Methods Integration


Paper:

Methods Integration: Time for Reflection (and Reorientation?)

John Cooke
Methods Integration: Time for Reflection
(and Reorientation?)

John Cooke
Dept of Computer Studies, Loughborough University
Loughborough, LE11 3TU, UK

Abstract

Taking the presentations given at this workshop as a snapshot of current work on “Methods Integration” (MI), we consider current and future aspirations for the topic within the overall context of software production.

There are significant differences in what has been attempted. The intellectual gulf between the types of methods which workers seek to bridge is huge. And the rationale for seeking to link them also varies widely; some try to hide any semblance of formalism, others seek to make formal methods more accessible.

Whilst making undoubted progress in numerous local situations, we are, at the same time, fostering diversification. Unfortunately the substantial effort invested tends to reinforce the ‘not invented here’ syndrome which bodes ill for those who search for a single universal method. Once we have achieved sufficient understanding of the problems and processes involved, all should become clearer and the methods easier to use. As a small contribution in the search for simplicity we attempt rationalisation of certain key terms/procedures, and suggest how MI might be embedded within the software construction process.

1 Introduction

At face value, little has changed since the appearance in 1992 of what has become a pivotal paper by Semmens et al. [23]. As witnessed by the papers presented at this workshop [4] there are still several different agendas, the reasons for wishing to ‘integrate’ formal and informal approaches to software development are various and diverse, as are the specific different ‘methods’ which they address. In section 2 we attempt a broad classification of the presentations in terms of their objectives, and set them in the context of other recently published work (section 3).

Historically there are basically two, diverging, trains of thought; one seeks to make formal methods more accessible, the other seeks to hide them - using a formalism merely to underpin high-level, and apparently ‘informal’, methodologies. However, in both directions there is still some clouding of the key issues. This stems not only from the blurring of the formal/non-formal interface but also the lack of clear, and agreed, distinction between the requirements, the specification, and the design of both systems and software. We address the question of terminology in section 4; putting forward a possible taxonomy which is then developed in sections 5 and 6 into an overall framework in which much of the current MI work can be placed. The hope being that it will harmonise and converge - producing less diverse but more manageable and productive methods based on sound Computer Science and suitable for most practitioners. It would also be desirable if the same underlying mathematical model could be sufficiently open-ended so as to provide a basis for the development of more specialised software systems. Section 7 contains a brief summary and some concluding remarks.
The Workshop Papers

Most of the papers presented at the Leeds workshop [4] describe the use of less formal methods as a stepping stone to the construction of formal specifications which can then be used in the generation or verification of appropriate designs or implementations. Typical of what was to be expected is the paper by Walker et al. [25] which describes the use of the Yourdon structured method in the construction of specifications expressed in the VDM-SL notation. Effectively this yields an extended VDM notation; but few surprises. Similarly, Facon et al. [8] use object models in the construction of specifications expressed as an abstract machine in the B language. So here we have the same idea but different parameters.

Despite (or perhaps, because of) the fact that that schematic diagrams can be described by mathematical expressions, diagrams are often perceived as informal (and easy to understand) representations of certain aspects of software or system artefacts. As such they are used widely, and to great effect, in structured and object-oriented, O-O, methods (and in formal methods too, though this is less well appreciated or admitted) and several of the workshop papers stress this aspect. Pictures are never very far away - and this is a very desirable state of affairs. Kasurinen and Sere [14] present a way of deriving Z specifications starting with ER diagrams. This technique can be used in conjunction with structured or O-O (or other kinds of) development methods. Lano and Goldsack [17] use O-O methods to construct specifications in extended VDM (VDM++). Many kinds of picture are used here, most notably statecharts which seem to provide a possible means of communication between the rather entrenched sub-cultures which pervade the software engineering community. Here they are perhaps used more overtly and to greater effect than elsewhere. (This is the first instance where there is a culture clash - the formalists finding comfort in precision and detail when others seek to defer what they see as undesirable technicalities. Paradoxically, formalists also seek to defer unnecessary detail, in the form of design decisions, for as long as possible.)

Rawson and Allen [20] contribute to the development of a ‘method’ for constructing a Z specification starting from an Object-Oriented Analysis model. This paper is not so much about MI but about specification construction, but is still a most useful contribution. Likewise, the paper by France et al. [9] is not about integration per se but gives us two case studies on the construction of “rigorous requirements specification” (there is some divergence of terminology here which will be addressed in section 4), both using diagrams as a starting point. The first uses Petri Nets to prototype an SA/RT model. The second is basically the Fusion OOA method using Z “mini-specs” – but you need both the diagrams and Z in order to deduce the full behaviour. This point serves to emphasize the obvious logical relationship between different levels of formality, and the need for some direction between them in order to make progress when constructing software.

The paper by Bonnet et al. [2] presents an example of refinement between two specification levels. It is interesting because it uses TLA, Lamport’s Temporal Logic of Actions (see [15] for a related reference), and this includes more abstract, less implementation “aware”, specifications. The paper includes “retrospective” proofs rather than constructive ones which, in the guise of applying pre-proven rules, may be more easily utilised by practitioners. This is another cultural dichotomy; viewing essentially the same activity from different ends of the construction process, from one standpoint the answer is ‘known’, from the other, it has to be found. Likewise, this paper also raises the question of whether parallelism should be indicated in a specification or whether a higher level of abstraction should be sought - from which many solutions architectures (serial, or distributed, or parallel) may evolve.

Other presentations at the workshop were more individual in character. That by Bustard and Lundy [5] might arguably be regarded as Requirements Engineering. It describes the optional use of LOTOS to clarify (specify?) soft system models, but the authors’ reasons for seeking formalisation are rather vague. We try to clarify the issues involved in section 4.

Similarly, Dunckley and Smith [7] claim to provide links between MERISE (a popular structured methodology in France) and Z. Actually it uses Z to underpin (parts of) MERISE, and betrays the authors’ lack of appreciation of how formalisms can deal with large complex systems.
Other papers adopt a more general stance. That by Tse and Cheng [24] tells us about ‘Noodles’. Here the content is not MI as such but a possible net-based model which attempts to explain how O-O analysis and design methodologies may be given an underpinning theory. Regrettably the paper suggests that O-O methods have no formal basis, but this does rather deny that the many years work on ADTs etc. have any relevance to O-O technology. It should be admitted that there is little in software engineering that is genuinely new, most of it is quite old, albeit repackaged and presented in a more accessible and more usable form.

Finally, we come to two papers which present a more global view. Hooker et al. [12] cover a great deal of material but essentially their paper is about a tool for the translation of a particular Z specification into ML. In a wider context it addresses the development of Ward-Mellor models into a variant of Milner’s CCS. By reading this paper alone the reader gets an overview of many of the numerous methods/theories associated with real-time system specification and modelling. Chasing its references is a worthwhile exercise for those who are unfamiliar with the field.

Even though the paper by Hinchey [11] has been cut down, so as to concentrate on a single large example - and, at the behest of the referees, side step much of the theoretical underpinning - it is still a mega opus. The paper epitomizes the author’s on-going search for an all-embracing specification and development ‘method’. He remarks that reactive systems are more complex than concurrent ones which in turn are more complex than serial systems, and sets about describing how we might tackle all of these in a single framework. In doing this he brings together much of the leading edge theory. Having made the point that many so-called methods are not methods at all and that some specification languages have no formal basis, he introduces the ‘Hi-Ride’ elevator system as his big running example. His system specification diagrams give rise to parallel compositions of RPT descriptions (RPT is Receptive Process Theory - derived from CSP) from which he moves to Lamport’s TLA (behaviour of components + fairness constraints). Effectively these are action specifications (given in Z) plus a superstructure indicating how they may/should be sequenced. From the computer science standpoint, his paper represents the most research oriented contribution to the workshop and is therefore somewhat out of place. However, should there be another such workshop ten years hence, we would hope that the material presented then would be more closely related to this paper than the rest; by then the work reported in the other papers should have either been superseded or absorbed into common usage.

All the papers mentioned above assist in developing and promoting a better understanding of various aspects of the software construction process. However, taken as a whole, they also emphasize differences in what is viewed as acceptable input (what form of ‘specification’) and what level of output is sought (a full implementation or simply a design at a lower level). Of most significance is the distinction between individual software components and complete systems and, in the latter case, the recognition that defining ‘what’ has to be done is not enough, we must also indicate ‘when’ certain actions are appropriate/allowed/necessary. How this extra requirement, and the extent to which implicit system properties or explicit definitions of operations (etc.), might be unified into an overall scheme will be addressed in section 6.

3 A Wider View

Methods Integration within the Software Engineering process is a misnomer. The ‘integration’ refers to attempts to form an harmonious and productive blend of (informal?) structured, and diagrammatic, methods with formal representations of software artefacts and/or their characteristics. Structured methods, with their extensive use of diagrams, are easily understood by systems analysts and students of, what in former years would have been called, data processing. On the other hand, Formal Methods is characterised by mathematical expressions and formulae.

It is true that diagrams can be interpreted formally and can therefore be regarded as merely another way of representing certain mathematical relationships; and ‘pictures’ are certainly useful when considering how mathematics problems may be ‘solved’ or how mathematical information can be manipulated - particularly if we have large amounts of information that we wish to restructure. By the very nature of present day computing machinery, the outcome of any successful software construction process is a segment, or
segments, of code written in some language which is executed in a deterministic fashion by the hardware employed. [Even if the execution includes a degree of variability - by use of ‘random’ values accessed by the software - the results delivered by the program from given data are in accord with the semantic rules of the language and the functional characteristic of the hardware]. The final system is a formal entity, unless the hardware ‘breaks’, the properties of the system are mathematically determined.

At the other end of the spectrum, because of the need to interact with less predictable entities such as people and (non-electrical?) machines, and the imprecision with which required systems are usually described, we start with an inherently vague, inaccurate and incomplete set of requirements. Now it is possible that specifications of desired systems can be intentionally vague, allow variation within certain tolerances, and include ‘unspecified’ aspects but it is not uncommon for specification to be, at least in part, not as intended - i.e. wrong! So, given that we can be precise about what is required of a computer system (including precision regarding any allowed lack of precision in the product!) the basic task is, therefore, to progress from a potentially faulty set of requirements - which we assume to be expressed informally - to a piece of software which drives hardware - absolutely formally - to behave in the way we require.

Put another way, we try to formalize and clarify/correct a specification without encroaching upon the software design process. Software development is then about the construction of abstract, but adequate, specifications which can subsequently be refined to more acceptable forms using various paradigms. Moreover, since any benefit which might be gained from using a formal specifications is put at risk if a less than formal refinement process is used in its design/implementation, the relationship between the formal and less formal approaches is directed. Becoming more formal adds details and lessens non-determinism, moving in the opposite direction without any essential loss of information is only possible if certain ‘informal’ entities have fixed interpretations - and hence we will have used some formal mechanism to define these entities. This amounts to the use of ‘formal methods’ to underpin the less formal.

Although the subject of refinement, and the use of specifications in general, has been reported widely in the journals over many years - indeed it has developed in leaps and bounds - it’s break-through into the text-book arena has been very slow. It will be interesting to see how the recent, and much delayed, text by Woodcock and Davies [26] will improve awareness of post-specification formal methods. Likewise, on the O-O front, the books written by Lano [16] and edited by Goldsack and Kent [10] and the many subsidiary references to be found therein ought to make an impact in explaining the theory and practical utility of O-O methods.

From appropriate formal specifications, systems can be mechanistically (not purely mechanically) constructed. For non-trivial systems - and it cannot be said too often that most end-user computer systems are computationally simple - we have a long way to go; we are still trying to understand the problems and devise general theories of how they may be solved. See Lowe and Zadeh [19] for a recent contribution to this quest. Indeed, we need to make practitioners (at all levels) aware of available and potential technology associated with all aspects of the software construction process - it is not just compiler writers that provide useful intermediate “protection” from computer hardware.

4 Terminology and Concepts

In attempting to cross the ‘formality divide’ we need to clarify what the various paradigms are attempting to achieve; and to say something about the misconceptions that are often prevalent.

Structured methods are concerned with ways of representing large (but not necessarily complex) data sets and associated inter-relationships, and their subsequent analysis/processing. What is being attempted here is generally well understood and although there are numerous ‘different’ methodologies, many of them are very similar; indeed variants of the same basic system are common - just consider how many are derived from the early work of Yourdon, e.g. Ward-Mellor (see [12]). The majority of situations where structured methods are applied (and indeed O-O methods, in the Information Systems domain) are computationally simple, but the volume of data and the required speed of access to it are often important. It is therefore perhaps a little inconsistent when the structure of the problem is strongly reflected in the resultant system design rather than the data being held in such a way so as to speed up processing.
Over recent years object-oriented methods of problem analysis and of system design have become very popular and may be regarded as a half-way house between formal and informal. Central to O-O methods are the concepts of encapsulated data types/classes and inheritance. Object-orientation is essentially about re-use; parameterisation of data types, of procedures and of software construction in general. Despite the outbreak of many languages with names of the form “X++” - pick your favourite “X” - the ++ postfix has nothing to do with O-O per se. C++ is not a good O-O language but, like many other examples in computing, happened to be around at the right time and satisfied a need - but it ought in all honesty be better to regard it simply as a new, improved, version of C. (Unfortunately it is more than just possible that there are those who do not appreciate that “++” is the unity incrementation operator in C.)

Much is currently said about “programming for re-use”; the desire to produce more general (sub-) programs. This excursion into clairvoyance involves, at the very least, the use of named constants to facilitate easier change or parameterisation, but opens up a debate about the economies of software production rather than the cost of running a particular program. Paradoxically we again have an apparent lack of concern about execution efficiency, bringing us full circle to the on-going debate on structure preservation (resulting in what are essentially simulation programs). As we shall propose in the next section of this paper, structured (problem) analysis followed by system design resulting software specification is fine, but there should be no compulsion to preserve structure.

A bolder step in the direction of re-use is presented in Bird et al.[1], “Generic Functional Programming with Types and Relations”. This is real parameterised programming; parameterised at a much higher level of “class”.

The more practical (less abstract, more mechanical) “methods” are less powerful but better understood. Formal methods, even the “what?” and “why?” of formal methods, are less well understood.

What is Formal Methods?

Within software engineering, the term ‘Formal Methods’ refers to the use of mathematics to construct and analyse software artefacts. Despite the common usage of the plural form, it is not necessarily concerned with a collection of methods - just as “mathematics” is a body of knowledge relating to the activity of doing mathematics, “formal methods” is a body of knowledge. Neither is it a ‘methodology’ (à la SSADM, Yourdon, JSP etc.), again like mathematics, formal methods is not prescriptive. Many would-be users criticise it exactly for this reason. This criticism merely serves to underline their lack of appreciation of what formal methods is really about. This is not an omission, formal methods were never intended to be methods as such.

The discipline of formal methods does not explicitly give you ‘answers’; there is still a need to understand the problem (such as the calculation you are trying to realise) and to solve it. Traditional methods of problem solving are still applicable but now they must be executed within the constraints of a formal (mathematical!) system. So, put the other way round, formal methods require that you solve problems within a set of constraints and hence deny the introduction of errors. The constraints are essentially the rules obeyed by the data types and languages used, and the rules of logical reasoning.

So, using formal notations, and having agreed meanings (semantics) for them, allows us to reason about software. Principally, we can formulate a notion of correctness, or compliance. Unfortunately, just as more people are appreciating what correctness and verification are all about, the process of verification is becoming sidelined (hidden from view for practitioners, required only by experts). The kinds of proofs that relate to software are of two kinds. Internally, we have correctness proofs (which are needed to show that a collection of “program” specifications are consistent with the (software) system requirements they are designed to meet, and similarly, that implementations/designs meet their specifications). In much the same way that few mathematicians actually perform proofs - they are too hard for most practitioners - they use pre-proven results, pieces of algebra, and ‘transforms’ etc.. Likewise, much verification work is actually performed by calculation, the correctness proofs being constructive - you show that a program is correct by constructing it from its specification. External to the software, we may wish to investigate consequences of use within the surrounding system. To do this requires relevant domain knowledge, and perhaps even a different logic. This
may be outside the realm of expertise of the software engineer but in any case should be argued from the software specification, not its implementation.

In certain problem domains general strategies may be discovered and be found to be ‘good’. From the formal methods standpoint these would not be enforced, except perhaps by local management in which case we get a house style and maybe a local methodology (handle turning?) which may be automated by appropriate tools in which the associated mathematics and software engineering is frozen. This approach could give rise to a safe and profitable harnessing of formal methods applied to a limited but widely used theatre of usage.

5 Separation of Concerns

Merely interfacing, structured and formal methods makes no sense. Moving from formal to informal serves no purpose except when the loss of information corresponds to a projection onto a specific realm of interest in order to investigate certain aspects of a “system”. It is, however, now recognised that, when employing structured methods to develop software it is often useful to express certain requirements (etc.) in a formal notation. Indeed, it could be argued that, if the ‘programming’ phase is to be carried out with minimal reference back to a systems analyst, the programmer should be given a specification from which he could, in principle, adequately test the acceptability of the code he produces; though he would usually not have the resources to do this. Regardless of how it is presented, such a specification is (by definition) formal.

5.1 On Structure

Most ‘methods integration’ work is concerned with augmenting structured methods with mathematical notations to produce formal specifications of software systems. In doing this, it is often the case that a larger system is not only specified but, in part, designed. Of course, no computer system works in isolation; the enclosing system needs specifying and designing, and part of its design will be the specification of the embedded computer system.

To allow maximum flexibility in the way that the software/computer system is built, the specification of this element of the overall system should state what actions (re-actions?) are required, under what circumstances such an action would be enabled (i.e. allowed), and, if there are conflicts arising from more than one action being enabled, what priorities apply.

Of course some software systems are complex but many software specifications are more complicated than they need be. Often they include detail which relates to a particular implementation strategy, or they include redundant clauses for no good reason. They should be structured so as to aid comprehension of the task in hand, not presented in a form that is predominantly intended to assist in implementation/execution.

The construction of specifications of software is a very important task and is not easy. It is a necessary prerequisite to the use of formal methods and may use “formal methods tools” such as animators/exercisers in ‘testing’, but it is not part of formal methods activity per se - if the original specification is informal then application of such methods is impossible, if it is expressed in a formal language then no construction is required.

It is not our intention here to promote one notation in preference to another but to indicate how the adherence to any structure (implicit or explicit) within a specification biases the subsequent design and, at least intellectually, may hide other acceptable designs.

Not only is there a case for initial software specifications to be abstract, and as natural as possible for the purposes of comprehension/validation - so, for example, output values need not always be expressed as the subject of defining equations and hence look like assignment statements - but other, even less explicit, forms of definition can be used.

Methods Integration Workshop 1996
At the system level we might only have a vague, yet adequate, requirement such as (for a heating system)

\[
\text{henceforth the room temperature should be between 20}^\circ\text{C and 25}^\circ\text{C}
\]

there being no mention of the actual procedures of turning the heater on/off, the cooler on/off etc. At some stage in the design of such a system, these, together with appropriate enabling and invoking conditions will be needed and these will be expressed in some “process algebra” such as TLA. However the links between these levels and subsequent code generation are phases in the refinement process and need not, should not(?) be included in the initial statements of required system behaviour.

An axiomatic or relational definition says how entities ought to relate to each other but gives no direct clues as to how they may be constructed. Used to describe system requirements this is very abstract (and very good in that it includes no hint of any design or implementation bias) and fosters a top-down design process. On the other hand, such an approach must be used to describe the properties of the basic data types from which software is built. This is clearly bottom-up.

Consider a partial specification of a stack of integers on which the operations push and pop act. The signatures of the operations are:

\[
\begin{align*}
\text{push} & : \mathbb{Z} \times \mathbb{Z}^* \rightarrow \mathbb{Z}^* \\
\text{pop} & : \mathbb{Z}^* \rightarrow \mathbb{Z} \times \mathbb{Z}^*
\end{align*}
\]

or, in a form closer to English,

\[
\begin{align*}
\text{push}( \mathbb{I} \mathbb{N}, \text{STACK}) & \rightarrow \text{STACK} \\
\text{pop}(\text{STACK}) & \rightarrow \mathbb{I} \times \text{STACK}
\end{align*}
\]

and the single rule, given by the equation.

\[
\text{push}(i,s_1) = s_2 \quad \text{pop}(s_2) = (i,s_1)
\]

Now any implementations of pop and push are acceptable provided that together they always satisfy this equation for all values (variables) of the appropriate types. (The symbol “\(\bowtie\)” is used to separate the two expression instead of the more usual “=” sign which is commonly used in specifications and programs.)

The implicit approach may also be used in proper (high-level, abstract) statements of system requirements; they are minimal logical expressions from which the acceptability of an implementation can be judged.

Taking a so-called “state-based” approach to the specification of operations and functions we describe, preferably also in an implicit way that is devoid of implementation bias, what each component should do - but not necessarily how. Structure (bias) is introduced progressively, choices being delayed for as long as possible. However, even the existence of these internal operations reflects some aspect of the design process. So structure, or more precisely the introduction of structure, is part of the design process. Rather than having to replace one structure by another (which is possible, and sometimes necessary) it is easier to introduce new structure. This can usually be done in several ways as we shall illustrate in the following example; an example of problem reduction and program synthesis i.e. refinement.

### 5.2 An Example: The factorial function (yes really!)

We start from the simplest, most abstract, specification (definition). It is not recursive, since recursion suggests some underlying structure. Adding (or exposing) structure is a software engineering decision which reflects a choice made during the synthesis of a program design. The example also illustrates how diagrammatic notions from structured methods may be used - informally - to assist in the development of program designs from a formal specification.

The factorial function has been the butt of recursive ridicule ever since stacks were invented. Nevertheless it is well-known and a useful example. We shall use it to demonstrate the derivation of two recursive schemes. Moreover we shall intentionally not make use of all the properties of multiplication and hence increase the utility of the underlying transformations. The presentation uses mathematical manipulations illustrated in part by data flow diagrams (DFDs).
For $n \in \mathbb{Z}$ and $n \geq 1$

$$\text{fact}(n) \triangleq 1 \ast 2 \ast \ldots \ast n$$

‘$\ast$’ is a quantifier (being associative, commutative and having 1 as identity) so we could write

$$\text{fact}(n) = \{ i \mid i \in \mathbb{Z} \land 1 \leq i \leq n \}$$

- i.e. multiply together all the ‘$i$’s where each ‘$i$’ is an integer with a value between 1 and $n$ inclusive.

or, so as to avoid using the prefix ‘$\ast$’ symbol for multiplication, we introduce the name ‘mult’, to give

$$\text{fact}(n) = \{ \text{mult} i \mid i : 1..n \}$$

We shall only use the associativity of mult which is of type

$$\mathbb{Z}^2 \rightarrow \mathbb{Z}$$

(i.e. 2 integers in, and 1 integer out)

Partly for emphasis, partly because it is very useful, we introduce an induced list operation

$$\text{mult}^+: \mathbb{Z}^+ \rightarrow \mathbb{Z}$$

$\mathbb{Z}^+$ is the set/type/class of non-empty integer lists. (The use of superscript ‘$+'$, and ‘$\ast$’ is common in formal computer science and relational algebra. The use of these ‘decorations’ on function names indicates the extension of the function to non-empty and arbitrary lists. For those familiar with APL, this is merely an alternative way of writing the reduction operator. mult$^+\times X \triangleq \ast / X$, where $X$ is a non-empty list of integers.)

We are going to calculate $\text{fact}(n)$ by explicit multiplication of numbers in the list

$$<1,2,...,n> \quad \text{so } \text{fact}(n) = \text{mult}^+(<1,2,...,n>)$$

mult$^+$ has the properties

$$\text{mult}^+(<x>) \bowtie x$$

$$\text{mult}^+(L1 \bowtie L2) \bowtie \text{mult}(\text{mult}^+(L1), \text{mult}^+(L2))$$

where $L1, L2 : \mathbb{Z}^+$ and $\bowtie$ is concatenation.

The general ‘divide and conquer’ strategy applied to this problem yields the DFD below.

\[
\begin{array}{c}
\text{split} \\
\text{mult}^+ \\
\text{mult} \\
L : \mathbb{Z}^+ (|L|>1) \\
\end{array}
\]

$$L : \mathbb{Z}^+ (|L|>1) \rightarrow \mathbb{Z}^+ \times \mathbb{Z}^+$$

$$\text{split} = (L1, L2) \text{ where } L = L1 \bowtie L2$$
Notice that “split” is non-deterministic; lots of different implementations are possible, we shall use three of them.

Recall that, when \( n > 1 \),

\[
\text{fact}(n) \triangleq \text{mult}^+\left(<1,2,\ldots,n>\right) \triangleq \text{mult}^+\left(<1..n>\right)
\]

Now take a software engineering decision and particularise (refine!!) split to be split_left, so

\[
\text{split}_\text{left}(<1..n>) \triangleq (<1>,<2..n>)
\]

Using split_left, the full DFD for \( \text{mult}^+\left(<1..4>\right) \) is as follows

```
\begin{verbatim}
<1..4>
  \arrow{split_left} \
  <1>
  \arrow{split_left} \
  <2>
  \arrow{split_left} \
  <3>
  \arrow{mult} + 3 \
  <4>
  \arrow{mult} + 4 \
  \arrow{mult} 2 \arrow{mult} 1 \arrow{mult} 24 \arrow{mult} 24
\end{verbatim}
```

The order of evaluation suggested by this diagram is essentially deterministic; the only variability being the order of executing the 4 ‘mult’ operations, each of which does no more than extract the integer \( x \) from the list \( <x> \). However, since mult is associative, we can ‘change the plumbing’.

Methods Integration - Reflection & Reorientation?

Methods Integration Workshop 1996
Moreover, by staggering the order of evaluation of the ‘mult⁺’ operations - so as to derive 1, from <1>, before splitting <2..4>, we get the DFD:

Now to express this mathematically and use it to calculate a procedural program design. In outline this goes as follows:

\[
\text{mult}^+(L) \begin{cases} \text{if } |L| = 1 \text{ then head_of}(L) \\ \text{else } \text{mult}(\text{head_of}(L), \text{mult}^+(\text{tail_of}(L))) \end{cases}
\]  
[using split_left]

Next take the ‘else’ branch and introduce a new function based on the ‘else’ expression.

Let \( \text{improve}_1(x;\mathbb{Z},y;\mathbb{Z}^+) \equiv \text{mult}(x,\text{mult}^+(y)) \)

From this definition we can calculate:

\[
\text{improve}_1(x,y) \begin{cases} \text{if } |y| = 1 \text{ then } \text{mult}(x,\text{head_of}(y)) \\ \text{else } \text{improve}_1(\text{mult}(x,\text{head_of}(y)), \text{tail_of}(y)) \end{cases}
\]

This is tail recursion and we can apply a standard ‘formula’ to restructure the calculation into the iterative form.

\[
\text{[declarations]} \quad \text{i.e. } u;\mathbb{Z}, v;\mathbb{Z}^+ \\
(u,v) \leftarrow (x,y); \\
\text{while } |v| \neq 1 \quad \text{do } (u,v) \leftarrow (\text{mult}(u,\text{head_of}(v)), \text{tail_of}(v)) \\
\text{od; } \text{mult}(u,\text{tail_of}(v))
\]

(We could ‘simplify’ further using the property that \(\text{mult}(x,1) = x\) and extend the definition of \(\text{mult}^+\) to give \(\text{mult}^*\) so that \(\text{mult}^*(<>) = 1\) etc. but the fewer extra properties used the more general is the applicability of the ‘calculation’.)
Substituting back in \( \text{mult}^+(L) \) and simplifying, using the programming language rules, gives

\[
\text{mult}^+(L) \overset{[\ldots]}{=}
\begin{align*}
z &\leftarrow \text{head}_\text{of}(L); \\
M &\leftarrow \text{tail}_\text{of}(L); \\
\text{while } |M| \neq 0 &\text{ do } z \leftarrow \text{mult}(z, \text{head}_\text{of}(M)); \\
&\quad M \leftarrow \text{tail}_\text{of}(M) \\
\quad \text{od}; \\
z &\text{ end}
\end{align*}
\]

- for future reference we shall call this program 1.

Of course, we could transform the data-flow diagram for \( \text{mult}^+(<1..n>) \) in other ways. Using a ‘split-right’ we would arrive at the DFD shown below. (Again some detail is omitted so as not to detract from the main features - i.e. the shape).

Using functions ‘front_of’ and ‘last_of’ to split

\( <1..n> \) into \( <1..n-1> \) and \( <n> \)

and by a similar calculation, we get:

\[
\text{mult}^+(L) \overset{[\ldots]}{=} \begin{cases} 
\text{if } |L| = 1 & \text{then } \text{last}_\text{of}(L) \\
\text{else } \text{improve}_2(\text{mult}^+(\text{front}_\text{of}(L)), \text{last}_\text{of}(L)) 
\end{cases}
\]

where

\[
\text{improve}_2(x: \mathbb{Z}^+, y: \mathbb{Z}) \overset{[\ldots]}{=} \begin{cases} 
\text{if } |x| = 1 & \text{then } \text{mult}(\text{last}_\text{of}(x), y) \\
\text{else } \text{improve}_2(\text{front}_\text{of}(x), \text{mult}(\text{last}_\text{of}(x), y)) 
\end{cases}
\]
which then transforms into

\[
\text{mult}^+(L) \triangleright [...]
\]

\[
\begin{align*}
  z & \leftarrow \text{last_of}(L); \\
  m & \leftarrow \text{front_of}(L); \\
  \text{while } |M| \neq 0 & \\
  \quad \text{do } z \leftarrow \text{mult}(\text{last_of}(M),z); \\
  & \quad M \leftarrow \text{front_of}(M) \\
  & \quad \text{od}; \\
  z & \rightarrow
\end{align*}
\]

Let this be program 2. The similarity of these two programs is only to be expected, after all we are simply processing the list in the opposite direction; but see below.

Thus far, we have been manipulating the calculations in a compositional fashion - i.e. we can change the content of any ‘box’ without the need to change any other parts of the DFD. This corresponds to operational refinement and is compositional. We can also change the way that data is held/stored. This is data refinement and this cannot be compositional - if we change the way in which one piece of software stores data then we must also change the way in which other software retrieves it.

The factorial programs (1 and 2) give a simple example of this. Instead of using the list \( M (= <p..q>) \) say we can simply use integer variables \( p \) and \( q \). All the references to \( M \) must then be changed and we achieve the more familiar versions of the factorial calculation.

Program 1 becomes

\[
\begin{align*}
  z & \leftarrow 1; \ p \leftarrow 2; \ q \leftarrow n; \\
  \text{while } p \leq q & \\
  \quad \text{do } z \leftarrow z * p; \ p \leftarrow p + 1 \\
  & \quad \text{od}; \\
  z & \rightarrow
\end{align*}
\]

and optimises to

\[
\begin{align*}
  z & \leftarrow 1; \ p \leftarrow 2; \\
  \text{while } p \leq n & \\
  \quad \text{do } z \leftarrow z * p; \ p \leftarrow p + 1 \\
  & \quad \text{od}; \\
  z & \rightarrow
\end{align*}
\]

Likewise program 2 becomes

\[
\begin{align*}
  z & \leftarrow n; \ p \leftarrow 1; \ q \leftarrow n - 1; \\
  \text{while } p \leq q & \\
  \quad \text{do } z \leftarrow q * z; \ q \leftarrow q - 1 \\
  & \quad \text{od}; \\
  z & \rightarrow
\end{align*}
\]

which similarly this gives

\[
\begin{align*}
  z & \leftarrow n; \ q \leftarrow n - 1; \\
  \text{while } 1 \leq q & \\
  \quad \text{do } z \leftarrow q * z; \ q \leftarrow q - 1 \\
  & \quad \text{od}; \\
  z & \rightarrow
\end{align*}
\]
Notice we could even use “split_equal”, so

\[
\text{if } |L| > 1 \text{ then } \text{split_equal}(L) = (L_1, L_2)
\]

where \(2^*|L_1| = |L| \lor 2^*|L_1| = |L| + 1\) \(\land L = L_1 \bowtie L_2\)

This would then give rise to

\[
\text{mult}^+(L) \begin{cases} 
\text{if } |L| = 1 \text{ then head_of}(L) \\
\text{else mult}(\text{mult}^+(L_1), \text{mult}^+(L_2))
\end{cases}
\]

where \((L_1, L_2) = \text{split_equal}(L)\)

With our running example this would give the DFD

![DFD Diagram]

Here, the two embedded references to \text{mult}^+ can be computed in any of ways so far encountered. In particular, using integer division and \textit{parallel} execution we could have a program such as

\[
\text{fact}(n) = [...] \\
m \leftarrow n \div 2; \\
( z_1 \leftarrow \text{mult}^+(<1..m>) || z_2 \leftarrow \text{mult}^+(<m+1..n>) ); \\
\text{mult}(z_1, z_2)
\]

So, the “shape” in the specification need not be reflected in a resulting design/implementation. We suggest that it is not desirable that the structure of a specification should be imposed on any subsequent design and hence it is desirable that a specification should have a no unnecessary structure; and where a degree of structure is necessary to cope with “size” that structure should be as general as possible.

### 6 A Generic Framework

Using more abstract descriptions at all levels and hence avoiding over specification will enable us to give smaller, more comprehensible yet adequate set of conditions which define a software artefact - even one which is “large”. Of course, finding appropriate levels of abstraction and focussing attention on the problem rather than a solution is far from easy but this aspect of software engineering should not be avoided; unfortunately it often is. The recent text by Jackson addresses this point in a most elegant and approachable way [13].

From a statement of requirements or a specification the design process creates control structures and data structures which are consistent with the implied demands and constraints. This is proper software engineering and should refer back to the above descriptions rather than to the external sources from which those descriptions were derived.

Although the construction process is described as being top-down (as it must be for a new system) it can be partially executed bottom-up when reusable components are employed and hence the implementation level is effectively lifted.
Diagrammatically this is set out as follows:

```
System Requirements  abstract description of desired behaviour
     ↓                             ↓
System Design       System Design
     ↓                             ↓
SW Specifications etc., Operational Refinement, Software Designs
     ↓                             ↓
Data Refinement, Implementation
     ↓                             ↓
Operational Refinement, Software Designs
     ↓                             ↓
Implementation, (exactly) How
```

So we start by describing the desired system, characterising it by its required/allowed behaviours. This can be done by giving its persistent or eventual properties. Without mentioning any specific operations, describe the effects that you wish to achieve or maintain. Technically this requires temporal qualification, based on the logical notions of “henceforth” and “eventually”.

E.g. in the heat control scenario:

```
hereforth room temperature is in the range [15°C .. 20°C]
```

or, in an account package:

```
henceforth when access is allowed, debits = credits.
```

These characterisations are expressed in terms of external properties, no operations, no structure, no design and as such ought to fall into the emerging (sub-) discipline of Requirements Engineering. Indeed there is a school of thought that says that this aspect of software construction is exactly what requirements engineering is about, no more, no less. It is therefore slightly worrying that the new "Requirements Engineering" journal [21] actually mentions the “design of complex systems” as one of its areas of interest - surely that goes beyond “requirements”. Again we refer the reader to [13] in an attempt to reach a more general agreement of what this term actually means within our sphere of interest.

Next we move to systems level refinement. Here we must identify - invent - software/system operations/calculations which satisfy/achieve these requirements. It has to be admitted that this is a less well developed aspect of formal methods for which refinement ‘rules’ are still being evolved. In this phase we must derive specifications of activities, actions, reactions, operations, and calculations and how they may connect. Typically these may involve turning a heater on/off, issuing of commands that cause a train to accelerate/decelerate, or transferring funds within an accounting package. These can be specified by ‘traditional’ pre and post conditions. But there is also the need for a definition mechanism at a level above VDM-SL or Z, something like CSP, CCS, or perhaps less elaborate (less complex?) such as LTS, statecharts, Petri Nets, TLA, all of which have precise semantics. The important thing is to get this initial, unrefined, specification right and make no presumptions about implementation.

Finally we can apply "standard" refinement theory to achieve explicit program designs. This incorporates operational refinement, data refinement, optimisation etc. to generate a design which can be translated into executable code.

Within this ‘master plan’ there is a problem when a software practitioner wears two hats; those of both customer and provider. In this situation he may, inadvertently but legitimately, perform data refinement during the requirement or specification phase. This merely leads to confusion of roles and purpose since...
software design precludes its specification. In order to clarify responsibilities and operational dependencies, the logical ordering illustrated above ought to be followed; although, in cases of re-use, the implementation level may not be actual code but existing module specifications.

A consequence of separating system requirements from software specification (by an intermediate refinement process) is, in part, to deny the need to integrate formal and informal methods, but to emphasize the need to augment, extend, the specification notations such as Z and VDM with a ‘process algebra’ level which indicates - implicitly or explicitly - how operations and functions should be combined/sequenced. The structuring of informal methods is required in the initial, top level, phase and traditional formal specifications arise from a design process. They cannot truly integrate since they are separated by an intervening process. This idea is not a new one [6].

We respectfully suggest that formal “methods” don’t need to be integrated with structured methods but specifications of functions and operations, must be augmented with a process algebra layer to say how these may be sequenced.

7 Conclusion

In trying to draw conclusions from the papers “presented” it is also useful to take stock of the more general situation; to ask

where are we going?
how far have we got? and
do we still want to go there?

At the risk of mixing metaphors

What bridges are we trying to cross?

- there are several different bridges, they may interconnect (or they may be parallel), but we only require them to be one way!

Formal and informal characterisations of software are both very important within the software construction process. They are used for different but related purposes. Likewise, so-called ‘structured methods’ and ‘formal methods’ play their part in software construction but are not truly integrated - as design proceeds the descriptions become more formal, more deterministic.

[It is regrettable that certain groups within the software engineering community regard different paradigms as in conflict. The inaugural meeting of the SIG in the Teaching of Software Engineering actually had “Formal vs Structured vs Objected-Oriented” in its promotion literature. This could have been to incite debate but it is sad that anybody in the educational arena should even hint that they are incompatible.]

In this paper we have attempted to clarify the interrelationship between informal and formal aspects of software engineering and suggest how they may be ‘interfaced’.

A possible utopian (but perhaps questionable) scenario is to have a general purpose specification mechanism. The idea being that there would be no need to learn a completely new system for different kinds of software. This is reminiscent of the aspirations of Algol-68 [18], one of the greatest advances in programming, and it was missed by almost all of the non-academic world. (They are now in the process of reinventing most of it.) Many workers search for a unification of underlying software theory; see [3] in which Broy describes progress in his work on the mathematical foundations of Structured Methods and notions of refinement.

Problem-based diagrammatic representations (such as Labelled Transition Systems, statecharts etc.) play a part; initially they should not imply any non-essential structure and later they can be used, informally, in subsequent refinement manipulations.

At the system/software interface the use of Z, VDM-SL etc. in “mini-specs” plus a specification superstructure is advocated. Instead of grafting methods or notations, we suggest that one notation should be extended to include this extra layer - e.g. Statecharts or TLA(Z), TLZ, see [11]. Some convergence is actually taking place but in rather a clandestine fashion, O-O technologies are converging/rationalising (see [22]) and the notations of Z and VDM are gradually converging despite attempts to prevent this by freezing them in individual standards.
We are still generating lots of case law, lots of CASE studies (and CASE tools, which are obviously necessary for the eventual widespread use of formal methods, but are perhaps still being constructed on the basis of insufficiently mature theory). We first need to get the process as near right as possible - but commercial pressures probably won’t give us the time to do this. The overall scheme might go something like this:

- identify, and isolate, each level of the construction process
- at each level, be as abstract as possible in the statement of requirements
- justify the solution - either by construction or retrospectively (in terms of lower level artefacts)
- accept responsibility for your part in the design
  (i.e. if the substructures work as they should then what you have built will satisfy its requirements.)

This last point amounts to the standard engineering practice of signing off a piece of work for which you take responsibility. To make this a practical proposition needs a clearer stratification of (sub-) system responsibilities. Such a stratification means that validation becomes more important - it always was but this was not widely recognised - and emphasizes the role of requirements engineering in the creation of adequate (but design free) statements of requirements.

As regards “methods”, there is a trend to diversification. For practising software engineers there will be structured and Object-Oriented methods with hidden and fixed semantics; they will be limited in scope (but will be adequate for the domain in which they function) and, if any elements of formalism remain visible they will be derived from simplified subsets of todays specification languages - perhaps an extra outer, structuring, layer added to a collection of ‘mini-specs’. This will allow the research community to divert its energy to the development of proper theory and promote its use in the construction of more reliable software systems in a wider range of application domains.

Remember “In the land of the Informalist the person who knows a little Z notation is king (or someone who nobody talks to!)”. We must continue to strive to ensure that all software engineers understand what others are doing and that they all talk to each other, otherwise we shall keep re-inventing software wheels.

References


Methods Integration - Reflection & Reorientation?


