Rethinking Programming

W M Beynon, R C Boyatt, S B Russ
Computer Science, University of Warwick, Coventry CV4 7AL, UK

Abstract

The accepted view of programming, rooted in Turing's fundamental characterization of algorithms, has had a profound impact on the theory and practice of computing with yet broader implications for thinking about mind and culture. Where programming is traditionally conceived in terms of requirements, specification and implementation, this paper argues for a complementary conceptualization to support the development of the next generation of computing applications. It briefly reviews an extended programme of research into Empirical Modellng, an approach to creating interactive environments to enable programming based on ***identification*** *and* ***prescription****.*

Keywords: programming, specification, identification, prescription

1. Introduction

Research into different paradigms for 'computer programming' has been dormant for many years. The established view of computer programming is nevertheless problematic in some respects. How programming is conceived exercises a strong explicit constraining influence over the theory of computing, and a more indirect but pervasive negative influence over thinking about computing practice. Different programming paradigms reflect different ways of understanding and observing application domains, and though each may be well-suited to specific kinds of application, they cannot in general be used coherently in combination.

This paper argues that in developing the next generation of IT applications it will be vital to rethink computer programming. Specifically, programming has two complementary ingredients: the *identification* of patterns of reliable agency and state-change to embody the interaction with 'the computing machine', and the *prescription* of recipes to meet specific functional goals. The term 'identification' is used here in a broad sense to encompass the idea that a context for computation is not typically merely found, but engineered (cf. [6]). The term 'prescription' entails specifying what agents act and how this action is mediated by stimuli to which they can respond. It must be emphasized that 'identification' and 'prescription' are intended to designate activities that lead to the discovery of actual environments, artifacts and mechanisms that can be directly constructed, observed and experienced, and are terms intended to be used only with such concrete products and recipes in mind. This level of concrete engagement with experience is what should be read into the expressions "reliable agency and state-change to embody interaction" and "specifying what agents act and how this action is mediated by stimuli". In this context, it is helpful to think of computer programming of the most primitive nature, such as was current before the advent of abstract programming languages. Identification was then associated with creating and configuring an electronic device, and prescription with giving instructions to the user about the physical actions required to supply the input data and initiate the program execution. In understanding what is meant by 'rethinking programming', the major conceptual difficulty is that of appreciating that, whatever the level of sophistication of programming languages, there is always an implicit underlying account of how a program operates with reference to physical devices and explicit actions.

In the history of programming, the two ingredients of identification and prescription have been addressed in different ways according to the evolving nature of the programming environment, the underlying technology and the human agency involved. Identification and prescription are necessarily interrelated, in that the validity of a prescription is predicated on the reliability of the patterns of interaction that have been identified. Though these ingredients may need to be developed concurrently through incremental evolutionary design, the process of programming can be radically transformed by recognising their conceptual separation. In particular, the first ingredient of programming, that of identification, involves activity that is typically not specific to any particular functional goal, and is intimately linked with understanding agency within the domain in which the application is to operate.

The elaboration of the above ideas will be complemented by a brief review of Empirical Modelling (EM), an approach to creating interactive environments, extensively developed by the authors and their collaborators over many years [20], that specifically targets the identification aspect of programming. The term 'modelling' is used here in recognition of the fact that identification is a more primitive and general activity than programming, as it relates foremost to domain understanding. In this respect, identification resembles and overlaps with requirements capture, but does not typically give such prominence to the articulation of functional goals. As has been discussed at length elsewhere, the greater generality of EM prompts its consideration as a broader foundation for computing than the science of programming alone can supply.

2. Programming revisited

The concept of 'computer programming' is by now so well-assimilated into information technology that it no longer attracts academic attention. The idea of developing new programming paradigms has long since fallen out of fashion. It is widely appreciated that focusing on paradigms for giving instructions to the computer has limited value – the real challenge in building large software systems lies in capturing the system requirements and arriving at the specification of the processes that are to be implemented in software. What is more, the computing culture has developed to such a level of sophistication that raw programming activity has become much less prominent. Many people make heavy use of computers in their work or studies without ever having to be conscious of the basic elements of programming that were once essential prerequisite knowledge. Special-purpose tools and packages help to account for this, as do object-based environments and interfaces that hide the explicit generation of code. A significant development in such end-user programming applications is the shift towards environments based on spreadsheets, databases or rule-based methods that cannot readily be interpreted as varieties of classical computer programming executing beneath visual interfaces or at higher levels of abstraction.

Of course, there is still an important role for specialist computer programming skills. Computer science departments and companies continue to train professional software developers, and there is major interest in niche applications, such as programming games or special-purpose hardware devices, where optimisation is so vital that even low-level programming expertise is required. Educational software is one area where there is still motivation for getting the non-specialist involved in programming computers, either through training teachers to develop – or at any rate customise – software for their personal use, or through pursuing the ideals of constructionist learning, as first pioneered by Seymour Papert [15]. A level of expertise beyond that of the naive user is also significant wherever generic applications prove to be unsuitable, and special needs must be met. Notwithstanding this, one significant side-effect of side-stepping programming in modern computing culture has been a growing tension between theory and practice.

The underlying orientation of the theory of computing has changed little since its inception; it still places a conventional view of computer programming at its core. Seen in this light, the science of computer programming, and the theory of computation, are the key disciplines around which the whole of classical computer science is organised. The study of algorithms and the semantics of programming languages are then the foundation upon which all computing applications are deemed to rest.

The practice of computing, by contrast, has developed to embrace much more than computer programming alone. The task of large-scale program development is of such complexity and subtlety that a complementary culture of concepts and methodologies for software development has emerged. The increasing prominence of new media for computer-based interaction and communication has focused attention on the experiential aspects of computer use, motivating much greater concern for understanding how cognitive and computational activities are related, and how they can be integrated. The computer’s role in supporting activities that were once viewed as peripheral to programming, and its pervasive influence in design, business and engineering, challenges the idea of computer programming as the central and universal characteristic activity in computing.

Many different intellectual positions are represented in the response of the academic computer science community to this challenge. Some regard the theory of computation and its applications as defining the boundary of real computer science, and the only academic focus of study that is truly independent of the relatively ephemeral technologies and fashions that characterise computing practice. Others seek to demonstrate that the scope for applying science to computing can be radically enhanced through developing ever more sophisticated mathematical semantic models and specification techniques, and complementing these with empirical studies of programming practice. Some adopt a pragmatic engineering-oriented attitude to computing, and regard it as a broad applied science. Yet others consider computing-in-the-large to be a discipline to which hard scientific principles are of marginal importance, and locate computer science in a subfield of social studies. Whatever their outlook, most commentators would agree that, whilst the precise concept of 'effective procedure' that was introduced by Turing and others has proved to be a powerful abstraction that has brought clarity and insight to the study of algorithms, there is as yet no comparable more general 'theoretical' framework that can help us to determine which of the broader activities carried out under the banner of 'computing' are well-conceived and will represent a long-term contribution to the emerging discipline. They would also recognise the need for principles to guide emerging practice, and to discriminate between good and bad techniques and applications.

The problematic nature of the relationship between computing theory and practice has implications for students and practitioners. Students who are attracted to computing through applications can find the discrepancy between informal and formal approaches to programming demoralising. Their aspiration is to use the computer in a creative and imaginative fashion, but they are obliged to conform to a discipline that obliges them to think in highly abstract and technical terms. Even those students who adapt well to learning computer programming are conscious that it is distinct from other essential computing skills. They also have to come to terms with the fact that the processes by which complex systems are developed are far from well-understood, and are typically the focus of controversies that are not resolved even after a specific choice of methodology has been made. All in all, the informed insider's view of software development is altogether more complicated and laced with uncertainty than that of the software adopters. In application areas such as education and business, it is unusual for those who champion the use of technology to recognise the potential relevance of these areas of doubt and obscurity for the computer specialist. This leads to conflicts similar to those experienced by pupils who learn mathematics in a constructionist idiom at school through writing programs in Logo, only to find that its procedural style is deprecated by computer scientists at university.

3. Programming = identification+prescription

It is helpful to review different varieties of computer programming activity with reference to the conception of programming as identification-and-prescription introduced above. This helps to elaborate on the meanings of these two ingredients of programming. It also has the side-effect of clarifying the way in which the meaning of the term 'programming' has developed, and sheds further light on its influence on computing culture.

The most appropriate setting in which to appreciate the idea behind programming as identification-and-prescription is that of programming a reactive system (cf. [9]). Within such a system there are many state-changing agents – they might include actuators, sensors and electromechanical devices as well as human agents with different levels of privilege to effect changes of state. Before any assumptions can be made about reliable patterns of interaction within such a complex collection of agents, much empirical study of their possible organisation and interaction must first take place. Even when the identification required to support prescription has reached a mature stage of development, it is still probable that the reliability of patterns of interaction will depend upon factors that are beyond absolute control. For instance, reasonable engineering assumptions will have to be made about the tolerances within which devices operate, which unavoidable natural hazards have been taken into account in the operating environment, and the extent to which successful or safe operation relies upon human discretion. In the narrow sense, the identification and prescription required to program the software components in such a system cannot in general be considered in isolation, since the assumptions about patterns of reliable interaction to which they are subject are intrinsically bound up with assumptions about the system as a whole. This argues for evolution of software that takes place in conjunction with the entire system development, in a manner that can perhaps only be abstracted from the whole task after a high degree of stability has been attained in the system conception and design. In effect, at any rate in the earlier stages of devising a reactive system, computer programming in the narrow sense involves identification and prescription that is inseparable from identification and prescription that is required to "program the system" in the broadest sense.

As Harel remarks in [9], programming reactive systems is much more problematic than what he calls "one-person programming" applications. In the early batch-processing applications of computer programming, the programmer scarcely had any need to address the identification aspect of programming. The computing machine was a given (even if its reliability was more often suspect!) and the agency of the user was very limited. The embodiment of the interaction with the machine only required the design of suitable conventions by which data was to presented to the computer and returned to the user. The programmer's primary task was then prescribing recipes. It was in this context that the archetypal view of computer programming was established: programs computed an input-output relation by executing an abstract algorithm in a fashion that was closely matched to Turing's mathematical model of computation.

As the very connotation of the term itself suggests, 'programming' is first and foremost associated in traditional use with *prescription*. This emphasis makes most sense in the classical programming setting. The fundamental thinking that surrounds batch programming is that the program prescribes the sequence of actions to be carried out automatically by the computer, that this prescription can be set out in a formal high-level programming language, and is amenable to all manner of abstract transformations that still compute the same intended input-output relation. This establishes a way of thinking about prescription that has prevailed throughout all subsequent developments of the programming context. It is also fundamentally misleading.

In general, the product of the identification activity is quite different from an 'input-output' computer. In programming a reactive system, prescription is associated with conceiving how the various agents are to interact in a sense that perforce engages with the actual observables that mediate their interaction, and has to take into account the peculiar characteristics of each component, whether human, electronic or mechanical. In establishing that such a system has been 'correctly' programmed, it is significant that – for example – sensors react only to particular kinds of stimuli within suitably maintained environments, that they respond within certain tolerances, typically autonomously and instantaneously, whereas human agents require visual or aural stimuli that are sustained long enough to attract intention and be interpreted, respond much more slowly and have discretion about whether they respond at all. In this context, there is in general no counterpart of the textual program for prescription, and often no evidence apart from empirical data to justify any claims about the correctness of a recipe, or to assess in what sense alternative recipes are equivalent.

Formal approaches to programming aspire to address this problem. A key notion is that of *specification*. A key aspect of the specification concept is already implicit in the classical programming scenario: a high-level computer program is itself a formal linguistic artefact that is potentially amenable to reasoning, and – at the same time – is in some way sufficient to prescribe the actual pattern of physical interactions that mediate between the input and output the user and the computer. In this context, the fact that specification indeed can act in this dual role relies heavily on two simplifying assumptions: that what is required of the program is no more than the computation of an abstract functional relationship, and that the time taken for this computation is immaterial both in the sense that the user can wait, and that the machine will remain oblivious to external agency. How far this notion of specification generalises to other programming contexts is unclear. Many sophisticated specification methods make it possible to describe the intended outcome of a program precisely in ways that are useful in reasoning without also admitting an interpretation as a prescription such as, for example, a classical declarative program does.

4. Motivations for rethinking programming

In classical programming, the separation between identification and prescription is so sharp that the programming activity can be seen purely as prescription. As the programming context has become more subtle, the aspiration for software development methods has been to seek a process of identification that delivers an abstract specification to serve a dual descriptive/prescriptive role in respect of every software component. This is to presume that, in the identification process, the structure and pattern of interaction amongst the state-changing agents will stabilise, and the ways in which this interaction is mediated will be expressible in terms of functional relationships between identifiable observables whose values can be computed. This can be viewed as trying to reproduce the classical duality between 'engineering the computing environment' and 'programming the computer'.

This approach to generalised programming succeeds in certain contexts and domains. It can work well, for instance, where the identification of agents and their interactions conforms closely to an object-oriented ideal, so that each agent is a programmable device with a repertoire of standard operations for changing its own state, and can effect changes to the state of other agents only via message passing. The primary agency and interaction in a mobile telephone network is of this kind. It is appropriate when the actual configuration of agents and interaction patterns is already well-established and is proven to be effective. This might be the case when the agents of an existing system can be replaced by programmable electronic devices that can mimic their interaction faithfully, as when a routine manual activity is being automated. It can apply to situations in which the patterns of agency and interaction in a domain are so well-understood and circumscribed for the purposes of the application that they can be predicted from theory. This is the case in typical applications of a 'scientific computing', or 'engineering simulation' nature.

One response to the challenge presented by reactive systems outside such categories has been to try to connect identification more closely with theories of the domain. In this spirit, Turski and Maibaum ([18], p.100) promote the idea of 'specification building [as] a lot like theory construction in science'. Such an approach is reductionist in spirit; it seeks to generalise the rational techniques that work in a 1-person programming context to such an extent that they can cope with the wilder problems of identification that reactive systems present. By way of illustration, Turski and Maibaum express their concern "that [the] very large class of … so-called real-time applications … is almost entirely dominated by specifications constructed from pragmatic observations" ([18], p.16), and envisage that "a much more concise and general theory would arise form the study of relations and functional dependencies between events than from the registration of absolute time-intervals". In effect, they propose that better ways of analysing and representing behaviours will enhance the role of reasoning, and diminish that of experiment. This outlook is consonant with the emphasis placed on the development of 'the verifying compiler' in meeting the Grand Challenge of developing dependable systems [19].

Tensions between pragmatism and theory are apparent in other aspects of complex software development. The "inevitable intertwining of specification and implementation" [17], and the topical interest in eXtreme Programming [3] testify to the fact that specification techniques alone cannot express the essential content that is associated with identification. Similar issues arise in specifying the semantics of families of interacting objects, and are reflected in the motivation for Harel's proposed development of software using Play-In scenarios [10]. Loomes's critique of reification in software development [13] also calls into question the extent to which abstract specification can be interpreted as prescribing concrete experiences of interaction in general.

Engineering practice provides a helpful alternative perspective on identification. The development of a complex system targets the creation of reliable patterns of interaction and interpretation in a very general sense, and these may be exploited in ways quite distinct from that associated with classical computer programming. Such a development may proceed through the construction of a succession of artefacts involving holistic observation supported by iteration and experiment. In such activity: abstraction, formal specification and reasoning need not play a significant role; reliability may be based upon no more than specific empirical evidence; there may be no comprehensive specification of function in input-output terms; nor any language in which to express the possible ways in which the product of identification can be 'programmed' to participate in qualitatively different interactions. In this context, it may be that the product of identification is an end in itself, in that it serves as a useful interactive environment like that associated with the classical computer itself, with a public building, or with a musical instrument.

Whether or not we would regard such engineering activity as itself a form of generalised programming, it highlights highly significant issues concerning identification in the modern programming context. Well-conceived engineering design activity delivers much more than a functional system. The nature of the development process makes it possible to take experiential aspects of an emerging system into account during its development. Because design decisions are based on experiment, the engineer typically has empirical knowledge of how the components of the system might behave "in the neighbourhood" of their normal working environment – that is to a say, if abnormal values are encountered, or exceptional circumstances arise. For similar reasons, good engineering design practice makes it possible to understand and audit the relationship between how the system is devised and set up and the implicit assumptions that have been made about its environment.

This view of engineering practice is no doubt to some degree idealised. There are nonetheless practical reasons to suppose that programming as practised can benefit from such an empirical approach to design. Reactive system failures have been attributed to predictable commonplace singular events, such as a battery running down, for which no provision has been made in the abstract model of normal behaviour. Pragmatic iterative development has delivered working Internet protocols for which there is as yet no formal specification, and where attempts at development using more formal procedures have been unsuccessful [16]. In contexts where model-building has been used to support historical investigation, as in the virtual reconstruction of Roman theatres by Beacham et al [2], it has become apparent that being able to adapt models to take account of different interpretations is of the essence. This is not to deny the vital importance of techniques of verification and optimisation in effective software development (consider, for instance, the security loopholes that formal analysis has disclosed in the Needham-Schroeder protocol after many years of apparently uncontroversial use), but to argue for a richer integration of formal and empirical perspectives.

5. Broader implications

The implications of rethinking programming are potentially much broader. Intellectually, the remarkable nature of Turing's achievement in characterising computational procedures has had a major influence on our models of mind. Because of the pervasive nature of computing, the way we think about programming has also had a profound effect upon our ideas about society and culture. It is tempting to construe ourselves as computational agents responding to our environment in ways that – at some level – can be rationalised as directed towards specific functional goals. The analytical stance associated with classical programming and specification can too easily be read – or misread – as endorsing such a conception of human behaviour. Consider for instance Turski and Maibaum's contention that: "Most frequently, the attempts to provide a computerized service for an application domain are at the same time first attempts to provide a workable theory of the domain." ([18], p16), or Gabbay's comment to the effect that "A detailed rule based effective but rigid bureaucracy is very much similar to a complex computer program handling and manipulating data. My guess is that the principles underlying one are very much the same as those underlying the other." ([7], p.ix).

This paper contrasts two discourses about programming, one expressed in terms of requirements, specification and implementation, the other in terms of identification and prescription. The *rational* discourse is primarily concerned with prescribing behaviour on the basis of certain knowledge, as in teaching and learning that involves revisiting what has already been established, or conforms to recognised pedagogical patterns. The *pragmatic* discourse is primarily concerned with identifying stable interactive environments and exploring what kinds of behaviours can be prescribed and interpreted within them, as in the personal learning activity involved in making sense of what is unfamiliar. In rethinking programming, it is necessary to support both these discourses, and find a coherent perspective from which they can be understood in their relation to each other. This has been one of the principal motivations for our research into Empirical Modelling. By way of conclusion, we shall present a brief outline of EM, together with references to many auxiliary papers specifically relating to this theme (see [20], where specific papers will be referred to by their publication indices).

EM is centrally concerned with activities relating to identification. These are associated with investigating the observables, dependency and agency that characterise the application domain. The reliable patterns of agency and dependency that relate observables are embodied in artefacts whose state is specified by a network of functional dependencies similar in character to that underlying a spreadsheet. These artefacts are constructed using special-purpose notations for expressing dependencies between observables of many different kinds, and hybrid tools that support dependency maintenance, triggered actions and a range of standard procedural programming constructs. The initial motivation for developing EM principles was to devise an approach to program development that would combine the best qualities of declarative and procedural programming idioms. The combination of declarative and procedural elements in EM resembles that to be found in the spreadsheet: definitions declare the extant functional relationships between observables, and a set of redefinitions, executed in parallel, represents a family of concurrent actions on the part of agents. In this context, each redefinition represents an atomic action, and the consequent changes of state that are entailed in the set of actions are mediated by the dependencies. An EM artefact is in the first instance a personal construction, similar to what Gooding [8] has characterised as a 'construal', with an informal semantics implicit in the interpretations of interactions that are projected on to it by the modeller.

EM has been studied in relation to design, modelling and programming activity in a wide variety of application domains. These include: engineering design, educational technology, humanities computing, and decision support. In these domains, EM respectively provides: a framework for supporting concurrent design activity [20:034]; a medium for model-building in the constructionist idiom [20:080]; principles for building artefacts to express and negotiate personal meanings [14]; support for open-ended decision-making based on subjective and qualitative criteria [20:061]. Note that, in all these areas, the qualities of spreadsheet models have already been observed (cf. [1, 12]). The practical contribution of EM has been to further develop principles and tools for modelling with dependency in support of the pragmatic discourse on programming with which spreadsheets are primarily associated. EM also proposes a perspective on computing that unifies the rational and pragmatic discourses. A simple model that illustrates one aspect of this unification can be found in [20:051], and a deeper discussion of the significance of this agenda in relation to mathematics and computing in [5]. From a philosophical perspective, unifying rational and pragmatic discourses on programming is bound up with longstanding controversies about the relationship between logic and experience. Consider for instance Livingston's observation, in his discussion of the controversy that surrounded Husserl's phenomenological method: that, "for deep-seated and internal reasons, the logical structure of experience may not be expressible in linguistic terms" [11]. The strong links that can be made between EM and William James's 'philosophic attitude' of Radical Empiricism offer a possible way to address this issue [4].

6. Acknowledgements

We are indebted to Jane Sinclair for many discussions relating to the general theme of this paper, and to Martin Campbell-Kelly for prompting us to revisit programming.

7. References

[1] J. E. Baker and S. J. Sugden. Spreadsheets in education – the first 25 years. *Spreadsheets in Education*, 1:18-43, 2003.
[2] Richard Beacham and H Denard, Roman Theatre, Frescos, and Digital Visualisation: Internedial Research, in *Proc. 4th Int Symposium on Virtual Reality, Archaeology and Cultural Heritage*, 2003
[3] Kent Beck. *Extreme programming explained: embrace change.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[4] W M Beynon, Radical Empiricism, Empirical Modelling and the nature of knowing, Pragmatics and Cognition, 13:3, 2005, 615-646
[5] W M Beynon and S B Russ, Redressing the past: liberating computing as an experimental science, Computer Science Research Report 421, January 2006, University of Warwick.
[6] W. M. Beynon and S. B. Russ. The interpretation of states: a new foundation for computation? Computer Science Research Report 207, February 1992, University of Warwick.
[7] D. M. Gabbay. Editorial Preface, *Handbook for Philosophical Logic*, volume 1, 2nd edition, Kluwer Academic Publishers, 2001.
[8] David Gooding. *Experiment and the Making of Meaning*. Kluwer Academic, 1990.
[9] David Harel. Biting the silver bullet: Toward a brighter future for system development. *Computer*, 25(1):8-20, 1992.
[10] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's & the Play-Engin*e. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
[11] P Livingston. Husserl and Shlick on the logical form of experience. *Synthese*, 132:239-272, 2002.
[12] Willard McCarty, *Humanities Computing*, Palgrave MacMillan, September 2005
[13] Loomes, M. J. & Nehaniv, C. L (2001). Fact and Artifact: Reification and Drift in the History and Growth of Interactive Software Systems, *Proceedings of the Fourth International Conference on Cognitive Technology: Instruments of Mind*, Springer Lecture Notes in Computer Science, vol. 2117, 25-39
[14] W McCarty, W M Beynon, S B Russ, Human Computing: Modelling with Meaning *in ACH/ALLC 2005 Conference Abstracts*, Humanities Computing and Media Centre, University of Victoria, 138-145
[15] S. Papert. *Mindstorms: Children, computers and powerful ideas.* Basic Books, New York, 1980.
[16] Andrew L. Russell. "Rough Consensus and Running Code" and the Internet-OSI Standards War. *IEEE Annals in the History of Computing*, 2005. (Forthcoming).
[17] William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438-440, 1982.
[18] W. M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.
[19] Grand Challenge 6: Dependable Systems Evolution `http://www.fmnet.info/gc6`
[20] The Empirical Modelling website at: `http://www.dcs.warwick.ac.uk/modelling`