The Purpose of Specification

Dr R G Stone
Department of Computer Studies
Loughborough University
Leics, LE11 3TU
R.G.Stone@lboro.ac.uk

&

Dr D J Cooke
Department of Computer Studies
Loughborough University
Leics, LE11 3TU
D.J.Cooke@lboro.ac.uk

Abstract

Specifications, that is to say formal specifications in a computing context, should be correct, concise and complete and intelligible to their intended audience. We consider the changes in our own thinking about notation and styles of writing specifications and our success (or lack of it) over the years in teaching the ideas behind specifications. In particular we are interested in the choice of specification notation to carry the design process by a series of transformation steps.

Keywords: POP-I.B.Choice of language, POP-II.B.Formal Specification, POP-III.C.Functional, POP-IV.A.Transformation

1 Introduction

A common criticism of formal specifications of systems is that they are unreadable by anyone other than their creator(s). The factors blamed for this impenetrability are principally size and notation. This paper reflects on our continuing search for solutions to these problems. It includes observations on the way that programming languages have been perceived and designed over the years and tries to anticipate what a similar progression might produce in the world of specification.

In the very beginning writing computer programs was a job for specialists only. After all it was hard work before compilers came on the scene. As time went by the perception changed so that everyone and anyone could write programs if they had a mind to. At this time all students on Computing courses wanted to learn to 'program'. The current situation has changed so that many students on Computing courses positively do not want to do any programming at all. This trend begins to put programming back in the hands of those who have chosen it as a speciality. Eventually this will be good news for teachers (like us) because we will end up teaching programming only to well motivated students, but at present we are still counting the cost of trying to make programming approachable to all.

If we now look at the progression of the perception of specification, in particular formal specification, we can see that it is going through the same evolution as programming but has not yet progressed so far. It would be good to short-circuit the phase of 'specification for everybody' and subsequent disillusionment and proceed straight to specification for specialists. This theme is taken up from opposite directions in [5] which reports the lack of success in teaching the ability to read formal specifications and the section by Gries in [6] where he reports enthusiastic uptake of the use of a system of equational logic, a possible computer-based tool for which appears in [4, 7].

2 Specification

2.1 Specification for Specialists

The construction of correct, efficient software is not for the amateur, nor for the computing novice. Placing tasks in the hands of relevant specialists will relieve us from the need to study the difficulties experienced by people trying to program using inappropriate languages, paradigms and systems. What is important is that the requirements of the system be adequately conveyed to the software engineer. Abstract specifications or requirements (as opposed to pseudo implementations) can then be drawn up and agreed so as to provide an acceptable and usable interface between those who want such a system and those capable of building one. References [1, 2, 3] provide further discussion on the "separation of concerns" within the software construction process.

2.2 Specification for Non-Specialists

An analogy that we want to make is to compare the task of writing program specifications with the specialised area of the specification of the syntax of a programming language. Many textbooks show syntax diagrams which display the grammar rules of a programming language in a compact way that is readily acceptable to a large audience. The same syntax has to be reworked by a compiler writer using standard techniques to produce the specification of a mechanical parser for the language. This refined specification is usually longer and more difficult to assimilate. However it is not intended to be seen by the users of the resulting compiler. We are on the lookout for compactness, clarity and ease of comprehension in all specifications, initially at the customer/designer boundary and also at every subsequent level where the audience is different. The big question is whether we can do this in general or only in specific areas (like the specification of syntax).

One reason for the apparent complexity of many specifications is that they include detail which they ought not to. Similarly it is often the case that system requirements are not sufficiently abstract; they are frequently cluttered up with elements of design. The various aspects of the system construction process should be separated so as to clarify the intent of each phase, to simplify the (minimal yet adequate) descriptions required in each intermediate design, and each of these designs should be expressed in a language suitable for its target audience. Presently we do not recognise that there are different audiences, customers are not (usually) systems engineers, who are seldom software engineers. What we need to recognise and promote is that the engineering of specifications into implementation designs is a technical and specialised task - not for the feint hearted, not for the amateur. The same is true of the “distance” between a statement of system requirement and a system design - here the gap is filled by a systems engineer and it is part of his job to draw up the specifications which are then passed to the software builder. The entire process therefore consists of various experts in diverse but related fields. They need to talk to each other but they need not be conversant with the details of each others tasks, they do not need to speak or read each others' private languages. There is no need for a customer to delve into (internal) specifications of software components. It is perhaps only a mark of desperation and lack of confidence that some customers demand to see these specifications.

3 Abstraction

One reason for undue complexity in specifications is that they are not given at a sufficiently high level of abstraction. In order to give substance to the concept of abstraction, consider as an example the history of the specification of sorting. In relation to computer programming, sorting was originally perceived as being associated with swapping numbers round in an array until the desired condition was reached. After a while this was abstracted to dealing with lists in which the notion of a labelling of elements by indices could be dropped. At this stage the specification of sorting a list would typically require that the result was a permutation of the original. This concept is considerably simplified by abstracting to bags (or multi-sets) where the permutation requirement simplifies to equality of the bags formed from the input and output. Of course the move from arrays to lists also allows a single specification to encompass what have become known as external sorts as well as internal sorts.

Somewhere in the specification there will be a part which fixes the result required as ascending or descending order. In the quest for abstraction we might reasonably ask what is the concept that embraces and unifies both ascending and descending order. One suggestion is the notion of contrast. In viewing the elements of a list one by one, the amount of contrast between each element and the next can be measured. The concepts we know as ascending order and descending order result from minimising the total contrast. [It is often worth experimenting with a new abstraction to see what new avenues it opens up - for example in this case we could ask what is the result of maximising contrast instead of minimising it?]

4 Programming by Construction

4.1 Notation

We are interested in program construction by transformation of specifications. So far we have been discussing the development of a notation and a fluency in writing down abstract specifications. Now we look at the process of transforming the specification into an implementation. [This is in contrast to inventing an implementation and then proving it correct.] In other words we are looking at a way of formally recording design steps each with their justification.

We have already mentioned some of the history of our notation in relation to the problem of sorting. We have found that it takes considerable training before our students can firstly read and then construct simple specifications that rely on quantifiers and implication. We have recently been experimenting with new ways of encoding the quantification and implication that are trademarks of abstract specifications. The kind of specification extract that we wished to find another way of presenting includes universal quantification, existential quantification and implication as follows

for all ... , there exists ... such that <u>*this phrase*</u> *implies* <u>*that phrase*</u>

Our current replacement, which is an extension of the set-of-all idea, is of the form

'and' together all the versions of <u>*that phrase*</u> *that can be constructed on the basis of* <u>*this phrase*</u>

The apparent benefits if this notation might be attributed to a combination of the following changes. Firstly the the key phrase *that phrase* has been repositioned at the beginning. Secondly all the quantification is handled at one level in the set-of-all-versions-of construct and the explicit occurrence of quantifier symbols has been removed. Thirdly the dependence on implication has been hidden. This last point is indicative of the numerous rules for manipulating quantified expressions. The trick would seem to be to find a form in which the balance between emphasis and hiding simplifies the subsequent synthesis of a software design.

4.2 Transformation Steps

Programming by Construction now requires that we take the specification and, working within the rules which govern the relevant data types etc., apply appropriate problem solving strategies to reorganise the specification into a design for an executable program . All terminating calculations are completed in a finite number of stages and all pieces of valid data are also finite. Therefore we must also either replace infinite types by finite approximations or take other steps to guarantee finite ranges etc. so as to ensure that the bag of values fed into the set-of-all term is finite. For example in a sorting problem, one way of expressing the ascending condition involves counting the number of elements which are 'out of place'. Such measures are useful since, as part of the synthesis procedure, we shall have cause to introduce recursion. When this is done we need to ensure that we use a well-ordering, a strict reduction in some characteristic of the problem which uses the positive integers. This guarantees convergence of recursion. Moreover if the recursion translates into iteration then that iteration will terminate by construction.

There are a number of generally applicable design tactics which can be applied. Using our own naming system we can categorise them in four groups. Firstly there is the domain split where a property of the data structure (such as the length of a list being less than or equal to one) is used to provide separate cases to consider. Secondly there is the structural split where the structure itself is split (for example splitting a list into two non-empty sublists). Thirdly there is a predicated split where the individual components of a data structure are grouped according to a predicate (for example partitioning the elements of a list into two groups of elements which do (or do not) satisfy the property of being less than or equal to a certain value). Finally there is the tactic of problem reduction. This is simply the tactic of expressing the problem at hand in terms of a related problem which is "smaller" than the original problem but has the same answer. These tactics are normally applied in combination. For example domain splitting is usually a necessary introductory stage, the need for which becomes apparent when trying to apply one of the other tactics. For example a structural split only is only possible when the data structure is 'big' enough so the first tactic is a domain split into 'trivial' and 'non-trivial' cases.

Having established our favourite tactics we find that students often have the insight as to what might be a successful tactic but write down almost complete gibberish when asked to justify their choice. Originally we took the view that the ideal design presentation was one which continually and constructively transformed the specification until all the parts were expressible in the chosen programming language. We have found this too strict and too demanding and we are now content to allow small Eureka steps. This means that instead of transforming a part of the specification into a particular form we invent a new form and, by 'substituting it in', check that it satisfies the original part of the specification. For example, in a number sorting problem we might use a predicated split to work with two groups of numbers in which the numbers were respectively below and above a certain value. The design might proceed by producing a sorted list from each group. It is then much easier to demonstrate the intuition that the 'answer' is the concatenation of the two sorted lists rather than trying to deduce this by further constructive transformation of the original specification.

5 Some Open Questions

Some of the questions that the psychology of programming community might have answers to, or consider trying to find answers to in the context of formal specification are listed below. They are listed in approximate order of grandness of scope and of course the way in which the question is phrased may betray our preferred answer.

(i) Is 'implies' really a hard concept or have we just not found the best way to teach it? The truthtables for 'not', 'and', 'or' & even 'nand', 'nor' are instantly acceptable to students whereas the truthtable for 'implies' is mysterious and problematical. The presentation as if ... then ... is helpful, but only really delays dealing with the "else what?" issue.

(ii) Is Quantification a harder concept than 'for'-loops? Explaining that universal quantification expands into a conjunctive expression with one term for every possible value of the quantified value is like explaining that a 'for'-loop is really like writing out multiple copies of the loop code in sequence substituting each different value of the for loop variable in turn. But quantification is regularly shown over infinite collections of values and has no inherent sequencing. The retrospective derivation in quantified form of the logical properties - invariants and variants - of program loops can be very complex and could be said to have been the inspiration for the constructive development process.

(iii) Is our proposed quantification notation really likely to be more acceptable, based as it is on the set-of-all concept? Of course we are convinced but then we are biassed!

(iv) (This is the one that we hardly dare ask.) Are we right to pursue the constructive programming approach in the hope (or expectation) that it will eventually become, after contributions to notation and style by innovators and after gaining familiarity, as attractive as any other method of programming? That is to say as attractive as any other method of programming which produces code that meets a formal specification. What we are really seeking is a way of putting Science into Computer Programming, and Engineering into Software Engineering.

6 Conclusion

The advice often given to students of programming is that simple, obvious, clean, short is good. By implication fancy, clever, tricky, long winded is bad. This advice holds good for specification as well. The problem is that instead of working within the confines of a fixed computer programming language, making a specification simple requires positive, creative, abstract (dare we say) mathematical thinking. Having written a suitable specification there comes the task of transforming a specification by a series of justifiable steps. This relies more on the mathematical concepts of algebraic manipulation and (program) calculation than the traditional craft of 'writing a program'. These are both jobs for a specialist!

References

[1] Butterworth, R.J. and Cooke,D.J., "Using Temporal Logic in the Specification of Reactive and Interactive Systems", in proc. "Formal Aspects of the Human Computer Interface", C.R.Roast and J.I.Siddiqi (editors), Sheffield, September 1996, pub Springer, Electronic Workshops in Computing.(1996)

[2] Cooke, D.J. and Stone,R.G., "The Interface Between Information Systems Engineering and Software Engineering", in proc "Information Systems Teaching, Improving the Practice" (ISTIP92), Sunningdale, September 1992, pp11-16. (1992)

[3] Cooke, D.J., "Methods Integration: Time for Reflection (and Reorientation?), in proc "Methods Integration", Anthony Bryant and Lesley Semmens (editors), Leeds March 1996, pub Springer, Electronic Workshops in Computing. (1996)

[4] Cooke, D.J., Stone, R.G., Bridson, A.J., Collett, M.C., The Loughborough contribution to the "WISDEN CD-ROM" - Containing the Software Engineering teaching materials from the TLTP2 funded project, led by Prof. Allan Norcliffe, Sheffield Hallam University.

[5] Finney, K., "Mathematical Notation in Formal Specification - too difficult for the masses", IEEE Transaction on Software Engineering Vol 22 No 2, pp 158-159 (1996)

[6] Saiedian, H., "An Invitatio n to Formal Methods", Computer, Vol 29 No 4, pp16-30 (1996)

[7] Stone, R.G., "Gadgets for Learning Discrete Mathematics in Computer Science", submitted to the Computer Journal.