CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Universidade do Minho: RepositoriUM

7.10

MODELLING IS FOR REASONING

Luís Soares Barbosa[1] and Maria Helena Martinho[2]
Minho University, Braga, Portugal

Abstract–*In a broad sense, computing is an area of knowledge from which a popular and effective technology emerged long before a solid, specific, scientific methodology, let alone formal foundations, had been put forward. This might explain some of the weaknesses in the software industry, on the one hand, as well as an excessively technology-oriented view which dominates computer science training at pre-university and even undergraduate teaching, on the other. Modelling, understood as the ability to choose the right abstractions for a problem domain, is consensually recognised as essential for the development of true engineering skills in this area, as it is in all other engineering disciplines. But, how can the basic problem-solving strategy, one gets used to from school physics: understand the problem, build a mathematical model, reason within the model, calculate a solution, be taken (and taught) as the standard way of dealing with software design problems? This paper addresses this question, illustrating and discussing the interplay between modelling and reasoning.*

1. INTRODUCTION

The use of computer-based systems to support mathematical modelling is a recurring theme in the practice and research of most of our readers. On the one hand, computers have radically expanded the range of problem-solving and decision-making situations that can be effectively tackled. On the other, they play a fundamental role in training modelling skills and promoting associated competences. In this paper, however, we take the dual viewpoint: we will not be concerned with computers as modelling aids, but instead with the use of mathematics to model and reason about computer-based systems. Maybe such a shift of concern deserves some explanation.

The exponential increase of both the availability of processor power and the complexity of the problems computers are requested to solve, is unprecedented in any other engineering domain. Even so, software remains hard to develop, it is often unreliable ('faulty goods delivered over budget and behind schedule'), difficult to re-use and excessively costly to modify and maintain. Traditional design methods emphasising diagrammatic or textual descriptions, with an informal semantics, have created the illusion that software development was little more than a balanced compromise of intuition and craft.

As a result, *conceptual* questions are often relegated to a secondary level of attention, and the mastering of particular, often ephemeral, technologies appears as a decisive requirement, for example, on recruitment policies. Often, in industry, the whole software production is totally biased to a specific technology or programming

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

system, encircling, as a long term effect, the company's culture in quite strict limits. In a broad sense computing is an area of knowledge from which a popular and effective technology emerged long before a solid, specific, scientific methodology, let alone formal foundations, have been put forward.

This situation has to be contrasted, however, with the increasing demand for *quality certified* software, namely in safety-critical systems, which requires development approaches in which a system would be unacceptable unless accompanied by a *guarantee* that it respects a rigorously *specified* behaviour. This is the point where, from our perspective, mathematics comes into the picture. Or, more precisely, where software development is framed as a *mathematical modelling activity*.

In fact we are beginning to collect the fruits of more than four decades of intensive, even if sometimes neglected, basic research on the foundations of computation and programming semantics, upon which a true engineering discipline for software design can be based. Such research helped, in particular, to shed light on the underlying mathematical structures and reasoning principles, and to establish the connection between mathematics and computing at a foundation, rather than application (as, *for example,* in numerical analysis), level.

In such a context, the starting point of this paper is that to become a mature engineering discipline, software design has to adopt the basic problem-solving strategy one gets used to from school physics:

- understand the problem,
- build a mathematical model of it,
- reason in such a model,
- upgrade the model whenever necessary,
- calculate a solution, which, in this domain, means a *program*.

Moreover we would like to underline two concepts in this strategy:

- *Modelling*, understood as the ability to choose the right abstractions for a problem domain;
- *Calculation*, in the sense that such abstractions should be expressed in a mathematically rich framework to enable rigorous reasoning both to establish models' properties and to transform models towards effective implementations.

The context for this research was the assessment of two concrete educational experiments specifically designed to introduce modelling as a central issue in the computer science curriculum. They were conducted at both undergraduate and professional, post-graduation, training levels.

Some lessons learnt from these experiments are reported in §4, which concludes and raises some topics for further research. Before that, §2 and §3 discuss, respectively, the role of modelling and calculation in software design.

2. SOFTWARE DESIGN AS A MODELLING ACTIVITY

Software design is concerned with the engineering of computer-based solutions to real-world problems in which information, its acquisition, flow and transformation, plays a key role. The starting point is often a collection of more or less structured requirements, usually stated in plain English. Consider for example

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

the following fragment of a statement of requirements placed by a mobile phone manufacturer:

> *For each row of calls stored in the mobile phone (for example, numbers dialled, SMS messages, lost calls, etc.) the store operation should work in a way such that (a) the more recently a call is made the more accessible it is; (b) no number appears twice in a list; (c) only the last 10 entries in each list are stored.*

In the analysis of an information system a fundamental distinction is drawn between *entities*, which represent information sources, and *transformations* upon them. A similar distinction appears in the definition of *algebras* (as collections of sets and functions) which makes them interesting models for this sort of systems. An elementary grammatical analysis of the requirements above provides the basic ingredients:

- *Nouns* (such as “call” or “raw of calls”) lead to the identification of the fundamental information structures, or data domains, which, at a latter stage, will originate what is known as the program *data types*.
- *Verbs* (such as store) identify the services to be made available from the system. They denote processes which transform information, and therefore can be modelled by *functions* or, more generally, by *relations*.
- *Integrated sentences* (such as requirements (a) to (c)) identify *properties* or *constraints*, corresponding to predicates which tune the model to its specific purpose.

Software designs can be naturally expressed in the centenary notation of set theory. Notions such as set, sequence, Cartesian product, function or relation have the potential to provide expressive, but rigorous, descriptions of a design. For several years in their past education students (and professionals) have become familiar with this sort of notation as a *tool to think with*. Our courses in software design intend to build on such a background.

In the example considered here, a “row of calls” can be modelled as a sequence of whatever a “call” is. Registering a new call in the row amounts then to place the former in front of the latter, a well-known operation in the algebra of sequences that we denote by the **:** (read *append*) combinator. This leads to elementary model shown in Figure 1.

$$CallRow = Call^*$$
$$store : Call \times CallRow \longrightarrow CallRow$$
$$store\ (c, l) = c : l$$

Figure 1. An elementary model.

Note that data domain $Call$ is left unspecified: the initial requirements do not place any restriction on this structure, so, at the level of abstraction of this model, it is considered a primitive notion, *that is,* an unstructured element in the universe of discourse. Are we done? Is this model acceptable? Let us address these questions going through the problem constraints already identified:

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

- Constraint (a) is guaranteed by construction, *that is,* it is a direct consequence of the definition of service $store$.
- Constraint (b), on the other hand, can be stated as equation

$$\# \bullet elems = length \quad (1)$$

that is, the number of calls in a row (as measured by function $length$) is equal to the cardinal ($\#$) of the set of its elements (as computed by $elems$). Note that regarding a sequence as a set eliminates all duplicated occurrences of its elements.
- Finally, constraint (c) imposes a limit on the length of the sequence used to model a row of calls. It can be documented by the inequality

$$length \leq 10 \quad (2)$$

Therefore, to meet constraints (b) and (c), operation $store$ has to be modified: when storing element x in sequence l, l must be first depurated of any occurrence of x, and, after appending, the whole sequence reduced to its first 10 elements. Formally,

$$store(x,l) = take_{10}(x : filter_{\neq x} l) \quad (3)$$

where $take_n$ and $filter_\phi$ are combinators in the algebra of sequences. The former returns the first n elements of a sequence, the latter filters out elements which violate predicate ϕ.

This small example illustrates the *iterative* character of the modelling process: one starts with a very bare, but precise, model which evolves as the understanding of the problem increases. It also shows the fundamental role of the simple notion of a function in modelling software engineering problems. To be precise, not only the notion of a function, but that of the whole *algebra of functions*. As any other algebra, this defines the ways in which functions can be combined. And there are a number of different ones. Function composition provides a *pipeline* connection between functions with the right types:

$$A \xrightarrow{f} B \xrightarrow{g} C$$

Often in design classes one explores an analogy between composition of functions and multiplication of numbers. In this way, students soon realise that non commutativity of composition leads to *two*, rather then *one*, division problems for functions[3]. What is interesting, from our point of view, is that each of them has a particular modelling potential. Consider, for illustration purposes, diagrams in Figure 2, in the context of the mobile phones example. In each case, a function x, acting as an unknown, has to be found to close the triangle. In the first case x computes the amount to be paid for a call in a particular price plan. This means that through x function $type$, which assigns a price plan to each call, determines the cost of a call (given by $cost$). In the other example, x associates a call to a particular

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

network. Closing the diagram means that for each call a network is to be found such that the origin of the call and the location of the used network coincide.

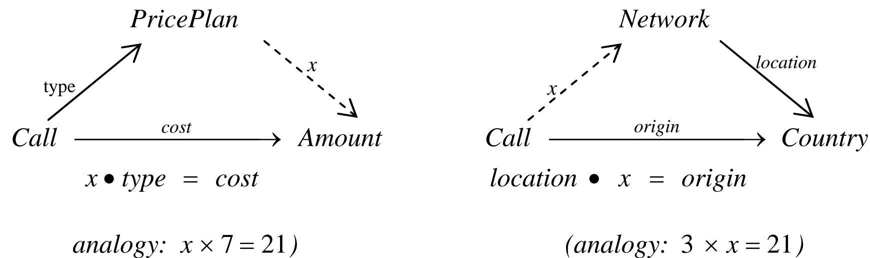

Figure 2. Composition vs Multiplication.

Combinator $\bullet$ provides a gluing scheme for functions, but it is not the only one. In particular, functions with a *common domain* can be glued by a *pairing* construction

$$C \xrightarrow{\langle f,g \rangle} A \times B$$

where $\langle f,g \rangle x = (fx, gx)$ Similarly, functions sharing a *common codomain* can be put together by an *alternative* construction

$$A + B \xrightarrow{[f,g]} C$$

which applies f or g depending on the argument coming from A or B ($A + B$ stands for the disjoint union of sets A and B).

The relevance of these composition patterns is that they correspond to different ways of composing services in a (model of a) computational system. Moreover, each of them entails a different way of modelling information aggregation. For example, *pipelining* leads to the notion of *function space* $A \rightarrow B$, which models functional dependencies. On its turn, *pairing* leads to *Cartesian product* $A \times B$ modelling *spatial aggregation* of information. Finally, alternative leads to *disjoint union* $A + B$, which models *choice* or *aggregation in the temporal dimension*.

In this section, we have highlighted the use of *functions* as a modelling tool. Of course, other problems may require different conceptual tools. For example, one may resort to *partial functions*, to model problems which remain undefined for a number of situations, or *relations*, whenever the outcome of a service is non deterministic (and therefore a functional dependency does not exist between its input and output). Or one may resort to some form of *automata* to express the dynamics of a computation. In contrast to other, more classical engineering disciplines, *continuous* mathematics is not the primary problem-solving tool here. Actually, classical models are manipulated either analytically or numerically, often resorting

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

to some sort of testing on a physical model of the problem. Software design, by contrast, resort to *discrete* mathematics, which is easy to understand and animate in a computer, but usually no physical models are available: computer science deals essentially with non tangible mathematical models (what Henderson (2003) calls *mental models*).

Similarly, however, to what happens in other engineering disciplines, the purpose of a model in software design is double: to provide insight into the problem/system structure, and to form a basis upon which one can *reason* about such structure. The latter is a fundamental step: it is the ability of calculating within design models that paves the way to the possibility of transforming them into effective programs and computational systems. This leads directly to the second topic of this paper.

3. MODELLING IS FOR REASONING

There are two ways in which the title of this section can be understood in the context of computer science education. In one sense it means that a model should be amenable to *experimentation*. In the other that it must provide a basis for *effective calculation*, for example to verify the equivalence between two designs or to transform one into the other by controlled introduction of detail. Let us discuss each of them separately.

Although mathematical notation is a very good way of expressing requirements and of communicating among the design team, it requires more and more precision from people. Furthermore, writing mathematics does not mean to write everything perfect at the first time. So, there is a need for tools for validating mathematical descriptions. Moreover, educational practice has shown that to be effective the whole modelling process must be supported by some sort of animation tool. That is, a computer-based tool which understands an elementary language of sets and functions and executes designs.

There is a variety of available animation tools used either in educational or professional contexts (see for example, Fitzgerald & Larsen, 1998; Abrial, 1996; Almeida et al., 1997). Such tools build a *prototype* out of a (formal) model, which can be executed, tested and modified on-the-fly. This is also an old idea in Engineering. Think, for example, in a wind tunnel test of an aircraft, where performance in checked against theory, or a mock-up for a building, in which design features are checked for usability. From our experience the use of prototypes provides:

- Early feedback on the model.
- Increased confidence in the models developed achieved by a check on its self-consistency and general sensibleness.
- More effective communication among the design team.

Furthermore it emphasises the incremental and iterative character of the software design process. Prototypes develop side by side with formal models, from the very beginning until a stable and detailed design is found. Each iteration is formally documented, and, what is more, such a document is executable.

But when is a software design equivalent to another? Or to a sub-model thereof? How can a real program be extracted from a design model? How can a particular

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

property be shown to hold of a given model? How is a model built to satisfy a set of properties? To answer these sort of questions is the purpose of a design calculus. But what calculus?

The definition of a *calculation style* to reason about software models is an essential ingredient to the success of the modelling approach. Actually, there is a well-established reasoning style in mathematics, the *theorem-followed-by-proof*, which is quite inadequate for the construction of computer systems. The reason is that it reflects a *guess-and-verify* approach which supposes the system is first built (out of the blue?) and then formally verified. If one has a model, however, the reasonable attitude is to use it to calculate the system, making progress through a whole chain of progressively more concrete models. A number of authors have discussed the dichotomy *verification-oriented* and *calculational-driven* styles of reasoning (see for example, Gries & Schneider, 1993; Zeitz, 1999; Backhouse, 2001) and concluding on the ineffectiveness of the former to Computer Science. There is also an extensive body of research on calculi for transforming software design models (see for example, Bird & Moor, 1997; Backhouse, 2003), which we actually use in the modelling classes.

Although this is not the proper place to introduce such calculi, we would like to briefly comment on a related issue which is often neglected: *notation*. Actually expressiveness in modelling and suitability for calculation may seem potentially conflicting aims. Mathematical modelling requires *descriptive* notations, often domain-specific, and hopefully intuitive. Calculation, on the other hand, requires notations that are *generic, concise and precise* (Backhouse, 2003) or, to put it in another way, *elegant*, in the sense the word has in the writings of Dijsktra: *simple and remarkably effective* (Dijsktra & Scholten, 1990), *that is,* easy to manipulate.

The extensive use of nested quantifiers in a logic formula, for example, may provide what one may think of as an intuitive description of a problem, but makes manipulation of such descriptions an uneasy, even overwhelming task.

Such a trend for *notational economy* is well-known throughout the history of Mathematics, as a sort of “natural language implosion”. The driven force has always been the same: facilitate formulae manipulations, therefore enriching its suitability for calculation. Contrast, for example, formula:

$$.60.\tilde{p}.2.ce \quad son \quad yguales \quad a \quad .30.co$$

used by Pedro Nunes, a Portuguese mathematician of the 16th century, in his *Libro de Algebra*, published in Coimbra, in 1567, with nowadays $60 + 2x^2 = 30x$.

Again the history of mathematics is full of examples in which not only different notations, but also different, although interrelated, conceptual domains are used for *modelling* and *calculation*. The former emphasises expressiveness and closeness to intuition, the latter manipulation simplicity. A classical example is the Laplace transform, which allows an expressive but complex model to be converted into a less intuitive but simpler (*that is,* linear) one.

Is there a similar *transform* to reason about software designs? The answer turns out to be very simple: just *avoid the variables*. In particular, in the algebra of functions briefly discussed in the previous section, replace function application by function composition and look for definitions in terms of generic properties rather

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

than *ad hoc* representations. The reader may recognise here the whole discipline of category theory (Mac Lane, 1971), but we will not elaborate further on that.

4. CONCLUDING REMARKS

As reported in the Introduction, the context for this paper was a reflection on two concrete experiments at Minho University. Both experiments have been conducted for five years now at two quite different levels: *first year undergraduate students in a computer science degree* and *professional training* at post-graduation courses for software engineers.

Although students' age, backgrounds and motivations are quite different between these two groups, we have found extremely relevant the explicit incorporation of modelling in the computer science curriculum. In particular we have been able to assess how this contributes

- To emphasise the *conceptual* rather than the *instrumental* aspects of an engineering carrier[4].
- To develop *design literacy*: reasoning flexibility and, as Lesh and Doerr (2003) put it, *a handful of models in your hip pocket*.
- To enhance both *communication* and *teamwork* skills.

From a technical point of view modelling and reasoning are intertwined. Moreover emphasis should be placed on the *construction* rather than the *verification* level, a point that has often been neglected in research. Another lesson learnt was that, in computer science as in mathematics, notations are not neutral. Well designed notations do make the difference when one has to reason upon a model. Also, as already commented, the crucial need for tool support, in particular for prototyping systems.

Formal concepts of the kind required by computer science, and both modelling and problem-solving skills develop slowly along long periods of time. Our experience with professional engineers that return to the University to participate in this sort of seminars, suggests such training adds up and is probably effective even when initiated later in life.

Modelling in software design, as in any other domain of application, enhances what is known as *mathematical fluency* (see Lesh, 1996; Kaput & Shaffer, 2002), which is at the heart of what it means to understand. In more general terms, however, assessing to what extent mathematics education, at both university and pre-university levels, is centred on the on going construction and revision of models rather than on the acquisition of self-contained (?) bodies of knowledge remains an open question. We believe there is still a long way to go in that direction. Actually, acquisition of facts, results and procedures are merely surface manifestations of what goes on when people learn. As Devlin (2000) points out, *we know they are surface phenomena since we generally forget them soon after the last exam is over*.

Finally, a word on the role of the 'teacher'. Our experience, however limited it is, suggests she/he is more likely to be expected to act as *coacher*, than as repository of pre-framed knowledge. The insistence on new educational practices would not be effective without an assessment of how typical university lecturers feel about that and how this interacts with their own images of their profession. Also at this level, further research is certainly needed.

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

NOTES

1. DI-CCTC, Minho University.
2. IEP, Minho University.
3. See Lawvere & Schanuel (1997) for a detailed discussion.
4. The following opening statement of Paul Halmos autobiography (Halmos, 1985) is particularly elucidative, written as it was by a mathematician, which in the 1950's, was director of doctoral studies in what was then one of the top Mathematics Departments of the world, in the University of Chicago: *I like words more than numbers, and I always did (...) This implies, for instance that in Mathematics I like the conceptual more than the computational. To me the definition of a group is far clearer and more important and more beautiful than the Cauchy integral formula.*

REFERENCES

Abrial, J. R. (1996) *The* B *Book: Assigning Programs to Meanings*. Cambridge: CUP.

Almeida, J. J., Barbosa, L. S., Neves, F. L., and Oliveira, J. N. (1997) CAMILA: Prototyping and Refinement of Constructive Specifications. In M. Johnson (ed) *6th International Conference on Algebraic Methods and Software Technology.* Sydney: Springer Lecture Notes in Computer Science (1349), pp554–559.

Backhouse, R. (2001) Mathematics and Programming. a Revolution in the Art of Effective Reasoning. Inaugural Lecture, School of Computer Science and IT, University of Nottingham.

Backhouse, R. (2003) *Program Construction*. Chichester: John Wiley and Sons, Inc.

Bird, R. and Moor, O. (1997) *The Algebra of Programming*. Series in Computer Science. Hemel Hempstead: Prentice-Hall International.

Devlin, K. (2000) *The Math Gene: How Mathematical Thinking Evolved and Why Numbers Are Like Gossip*. Basic Books.

Dijkstra, E. W. and Scholten, C. S. (1990) *Predicate Calculus and Program Semantics*. New York: Springer Verlag.

Fitzgerald, J. and Larsen, P. G. (1998) *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge: CUP.

Gries, D. and Schneider, F. (1993) *A Logical Approach to Discrete Mathematics*. New York: Springer Verlag.

Halmos, P. R. (1985) *I Want to Be a Mathematician*. New York: Springer Verlag.

Henderson, P. (2003) The Role of Modelling in Software Engineering Education. *33rd ASEE/IEEE Frontiers in Education Conference*. Boulder.

Kaput, J. and Shaffer, D. (2002) On the Development of Human Representational Competence from an Evolutionary Point of View. In K. Gravemeijer, R. Lehrer, B. v. Oers and L. Verschaffel (eds) *Symbolizing, Modeling and Tool Use in Mathematics Education*. Amsterdam: Kluwer Academic Publishers.

Lawvere, F. W. and Schanuel, S. H. (1997) *Conceptual Mathematics*. Cambridge: CUP.

Lesh, R. (1996) Mathematizing: The real need for representational fluency. In C. Janvier (ed) *20th Conference of the International Group for the Psychology of Mathematics Education*. Valencia: Universitat do Valencia, 3–13.

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.

Lesh, R. and Doerr, H. M. (2003) Foundations of a Models and Modeling Perspective on Mathematics Teaching, Learning, and Problem Solving. In Lesh, R. and Doerr, H. (eds) *Beyond Constructivism*. London: Lawrence Erlbaum Associates Publishers.

Mac Lane, S. (1971) *Categories for the Working Mathematician*. New York: Springer Verlag.

Zeitz, P. (1999) *The Art and Craft of Problem Solving*. Chichester: John Wiley and Sons, Inc.

Mathematical Modelling (ICTMA12) Education, Engineering and Economics
ISBN 978-1-904275-20-6 Chichester: Horwood (2007) 510pp.
Editors: Christopher Haines, Peter Galbraith, Werner Blum and Sanowar Khan.