manchester University
M13 9PL, UK
e-mail: cliffjones@acm.org

Dedicated to the memory of Heinz-Peter Chladek

Abstract. The formal description and development method known as VDM has been used extensively, its specification language is now an ISO standard, and it has influenced other specification languages. The origins of VDM are normally placed in language description or semantics but it is probably best known in the wider arena of formal methods for (general) program specification and design. This paper sets out a personal view of some of the key technical decisions which characterize the Vienna Development Method.

VDM is generally believed to stand for *Vienna Development Method*. The programming language description aspects of VDM were forged in the heat of a compiler development project in the IBM Laboratory in Vienna between 1973 and 1976; the technical decisions which characterize this work are outlined in Section 2. VDM is also a general formal method in that it can be applied to programs or systems other than compilers; scientific decisions relating to this more general area are discussed in Section 3. Preceding these substantive sections, the scene is set in Section 1. Some conclusions are offered in Section 4.

1 Background

It must be a relatively small proportion of scientific developments whose progress is defined by large discontinuities. In my opinion, progress in research on ‘formal methods’ has built steadily since the 1960s and the identification of new ideas has nearly always benefited from earlier work and has rarely forced a complete revolution in thinking. It is therefore worth reviewing the background pre-VDM.¹

During the 1960s, there was a growing awareness that programming languages were complex enough that their semantics needed to be described formally. This proved to be a less straightforward task than had been devising BNF to fix the context-free syntax of programming languages. A seminal conference was organised by Heinz Zemanek in

¹ The papers identified here as important not only represent a personal choice but their selection has also been influenced by accessibility.

Baden-bei-Wien in September 1964 (the proceedings [73] contain transcripts of some interesting discussions). One of the clearest semantic approaches proposed was to write a function which, for a given program and starting state, computes the corresponding final state² (see for example [61]³). Of course, this is what an *interpreter* does. An *abstract interpreter* was made (shorter and) more perspicuous by defining an abstract form of both the program and the state. Devising an abstract syntax for a language basically required choosing an abstraction of parse trees (but even here there are some detailed issues on which further comments are made in Section 2.1). Defining states abstractly involved finding convenient mathematical abstractions such as using a set of things in preference to a list if the semantics did not depend on the order.⁴

In the 1960s, IBM was developing (see [77, Part XIII]) a new programming language which became known as ‘PL/I’. In conjunction with members of the UK Hursley laboratory, the IBM Vienna Laboratory set out to provide a complete formal specification of the semantics of PL/I. This was a heroic effort resulting in many hundreds of pages of formulae (and special software for printing them). They called the work ‘Uniform Language Description III’ but JAN Lee coined the term *Vienna Definition Language* and it is ‘VDL’ rather than ‘ULD-III’ which stuck. The most readable description of this work is [57].

Technically, the definition of PL/I was an abstract interpreter whose intent was similar to [62].⁵ But the ‘richness’ of the PL/I language posed enormous challenges to the ingenuity of the definers. For example, PL/I programs could change their locus of execution both by goto statements and by an exception mechanism. Furthermore, in the original version of the language, a ‘tasking’ concept permitted parallel execution of portions of program which shared a common state. The basic abstract interpreter idea was extended and these concepts were modelled by keeping a whole control tree in the ‘state’. The idea worked in that the abstract interpreter could arrive at any of the intended final states but –as we shall see below– it made it difficult to reason about the definition.

There was in fact another impediment to reasoning about the VDL abstract interpreters. The concept of passing parameters by reference had been modelled by the (now universal) wheeze of splitting a mapping from names (program identifiers) to values into an *environment* which maps names to locations and a *store* which maps locations to values. *Locations* were an abstract surrogate for the machine addresses which an implementation might use. So far, so good. Unfortunately it was found necessary to retain the whole stack of active environments in the state.

² In its most obvious form, this was clearly limited to deterministic programs.

³ Note use of the word ‘description’ rather than ‘specification’ in McCarthy’s title; Mosses has also made the point that whether or not something is a specification is a question for standards organisations; in nearly all cases, formalists are producing descriptions.

⁴ This might sound trivial but, when the ECMA/ANSI committee used a method which revolved around an abstract machine, they eschewed the use of difficult abstractions like sets; the consequence was that if a state component was a list, the implementors had to search the entire definition to ascertain if any essential use was made of the order before they could decide on the design of their run-time state.

⁵ The Vienna group always insisted on also remembering the stimulus from that wonderful gentleman C C Elgot.

Although the VDL definitions were operational in the sense that they computed a (set of) final outcome(s) for a given pair of program and initial state, they were certainly not *Structured Operational Semantic* descriptions. This negative statement goes beyond the contrast with the neat inference rule style of [70]; it is more aimed at the way in which complex machinery had crept into the state.

It must however be remembered that we are talking about the mid-60s, that PL/I was by far the most complex language about at that time, that other languages which had had their semantics formalised were trivial by comparison, and –most tellingly of all– this was the first real formal description of a concurrent language. It must also be said that the process of writing the descriptions did something to clean up the language.⁶

It is also interesting to note that the basic abstract machine idea was embroidered with axiomatic extensions. For example, PL/I left a compiler freedom as to how compound variables such as structures were mapped onto store. The Vienna group used an implicit or axiomatic characterization of their storage model to capture this. The initial work was done in the mid-60s but is most easily read about in [10] which tackles a more general storage model.

Personally, I came into this story in 1968 when I persuaded IBM to send me on assignment to Vienna. I'd been working on testing techniques for the first PL/I product and had become convinced that *post-facto* testing of such complex software was never going to yield a reliable product and I wanted to see how formal descriptions could be used in a reasoned design process.

Vienna changed my life in many ways and I owe a huge debt of gratitude to the wonderful group that I joined in 1968. Peter Lucas was already working on compiler correctness proofs and his 'twin machine' idea [56] –applied to a non-trivial proof of the equivalence of ways of referring to variables– was a great starting point. We went on to give correctness proofs of standard compiler algorithms [29, 53]; to detect bugs in IBM products [28, 31]; and to devise new implementations based on the models of languages and target machines (e.g. [32]).⁷ Another important contribution by Lucas from this phase was his insistence that we try to separate *language concepts*.

In spite of the successes, we were paying a high price for the excessive machinery in the VDL-style of definition. One of the most troublesome lemmas in [53] stated that the interpretation of a single statement from the source language left the environment component of the state unchanged. The proof was non-trivial because the statement could be a block or a procedure call either of which did in fact install new environments; one had to show that they were popped properly.

So by the end of my first sojourn in Vienna, we had a clear notion of what was making it difficult to use operational definitions as a basis for compiler design. I had also proposed an exit idea which was to be the seed from which the exit combinator of VDM was to grow (here again, the early documentation was only in a report [30]). Essentially, the idea –in VDL terms– was to obviate the need for the entire program

⁶ One amusing example is the reluctant acceptance of what the Hursley implementors referred to as 'Bekič recursion' for proper static scoping rules.

⁷ It is unfortunate that much of the material was published only in IBM reports. The most accessible contemporary source was [53]: a testing collaboration which surprisingly led to friendship rather than (my) murder. That paper also contains further references.

being in the state by having a way of testing for abnormal return and taking appropriate action. The state of the abstract interpreter was smaller but, more importantly, what could *not* happen during interpretation was clearer.

Another interesting example of saying more by saying less came from proofs about abstract objects. Although this was to have more impact on the non-language research, the first application related to compiling algorithms. Lucas's 'twin machine' [56] approach tackled the job of proving the equivalence of two algorithms which employed different data structures. His approach was to define a new abstract interpreter which 'twinned' the data structures and prove that a predicate (which we came later to call a *datatype invariant*) remained true. Having shown that a certain relationship was preserved between the two data structures, one could then prove by induction over the computation that two functions always gave the same result. The final step was to 'erase' the unwanted components. I realised that –at least in this example– one of the data structures was more abstract than the other and that it could be recovered from the more detailed one. The proof in [38] called the function m (I did not use the term *homomorphism*) but the approach is similar to that in [65, 36].

There were also other seeds waiting to bear fruit. The Baden conference had been the genesis of IFIP's Working Group 2.2 and there had been an exciting meeting in Vienna in April 1969.⁸ Following on from that meeting, Dana Scott came back to the Vienna Lab. for a stimulating visit in August 1969. We had actually invited him to help us make sense of Floyd's [22] (about which I still had questions after visiting CMU with Ted Codd as my IBM minder!). In fact, Scott gave us a first glimpse of the then new *Denotational Semantics* ideas in [18]. Hans Bekič went to work with Peter Landin at QMC from November 1968–November 1969. Landin's role in the development of denotational semantics is often under-estimated but anyone who has read [54, 55] will have no doubt as to his seminal contributions.⁹

When I returned to IBM Hursley in late 1970, it was to manage my own 'Ad Tech' group which meant that I could send others crazy with my ideas. In the following three years, we initiated much of what was to comprise the non-language parts of VDM.

The first thing I wanted to do was to test out a 'small state' approach to semantics. In particular, because of the difficulties with [53], I wanted to show that while the state was an argument and result of most semantic functions, the environment only need to be an argument since it could not be modified. Furthermore, the exit idea gave us a way of defining what it meant to execute a goto without the heavy mechanism used in VDL. After some intense efforts to convince my new colleagues, we set about employing these ideas on Algol 60 and wrote [3]. I like to call this a *functional semantics* but it is actually close in spirit to Plotkin's *Structured Operational Semantics* [70] without the clear advantage of his rule notation. Furthermore, we had ducked the problem of concurrency. This work was to come to fruition in VDM.

⁸ Among other things, this was one of Tony Hoare's first attempts to describe an *axiomatic method*. Even I must have said something about post-conditions because I remember Belnap joking one lunch time that I should just pay at the beginning if I was only interested in the post-condition.

⁹ There was a fascinating workshop at Essex University in July 1971; John Laski was the organiser and most of the key members of the Vienna group came over; Landin was a key discussant.

Let us turn now to the efforts on developing ‘ordinary’ (non-compiler) programs. We needed for other reasons a table-driven parser and my colleagues had produced and debugged a nice implementation of Earley’s recogniser which they challenged me to prove correct. I tried and failed! I was mirroring Tony Hoare’s experience in his proof of FIND [35] in that the step from specification to design was too big to bridge in one stage; we both needed marry the advantages of formalism with a stepwise design approach. I finally constructed my own reasoned development of Earley’s recogniser in 1971 and it was published externally as [39]. Data refinement was crucial to this formal development. Interestingly, this paper did not employ the homomorphic mapping of both later and earlier VDM work but used a careful analysis of the properties required of abstract datatypes and a subsequent check that these properties were preserved in the representations. Having developed my program (which my colleagues insisted on being the first to test) a comparison with their code uncovered bugs in the latter.

VDM’s separation of pre-conditions from post-conditions as well as the use of post-conditions of two states date from [40]. In fact, the struggle with avoiding angelic composition is graphically clear in [40, Section 3.5].

So, by the time the ‘VDM group’ formed in 1973¹⁰ there was a lot from which to work.

2 Denotational Semantics

I vividly remember the call from Peter Lucas in late 1972 asking me if I’d like to come back to Vienna and try out formal methods on a full blown PL/I compiler for a completely new machine architecture. I said ‘yes’ with an alacrity which ignored the fact that my son Peter was due to arrive in the world. We also plotted who else we could get and agreed that I should call Dines Bjorner¹¹ who proved equally reluctant to join the team. I moved to Vienna in March 1973 on a ‘permanent’ transfer.

Even before the group fully assembled, we exchanged style proposals and there was a rapid and almost unanimous vote to write a formal semantics for (ECMA/ANSI) PL/I in the denotational style, a naive assumption that the IBM machine architecture would be more stable than the output from the standards committee, and an equally optimistic assumption that the formal definition of the machine architecture promised from Poughkeepsie would be adequate for our object-time design.

The remainder of this section discusses some of the technically interesting decisions which were made in the two years or so in which we were trying to design a compiler, refine our methods, and correct some of the wilder peculiarities of the object machine architecture.

2.1 Superficial points

The PL/I definition [8] is an interesting document: the base definition (Part I) is a fraction of the size of the VDL definition¹² and is matched by a Part II which contains a

¹⁰ Most of the Vienna group were diverted onto looking for hidden parallelism in FORTRAN programs whilst I was enjoying myself in the UK.

¹¹ Dines and I can still never agree where we first met!

¹² The ECMA/ANSI decision to drop PL/I’s tasking was a relief and clearly helps the comparison.

commentary of about the same size. Anyone wishing to see the style, might find it easier to tackle the ALGOL 60 description in [33] or that for Pascal in [4].

The obvious comparison is with the ‘Oxford style’ of writing denotational descriptions (see [67, 64, 75, 23]. One immediate –but superficial– difference is the use in the Vienna descriptions of longer names for functions and their parameters. I believe this comes from tackling larger specificands and reflects a tendency which is familiar to all who have written large pieces of software.

The Vienna group experimented with different document structures for presenting descriptions: [8] starts with the whole (abstract) syntax, followed by the so-called context conditions, then the state or auxiliary objects and finally all of the semantic functions. In [33], we tried to group the material (syntax, context conditions, semantic functions) by language construct. Similarly, [8] separated all of the commentary into a separate volume whereas Bjorner and his colleagues pioneered a style of numbering lines of formulae and keying comments to those line numbers after each substantial chunk of formalism (see for example their landmark description of Ada [14]).

Even with the abstract syntax, Vienna managed to depart from orthodoxy. As mentioned above, an abstract syntax defines a set of trees which contain the essential information in a program with none of the syntactic clues which make it possible for a parser to get at the information. This leaves the issue of how one distinguishes sub-trees with similar content. The ‘standard’ approach is to put the load on a disjoint union operator as in

$$Stmt = Assign + Goto + Call + \dots$$

We chose to use a simple set union and have a form of definition which forced a constructor:

$$Stmt = Assign \cup Goto \cup Call \cup \dots$$

$$\begin{aligned} Assign &:: lhs : Var \\ &rhs : Expr \end{aligned}$$

Such ‘composite object’ definitions give rise to a convenient set of constructor and selector functions:

$$\begin{aligned} mk\text{-}Assign &: Var \times Expr \rightarrow Assign \\ s\text{-}lhs &: Assign \rightarrow Var \\ s\text{-}rhs &: Assign \rightarrow Expr \end{aligned}$$

which can be used in defining, *unter anderen*, semantic functions by cases as in:

$$\begin{aligned} M[[mk\text{-}assign(l,r)]] &= \dots \\ M[[mk\text{-}Goto(id)]] &= \dots \end{aligned}$$

The Vienna style owes much to McCarthy’s ideas on abstract syntax: in [61] he gave an axiomatic characterization of constructors and selectors.

Another minor observation relates to what we originally called *context conditions*. While the sort of syntax described above succeeds in abstracting from the trivia of concrete syntax, it is still *context free*: just as in Floyd’s [21], there is no way –for example– of requiring that declarations and use of variables match. There is, however, a significant advantage in defining that subset of *Program* which the meaning function

is expected to map to a denotation: the semantics is not clouded with the definition of context dependencies.¹³ In the definitions written in Vienna and shortly thereafter, it was normal to define the context conditions as a recursive function over (a static environment and) the syntax tree:

$$is-wf\llbracket mk-Assign(l,r) \rrbracket \rho = \dots$$

An issue which was by no means minor or superficial was concurrency but I do not intend to go into it in this paper. As explained, the languages we described turned out to be sequential. Hans Bekič did propose a concurrent combinator but a full discussion of his work would require a comparison of the posthumous collection of his writings [7] with ideas like Plotkin’s *Power Domains* [69] and that is well beyond the intended scope of this review.

2.2 Use of combinators

Why is denotational semantics viewed as purer than operational semantics? Why is it better to define:

$$M: Program \rightarrow (\Sigma \rightarrow \Sigma)$$

than

$$I: Program \times \Sigma \rightarrow \Sigma$$

I suppose that the respectable answer uses the argument of *full abstraction*; my view is less absolute! Recall that we were writing a language description as the basis for a compiler design. We certainly did not want to have mechanisms in the definition (state) which would not be present in our implementation: we hoped that the abstractions in the definition would have one or more representations in our object-time store but that the implementation would never conflate two abstract states. It was actually unlikely that having the state-to-state function as the denotation for a sequence of statements would simplify the problem of justifying any cross-statement optimizations.¹⁴ What was crucial was the adoption of a ‘small state’ semantics (in contrast to VDL’s ‘grand state’): we needed a minimum of information in the state objects which were in the range of the (functional) denotations. With

$$Env = Id \rightarrow Loc$$

$$\Sigma = Loc \rightarrow Val$$

It seems to me, that there is an *essential* distinction between

$$M: Program \rightarrow (Env \rightarrow (\Sigma \rightarrow \Sigma))$$

¹³ The terms *static semantics* and *dynamic semantics* have been used for what is here referred to as ‘context conditions’ and ‘semantics’.

¹⁴ Optimizations are an interesting issue in formal design of compilers – see [45].

and

$$I: Program \times State \rightarrow State$$

$$State = Env \times \Sigma$$

because one needs to be able to see immediately that environment-like information could not be changed by executing statements. Furthermore, we certainly had to eschew any tricks like putting the program text in Σ .

One could in fact ask whether we needed a denotational definition at all. I believe that the answer was ‘yes, to keep us honest’ rather than the importance of having fully abstract procedure denotations. Had Plotkin’s nice SOS notation in [70] been known to us (rather than the heavier notation in [3]), I think we would have tried it although there would have still been a need for more notational invention to prevent the reduction rules becoming obscured by the need to have several auxiliary parameters for a PL/I description.

Well, that’s probably enough heresy for now.

Less controversially, we chose to use combinators to represent the compositions of denotations. Thus with

$$M: Stmt \rightarrow Env \rightarrow (\Sigma \rightarrow \Sigma)$$

$$M[[s1;s2]]\rho = m[[s2]]\rho \circ M[[s1]]\rho$$

was written as:

$$M[[s1;s2]]\rho = M[[s1]]\rho; m[[s2]]\rho$$

Where the second semicolon was defined (for the moment) in the obvious way. We pushed the combinator idea further and half a dozen such combinators are carefully defined in [12]. I have no doubt that they made our definitions easier to grasp.

2.3 The exit idea

The largest single distinction between the Oxford and Vienna styles came about with the description of statements which caused exceptional changes to the flow of control. For ease, we’ll focus here on goto statements but modelling PL/I’s exception mechanism (On statements etc.) presented the same challenge. This problem had been seen in the early in the work on ‘mathematical semantics’ and the idea of *continuations* was invented more-or-less independently by Mazurkiewicz⁷¹ [60], Lockwood Morris [66]¹⁵ and Chris Wadsworth [76]¹⁶.

Basically, by defining

$$Cont = \Sigma \rightarrow \Sigma$$

¹⁵ Morris’ title plays on Landin’s [55].

¹⁶ Peter Lucas has recently informed me that he first heard about continuations from van Wijngaarden.

the semantics of statements could be defined

$$M_c: \text{Statement} \rightarrow \text{Env} \rightarrow (\text{Cont} \rightarrow \text{Cont})$$

with the trick that

$$M_c[[s_1; s_2]]\rho\theta = M_c[[s_1]]\rho(M_c[[s_2]]\rho\theta)$$

$$M_c[[mk-Goto(id)]]\rho\theta = \rho(id)$$

We were aware of continuations in 1973 (probably via Bekič but I had attended some Oxford lectures by Christopher Strachey during my 1970–72 stay in Hursley and Lockwood Morris had also spent some time with me in Hursley). But I pushed hard to use a development of the exit idea from my first stay in Vienna. I remember some tough debates and I doubt that we would have used exits if the evidence of [3] had not been available: fortunately it was clear that exits could handle abnormal termination. Again to oversimplify somewhat, the idea was to use

$$M: \text{Statement} \rightarrow (\Sigma \rightarrow \Sigma \times \text{Abn})$$

and then test where appropriate for a non-nil abnormal return value (*nil* marked normal return, non-nil values gave some indication of what was to be done next). The notation that I had used in Hursley was ugly because it made every test explicit but the combinator idea came to our rescue and gave a surprising bonus. We defined a combinator which performed exactly the right combination of the denotations of *s1* and *s2* to get the denotation of *s1; s2* where either might have terminated abnormally.

What was this combinator? We redefined the (semantic) semicolon operator

$$f; g = (\lambda(\sigma, a) \cdot \text{if } a = \text{nil} \text{ then } g(\sigma) \text{ else } (\sigma, a))^\circ f$$

This is more than a symbol saving exercise. Prompted by Lucas' insistence to think about language concepts, we had all recognised that it was unreasonable to have to completely redefine a language when new concepts were added: for example, why did the semantics of expressions and assignment statements have to be defined differently depending whether the language had goto statements or not? The redefinition of the combinator seemed a more apposite way to reflect the distinction.¹⁷

The difference between continuations and exits excited considerable debate. No one who was at the delightful Winter School organised by Bjørner at Lyngby in 1979 (see [11] for proceedings), will forget Joe Stoy's *Deus es machina* drawing which expressed his distaste at the alleged operational exit; even in the paper in the proceedings, Joe catches his writing 'taking on a doctrinal flavour'. I could never understand this argument. I can see clearly that continuations are powerful; they can model co-routines which exits can't. But power is not necessarily a virtue: putting the whole program text in the VDL states was powerful in some sense. The limitation of exits appeared to me to be an advantage: where the weaker idea was adequate, employing it might say more.

¹⁷ Peter Mosses has commented verbally that our approach influenced his early thinking on what was to become *Action Semantics* [68].

Of course, there was a legitimate mathematical question about the relationship of these two approaches. In general, they could not be equivalent but definitions of what were intended to be the same language using the two styles ought correspond in some way. I took on the challenge of proving that *exit* and *continuation* definitions of a representative language correspond in [43] and [13, Chapter 5]. As one might guess from the multiple attempts, the proof was not simple and the process of teasing it into manageable steps is enlightening. For example, it is easily overlooked that one consequence of a continuation style definition is the need to put things in the environment which are not required for exit-style definitions.

2.4 Use in proofs

As has been made clear, the Vienna work was not just an experiment in definition style; nor was the definition of a large language an end in itself: our goal was to use the PL/I definition [8] as a basis for the design of a compiler. While it lasted, the project was successful. We wrote the definition of the source language; we also wrote definitions of the key parts of the novel object machine architecture; a formal mapping from our object-time state back to the state of the PL/I description was not only written but uncovered errors in our design; we used a similar *retrieve function* to record the relationship between the concrete and abstract syntaxes (thus creating a specification of the front end).

Unfortunately, in February 1975, IBM decided to cancel the machine architecture and our project no longer had a target machine. I'll make some comments about the diaspora which followed below but before it occurred, the essential ideas were written up in [9, 41]. These documents certainly suffer from the confusion surrounding the redirection of the Vienna Laboratory but they did serve as some sort of contemporary record. Fortunately, those who moved on spent time producing more polished examples of the key ideas which originated in those three exciting years.

2.5 Use and Dissemination of VDM for Languages

The main books on the denotational semantics work from Vienna were [12, 13].

Dines Bjørner moved back to (what he just about remembered was) his home country of Denmark. His team at the university and later the colleagues he advised at the Danish Datamatik Center pushed the ideas on language work forward. A major achievement was the creation of the first European Ada compiler to receive official validation; the definition was first published as [14].¹⁸

3 Specification and formal development of programs

Although compilers play a critical role in the creation of software, they comprise a small percentage of the total population of programs. A development method which

¹⁸ Subsequent research together with Egidio Astesiano's team led to SMoLCS. Comments on the RAISE work are given in Section 4 as are references to their work on database descriptions.

addressed only language description and processors would not receive wide circulation. Concepts like data abstraction and justified development steps permeate all applications of VDM but there are many technical commitments specific to the design of programs outside the compiler arena which deserve comment. This section discusses some of the decisions which relate to the specification and justified development of such software. Judging –for example– by the number of books which have been published, this aspect of VDM is now considerably more widely known than its specific application to formal semantics descriptions.

One pervasive idea is *compositional development*. This is not as easy to define as one might think at first sight. Assume that a (specified) component C is to be created and that its designer decides that a series of sub-components, say sc_i are needed. The essence of a compositional method is that the specifications of the sc_i should say all that it is necessary to insulate the development of the sc_i from the design of C . This means that the designer of C can verify its design without knowing the implementations of the sc_i and that the developers of the sc_i should not have their work rejected on any grounds other than those recorded in their specifications. In clear contrast is a development method which requires some extra proof obligation to be discharged once the development of all of the sc_i are available. There are however ways of meeting the above formal requirement without meeting its spirit. These issues become more delicate with concurrency so we'll leave them for now.

3.1 Why models?

A computer system contains a model of something in the physical world.¹⁹ For example, a database application might record information about students and courses or a process control system might maintain information about the required and expected values of measurement devices. Such a system provides a number of functions or operations which can be accessed via some interface. Such operations might change the state of the model inside the computer or simply access some (derived) values from that model. To describe or specify such a system is to pin down precisely the observable effects of these operations. There are two ways of tackling this. Operations can either be described solely in terms of each other by relating their input and output values or the operations can be described individually with respect to (an abstraction of) a state. Let's agree to use the terms *property-oriented* and *model-oriented* descriptions respectively for these two approaches.

The property-oriented approach has the potential advantage that a system might be described solely in terms of its visible behaviour. The archetypical example of a successful property-oriented specification is a *stack*: one can say that the last value pushed is what is returned by a pop and that pop/push pair leave the stack unchanged. Although not widely circulated, the first example of such a specification that I saw was from Lucas and Walk in their 'Documentation of Programming Ideas' which was presented to patent experts in 1969: they specified a stack in exactly this way²⁰. The

¹⁹ For the purposes of the current discussion, we can ignore the extent to which the computer system becomes a significant part of the real world system which –we are here arguing– is abstracted by the computer model.

²⁰ This was published a couple of years later as [58].

more influential references on this approach were by Guttag, Horning and Zilles around 1975 and this led to more than a decade of intense research on Algebraic specification methods (an up-to-date overview of this work with many references can be found in [5]).

The potential advantage of property-oriented descriptions is, however, sometimes unrealisable. There are formal difficulties like coping with non-deterministic and partial operations. It can also be shown that there are systems for which no finite set of axioms (without hidden functions) can capture their behaviour.

But it was pragmatic arguments which led VDM to adopt the model-oriented approach to the description of data types and systems. With an awareness of the advantages of implicit storage models (cf. Section 1) and of the basic idea of describing data types by properties, it appeared more realistic to choose an appropriate abstract model of the underlying state of a system and to define each operation individually by pre and post-conditions.

In the 1970s this was a controversial decision although Z, [72] and later Abrial's Abstract Machine Notation in B, [1] were in the same camp. It is probably fair to say today that it is acknowledged that there are roles for both property and model-oriented specifications and that most people would concede that reasonably complex systems are nearly always best described via models.²¹

The decision to focus on model-oriented specifications led to some acrimonious debates and even some scientific understanding. Let's stick to the later heading!

I must have felt under some pressure from the algebraic school which was fairly well represented in IFIP's *Programming Methodology* Working Group (WG 2.3). The question of how one could be sure that a model was not redundant had to be addressed and I remember thinking up the notion of *implementation bias* at the Niagra WG 2.3 meeting in August 1977.

Given that the model for a system can be chosen by the specifier, there is no limit to the possible choice of state. For example, one could model a stack by the sequence of its active elements in either order. Of course, these two models are isomorphic. But one could also choose a model which kept every element which had ever been pushed onto the stack and just flag the popped ones as unreachable. This might appear perverse but such a model could still describe the same external behaviour and the question is how to detect more subtle cases of redundancy.

It is important to understand the negative effects of such redundancy. The reason that I initially chose the phrase 'implementation bias' is that this is precisely what it could do: redundancy could make it difficult to spot certain valid implementation data structures. As explained below (Section 3.4), it is desirable to relate implementation and specification objects by a mapping from the latter to the former; this becomes impossible if two distinct specification states are 'represented' by the same implementation object. The key idea first described in [42] was that freedom from implementation bias could be shown if equality over the underlying states could be decided in terms of its operations. This is explained at greater length in [48, Section 9.1].

²¹ When it was decided in 1991 to widen both VDM-Europe and the sequence of Symposia, it was not quite clear what to encompass: there was serious discussion that an appropriate broadening would be model-oriented formal methods. Fortunately this was never enshrined in the statutes.

3.2 Form of pre/post-conditions

A specification for a single VDM operation might look like:

```
POP e:El
ext wr s : El*
pre q ≠ []
post  $\overleftarrow{q} = [e] \frown q$ 
```

It is worth taking a look at some of the decisions embodied in such descriptions.

Firstly, note that the pre and post-conditions are separate (in contrast to say Z where they are written as conjuncts of a single predicate – see citeZVDM93 for a fuller discussion).

The meaning intended for the pre-condition is that for any initial state (and input values) which satisfies the pre-condition, the operation must terminate and leave the final state (and any results) such that the post-condition is satisfied. Thus the pre-condition is not a firing condition but rather something that the designer must ensure –preferably by proof– is satisfied for any potential use of the operation. An implementation is not constrained to do anything at all in situations where the pre-condition is not satisfied.²² But unlike the pre-conditions in Hoare’s original axiomatic method [34], so-called ‘total correctness’ is required in VDM.

Notice also that the post-condition relates the final to the initial state²³ It was obvious to me from the beginning (cf. [40]) that this was the best way to document the intention of a state-transforming operation. The use of special variables in, for example, the *weakest pre-condition* system [19] struck me as arbitrary and error prone. I’m sure that the proof rules required for the VDM style of post-condition struck others as inelegant (at best) – but that brings us to the next section.

Before moving on it is worth mentioning the notion of *satisfiability*. The description earlier in this section of the responsibilities of an implementor can only be fulfilled if:

$$\forall \overleftarrow{\sigma} \in \Sigma \cdot \text{pre-OP}(\overleftarrow{\sigma}) \Rightarrow \exists \sigma \in \Sigma \cdot \text{post-OP}(\overleftarrow{\sigma}, \sigma)$$

This proof obligation exists for all VDM specifications but it is in nearly all cases used only as a mental check list item.

3.3 The proof rule saga

There is, of course, the question of the proof rules which are to be used in showing that composite statements satisfy their specifications (in the case of a non-trivial decomposition, under the assumption that the specifications of the sub-components are met). It is clear that Hoare’s system –as originally set out in [34]– benefits from the fact that

²² There are some auxiliary questions about what the implementor can assume on the types of values.

²³ We will pass over the wisdom or otherwise of the back hook notation (\overleftarrow{x}) for the initial state.

pre and post-conditions are both predicates of single states: the uniformity facilitates concise rules such as:

$$\boxed{\text{While-H}} \frac{\{P \wedge b\} S \{P\}}{\{P\} \text{ while } b \text{ do } S \text{ end } \{P \wedge \neg b\}}$$

My insistence on using post-conditions which were predicates of two states (initial and final) made it impossible to achieve this brevity. Furthermore, the rules in [46] were unnecessarily heavy.²⁴ It took the intervention of Peter Aczel [2] to show that rules could be formulated for the relational case that were almost as brief as in Hoare's system

$$\boxed{\text{While-A}} \frac{\{P \wedge b\} S \{P \wedge R\}}{\{P\} \text{ while } b \text{ do } S \text{ end } \{R \wedge \neg b\}} \quad R \text{ twf}$$

These rules actually have an advantage in that the well-foundedness of R automatically ensures termination. (In some ways this is a neat generalisation of the 'variant function' of weakest pre-condition systems.)

Soundness proofs of the rules used in [48] and subsequently are given—in terms of relations—in [49].

3.4 Data reification

Since the main points of the story of the data reification work are related in [50], this section can be brief. The transition from Lucas' 'twin machine' approach to the retrieve function idea is first recorded in Vienna in 1970; after the diversion of the more axiomatic approach used in the proof of Earley's recogniser, the idea of a 'retrieve function' (a homomorphism from the representation to the abstraction) became a key part of VDM's notion of *data reification*. Coupled with this was the important concept of *adequacy*.

The VDM books were I believe the first to give due importance to this aspect of program design. The idea of engineering handbooks for software was again the motivation behind the proposal to build up *theories* of data types [44].

3.5 LPF

Here again, much has been written about the work on logics which handle partial functions (or undefined terms). Although this is now a distinctive part of VDM, the reader is referred to [6, 15, 52] for further discussion of the *Logic of Partial Functions* (LPF).

3.6 Concurrency

A review paper is planned of the concurrency work in the near future. Suffice it here to say that the original rely/guarantee idea of [47] was progressed significantly in [74, 16, 17] and that subsequent work on the difficult issue of achieving compositionality in the presence of interference has gone in the direction of concurrent object-based languages [51, 37].

²⁴ Although they did permit discussion of some interesting distinctions between 'up' and 'down' loops.

4 Conclusions

After what the trade press called the 1975 ‘St Valentine’s day massacre’, many of those who had been involved in the PL/I compiler project in IBM Vienna –and thus the real solidifying of VDM– dispersed. At one point, the acronym ‘VDM’ was claimed to have stood for ‘Vienna, Denmark and Manchester’; at another ‘Very Diverse Methods’; my preference was for ‘Vorsprung durch Mathematik’. The real origin of the name owes more to Dines’s cocktails than anything else (or was that only the even worse ‘Meta-IV’?). There was some concern that ‘VDM’ was too close to ‘VDL’ and this did indeed confuse the unwary.

Be that as it may, at least the description part of VDM has made it to an ISO standard and this is one reason for now looking back on the venture.

I confess, that I personally preferred to see the whole exercise as more like the foundation of a ‘school’ than a single specific notation which was to be carved on tablets of stone. I had some doubts about the motives of some of the early standardisation effort and only got fully involved when I saw that (a) it was going ahead anyway and (b) some rather odd decisions were being made. The development of LPF is one example of something that was by no means fixed early in VDM’s history; concurrency is an issue which still requires more work.

One pleasing outcome of the the work has been its influence on other specification notations. It is worth mentioning at least VVSL [63], Irish VDM [59] Larch [26], RAISE [24, 25] and B [1]. The significant omission from this list is –of course– Z [72]. Jean-Raymond Abrial and I arrived at Oxford’s ‘Programming Research Group’ at almost the same time in 1979. We worked well together in a mixture of notations and I can’t help suspecting that VDM and Z could have become closer had Jean-Raymond have retained control of Z. Some evidence for this suspicion is that his Abstract Machine Notation in B [1] does fit more closely with aspects of VDM than Z.

It is fair to address the question of what –with the benefit of hindsight– one would have wished had been done differently.

I should have liked to have seen less emphasis on notation and more on concepts (for example, [27] discusses some of the more arbitrary divergences between Z and VDM). The RAISE effort was probably mistimed in the sense that cleaning up VDM first might have had the twin benefit of giving RAISE a better starting point and postponing the start on the concurrency aspects of RAISE (which must rate as its most difficult to use).

In spite of a number of efforts, projects to provide tool support have never been easy to justify.

Perhaps what remains –and might be seen as significant in years to come– is that there are many useful descriptions (and some specifications) around. Not all of these descriptions are public and I applaud Brian Monahan’s recent proposal to encourage a ‘open source’ approach to descriptions of widely used systems. In my own teaching, I moved to trying to communicate the idea of *abstract modelling* as a way of understanding computer systems.

Above all –for me at least– it was a great way to work on some real problems and with some wonderful colleagues. It is to all of them that I dedicate this paper.

Acknowledgements

This paper was drafted whilst I was still at Harlequin: I am grateful for the time made available for the work.

Peter Lucas was kind enough to comment on a draft of this paper for which I am very thankful.

References

1. J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.
2. P. Aczel. A note on program verification. manuscript, January 1982.
3. C. D. Allen, D. N. Chapman, and C. B. Jones. A formal definition of ALGOL 60. Technical Report 12.105, IBM Laboratory Hursley, August 1972.
4. D. Andrews and W. Henhapl. Pascal. In [13], pages 175–252. 1982.
5. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brueckner, editors. *Algebraic Foundations of System Specification*. Springer Verrlag, 1999.
6. H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
7. H. Bekič. *Programming Languages and Their Definition: Selected Papers of H. Bekič*, volume 177 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.
8. H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974.
9. H. Bekič, H. Izbicki, C. B. Jones, and F. Weissenböck. Some experiments with using a formal language definition in compiler development. Technical Report LN 25.3.107, IBM Laboratory, Vienna, December 1975.
10. H. Bekič and K. Walk. Formalization of storage properties. In [20], pages 28–61. 1971.
11. D. Bjørner, editor. *Abstract Software Specifications: 1979 Copenhagen Winter School Proceedings*, volume 86 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.
12. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.
13. D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982.
14. D. Bjørner and O. N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.
15. J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.
16. Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY*. PhD thesis, Louvain-la-Neuve, June 1994.
17. Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. Technical Report UMCS-95-10-3, Manchester University, 1995.
18. J. W. de Bakker and D. Scott. A theory of programs. Manuscript notes for IBM Seminar, Vienna, August 1969.
19. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
20. E. Engeler. *Symposium on Semantics of Algorithmic Languages*. Number 188 in *Lecture Notes in Mathematics*. Springer-Verlag, 1971.

21. R. W. Floyd. On the nonexistence of a phrase structure grammar for ALGOL 60. *Communications of the ACM*, 5:483–484, 1962.
22. R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.
23. M.J.C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.
24. The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992. ISBN 0-13-752833-7.
25. The RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, 1995. ISBN 0-13-752700-4.
26. John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. ISBN 0-387-94006-5/ISBN 3-540-94006-5.
27. I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. *ACM Software Engineering News*, 19(3):75–81, July 1994.
28. W. Henhagl. A proof of correctness for the reference mechanism to automatic variables in the F-compiler. Technical Report LN 25.3.048, IBM Laboratory Vienna, Austria, November 1968.
29. W. Henhagl and C. B. Jones. The block concept and some possible implementations, with proofs of equivalence. Technical Report 25.104, IBM Laboratory Vienna, April 1970.
30. W. Henhagl and C. B. Jones. On the interpretation of GOTO statements in the ULD. Technical Report LN 25.3.065, IBM Laboratory, Vienna, March 1970.
31. W. Henhagl and C. B. Jones. Some observations on the implementation of reference mechanisms for automatic variables. Technical Report LR 25.3.070, IBM Laboratory, Vienna, May 1970.
32. W. Henhagl and C. B. Jones. A run-time mechanism for referencing variables. *Information Processing Letters*, 1(1):14–16, 1971.
33. W. Henhagl and C. B. Jones. A formal definition of ALGOL 60 as described in the 1975 modified report. In [12], pages 305–336. 1978.
34. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
35. C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14:39–45, January 1971.
36. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
37. Steve J. Hodges and Cliff B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 1–22. Kluwer Academic Publishers, 1996.
38. C. B. Jones. A technique for showing that two functions preserve a relation between their domains. Technical Report LR 25.3.067, IBM Laboratory, Vienna, April 1970.
39. C. B. Jones. Formal development of correct algorithms: an example based on Earley’s recogniser. In *SIGPLAN Notices, Volume 7 Number 1*, pages 150–169. ACM, January 1972.
40. C. B. Jones. Formal development of programs. Technical Report 12.117, IBM Laboratory Hursley, June 1973.
41. C. B. Jones. Formal definition in compiler development. Technical Report 25.145, IBM Laboratory Vienna, February 1976.
42. C. B. Jones. Implementation bias in constructive specification of abstract objects. typescript, September 1977.

43. C. B. Jones. Denotational semantics of goto: An exit formulation and its relation to continuations. In [12], pages 278–304. 1978.
44. C. B. Jones. Constructing a theory of a data structure as an aid to program development. *Acta Informatica*, 11:119–137, 1979.
45. C. B. Jones. The Vienna Development Method: Examples of compiler development. In M. Amirchahy and D. Neel, editors, *Le Point sur la Compilation*, pages 89–114. IRIA-SEFI, 1979.
46. C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980. ISBN 0-13-821884-6.
47. C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.
48. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1986.
49. C. B. Jones. Program specification and verification in VDM. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F: Computer and Systems Sciences*, pages 149–184. Springer-Verlag, 1987.
50. C. B. Jones. Data reification. In J. A. McDermid, editor, *The Theory and Practice of Refinement*, pages 79–89. Butterworths, 1989.
51. C. B. Jones. Process algebra arguments about an object-based design notation. In A. W. Roscoe, editor, *A Classical Mind*, chapter 14, pages 231–246. Prentice-Hall, 1994.
52. C. B. Jones. TANSTAAFL with partial functions. In William Farmer, Manfred Kerber, and Michael Kohlhase, editors, *Proceedings of the Workshop on the Mechanization Of Partial Functions*, pages 53–64, 1996.
53. C. B. Jones and P. Lucas. Proving correctness of implementation techniques. In [20], pages 178–211. 1971.
54. P. J. Landin. A correspondence between ALGOL-60 and Church’s lambda-notation. Parts I and II. *Communications of the ACM*, 8:89–101, 158–165, 1965.
55. P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.
56. P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, June 1968.
57. P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6 of *Annual Review in Automatic Programming Part 3*. Pergamon Press, 1969.
58. Peter Lucas. On the semantics of programming languages and software devices. In Randall Rustin, editor, *Formal Semantics of Programming Languages*, pages 41–57, Englewood Cliffs, New Jersey, 1972. Prentice Hall. Proceedings of the Courant Computer Science Symposium 2.
59. M. Mac an Airchinnigh. Tutorial Lecture Notes on the Irish School of the VDM. In [71], pages 141–237, 1991.
60. A. W. Mazurkiewicz. Proving algorithms by tail functions. *Information and Control*, 18(3):220–226, April 1971.
61. J. McCarthy. A formal description of a subset of ALGOL. In [73], pages 1–12, 1966.
62. J. McCarthy. A basis for a mathematical theory for computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland Publishing Company, 1967.
63. Cornelius A. Middelburg. *Logic and Specification: Extending VDM-SL for advanced formal specification*. Chapman and Hall, 1993.
64. R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.

65. R. Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Computer Science Dept, Stanford University, February 1971.
66. F. L. Morris. The next 700 formal language descriptions. Manuscript, 1970.
67. P. D. Mosses. The mathematical semantics of Algol 60. Technical Monograph PRG-12, Oxford University Computing Laboratory, Programming Research Group, January 1974.
68. Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
69. G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3), 1976.
70. G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.
71. S. Prehn and W. J. Toetenel, editors. *VDM'91 – Formal Software Development Methods. Proceedings of the 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 1991, Vol.2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
72. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.
73. T. B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.
74. K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. available as UMCS-91-1-1.
75. J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
76. C. Strachey and C. P. Wadsworth. Continuations – a mathematical semantics for handling jumps. Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, January 1974.
77. R. L. Wexelblat. *History of Programming Languages*. Academic Press, 1981.