The Interpretation of States: a New Foundation for Computation?

W M Beynon, S B Russ

Dept of Computer Science, University of Warwick, Coventry CV4 7AL

Abstract

Computation is a central concept in computer science. The mathematical theory of computation, with its historical origins that predate the computer, has been most influential in the development of programming languages, tools and methodologies. The study of psychological aspects of computation is by comparison a new concern that has developed largely as a *post hoc* investigation of contemporary methods of programming; it has yet to make a serious impact upon computer science theory.

In this paper, we shall argue that concerns about perception and interpretation that are significant in psychology should be a primary source of insight into the new models of computation that will shape future developments in computer programming. To this end, we reconsider fundamental issues concerning the nature of computation and its relationship to mathematics and cognition. In particular, we shall explain how computation has to be understood with reference to

- physical processes that are perceived to undergo changes of state
- conceptual methods for interpreting state changes in computational terms.

Brief references to our work on agent-oriented programming using definition-based representations of state will be used to explain our thesis.

Introduction

Understanding computation is a fundamental concern for computer science.

The work of logicians such as Turing, Post, Curry and Herbrand has made a major contribution to the theory of computation. We know what it means for a function to be computable. We have developed programming languages and techniques to implement computable functions. We can understand many programs both in abstract mathematical (declarative) terms and in concrete operational (procedural) terms. We have powerful methods for transforming functional and logical abstractions into executable programs.

Other aspects of computation are not so well understood. When specifying the user-interface, describing interaction and concurrency, representing program design or developing complex electronic / mechanical systems, we view computation in a different way. In these contexts, human interpretation of computation is at least as important as automatic execution. The way in which computational state is communicated – whether to the programmer, the user or to computational agents within the system itself – becomes significant. It is then harder to make effective use of non-procedural abstractions

based on conventional mathematical variables.

This paper describes our progress towards developing a more general theory of computation to suit the needs of modern computer science. The principal source of ideas for this theory is a programme of research into a new method of programming that is agent-oriented and depends upon the use of scripts of definitions for the representation of computational state. Detailed discussion of this research programme is beyond the scope of this paper; for more background, the reader may refer to [3].

Our presentation is influenced by a recent paper of Harel [10] to be discussed in §1. Harel draws a parallel between one-person programming and reactive systems[1]. His paper argues that the incremental developments that so transformed the process of writing correct and efficient one-person programs in the period 1950–1975 are a pointer to how the challenges of engineering reactive systems will be met in the future. Harel outlines a "vanilla" approach to reactive systems development that presumes no radical change to our existing theoretical framework.

Harel's paper is a good point of reference from which to view our proposals. We aim to be more discriminating than Harel in explaining why – and to what extent – "the werewolves of one-person programs have gone, never to return". In our view, the programming problems we have resolved since 1950 are primarily those within the scope of the conventional mathematical theory of computation, whilst the difficulties of reactive systems engineering stem from the limitations of the present theoretical framework. On this basis, we seek an approach to reactive systems development based upon more general foundational principles.

A second influence is a paper by Smith [18], in which he argues – contrary to the Logicist school of thought in AI – that variants of traditional logic[2] are an inappropriate basis for knowledge representation. Smith's concern is expressed in terms of the relationship between the *first* and *second* factors of formal models. For programming systems, the first factor is associated with program execution and the traditional theory of computation. The second factor is associated with program interpretation; it has yet to be satisfactorily formalised. Relating first and second factors is crucial in requirements analysis for reactive systems. This corroborates our view that alternative foundations are needed for a satisfactory account of these and other issues encountered in modern programming practice.

The overall plan and issues in the paper are as follows

- How does the traditional mathematical theory of computation influence computer programming in modern applications? (§'s 1.1, 1.2, 1.3))
- What are the problems, and how can we address them? (§'s 1.4, 2.1, 2.2, 2.3)
- How can we reconcile the different views of computation? (§2.4)

---

1. the term reactive system was introduced by Pnueli to describe a class of embedded, concurrent and real-time systems. Such systems are not data intensive, but have a complex interactive behaviour.

2. following Smith [], we use the term "variant of traditional logic" in a broad sense to embrace the exotic forms of modal and non-monotonic logics that have been proposed in AI. For us, the factor that characterises conventional foundations is the presumption that (at some level of abstraction) mathematical variables with static interpretations are invoked (cf []).

1. Modern computer science and the classical theory of computation

1.1. Programming and the theory of computation

In [10], Harel makes an informal distinction between two kinds of programming activity: one-person programming and reactive systems engineering. In this paper, we examine the relationship between one-person programming and reactive systems engineering from a different perspective. We shall argue that:

- one-person programming and reactive systems engineering are tasks of the same essential nature
- both involve an interaction between two aspects of programming, the first and second factors in the sense of Smith [18]
- the second-factor is relatively much more important in reactive systems engineering than in one-person programming
- the ease of a programming task depends largely upon the extent to which we can separate first and second factor concerns.

We also propose fundamental concepts for a theory of computation that takes account of the second factor.

1.2. The first factor of programming

Specifying a program has two aspects. A program has to be executable. It also has to be interpretable. Paraphrasing Smith [18], we shall regard "what must be directly realised in a physical substrate if a program is to do any work" as defining the first factor of a program and "what the symbols in the program mean, what they're about" as defining the second factor. An adequate theory of computation must allow a high degree of interaction between these two factors.

The term "programming" is primarily used with reference to the first-factor, as in the dictionary definition of a program [9]: "the sequence of actions to be performed by an electronic computer in dealing with data of a certain kind". This definition adequately characterises the first-factor component of one-person programs. A reactive system can also be interpreted as a program, subject to substituting "every agent that can potentially be programmed to change the state of the system" (be it a mechanical, electronic or human agent) for "electronic computer".

The mathematical theory of computation provides a rigorous concept of "program" that captures the notion of executability in a most satisfactory manner. This theory formally identifies programming with the solution of algorithmic problems. Such problems take the form: given an input of a certain class, compute an output that is a pre-defined function of this input. The Turing machine was conceived with computational problems of this nature in mind.

Programming a Turing machine to solve an algorithmic problem supplies the paradigm for all forms of computer execution. The programmer conceives how a functional relationship can be encoded in terms of input and output states of the computer and devises a computer program to define the appropriate transformation from input to output states. When the program is used, the user specifies the input state, the computer exe-

cutes the program and the user interprets the output state.

In computing, the study of algorithmic problems has had a central role. Historically, computers developed from devices for performing special-purpose computations, where calculation was performed mechanically without user intervention. The batch processing paradigm (preparing a program and data, presenting input to the computer, then consulting the output on termination) reinforced a computational stereotype well-matched to the classical mathematical model. Essential concepts such as computability and feasibility necessarily had to be defined with reference to algorithmic problems.

Semantic models for computer programs are generalisations of the algorithmic problem solving model. To describe a program formally is to preconceive its execution to such an extent that its behaviour can be described declaratively, that is, using only non-procedural mathematical variables [7]. Representing ever richer forms of preconceived knowledge has been an important influence behind the development of standard computational models and languages for practical use, and of mathematically sophisticated theoretical models. With ingenuity, functional relationships between input and output can be interpreted as programs with preconceived patterns of interaction (as in e.g. the pure functional programming language Haskell [12] §7). Similar development have occurred in AI, where logicism aspires to encapsulate knowledge prior to the execution of a plan [14].

Crucially, it is the conventions that define formality as it is generally understood that limit the scope of such generalisations. Constructing a reactive system involves a complex process of requirements analysis that must be computer-assisted, yet cannot be formalised in a conventional formal framework. Extensions of classical logic and declarative programming may in principle extend the range of programmed activity that can be preconceived, but interaction between interpretation and execution is essential in computer-aided design. And, as Smith remarks [18], such interaction between first and second factors is inconsistent with what most theorists deem to be "formal".

1.3. The second factor of programming

The second factor of programming is concerned with interpretability, with “what the symbols in the program mean”. As Smith observes in [18], this is curiously not what is referred to in computer science as the semantics of the program: “the only factor of computational systems that [theoretical] computer science talks about is the first”. Formal techniques for program construction reflect this emphasis; they produce executable programs, but typically leave second factor concerns to the user's imagination.

Conventional formal specification does not bind the structure of the model to its second-factor interpretation; all programs with a particular behaviour are regarded as equivalent. By way of illustration, a functional specification of a program to compute an integer function f(n) neither distinguishes between one evaluation and another derived by first incrementing then decrementing n ten times, nor specifies

- how the value of n is communicated to the machine
- how the value of f(n) is presented to the user
- how many arithmetic operations are required for evaluation.

Nor does the context in which the computation is to be carried out have an influence;

the fact that e.g. f(n) is the profit from the sale of n items is a separate concern.

Informally, the situation is quite different. In all specification activity, descriptive identifiers are typically used to suggest interpretations. In programming as most commonly applied in practice, the design of programs is based on data-structures that reflect the external interpretation, as when developing abstract data types or object-oriented models. An important reason for establishing informal connections between the requirement and the form of the specification is that requirements change. Prescribing the correct behaviour is no more important than constructing a specification that can be modified easily to reflect new requirements.

More discriminating techniques for modelling computation are not merely important from the programmer's perspective. Specific operational aspects of programs have to be considered in a theory of complexity or feasibility. Interactive programs have to be evaluated with respect to how internal behaviour is communicated to the user. In general, wherever two or more computational agents interact, as in a concurrent system, there is a need to specify how one agent is to interpret the computation performed by others.

Second-factor aspects of programming are closely associated with procedural activity. In interpreting a program in the process of design, we are concerned with a dynamic object that is subject to change in ways that can not be preconceived. In modelling a concurrent system, we need to represent those intermediate states of computations that are to be interpreted by the computational agents. Conventional formal methods can represent procedural activity only after it has been analysed to such an extent that it can be preconceived. As Smith argues [18], this is not a satisfactory basis on which to "formalise" present programming practice.

The spreadsheet is a particularly interesting example of interaction between first and second-factor concerns. The values associated with cells in a spreadsheet typically correspond to observations that can be made of an independent physical system, such as the profits and costs in a financial model. When the cells representing particular parameters of the model have been assigned values, the spreadsheet can be interpreted as specifying a state of an external system. By changing these parameters, we can predict the effect of changing parameters on the system state.

The principles we propose to apply in "formalising" the second-factor can be derived by generalising from the spreadsheet. Two features of spreadsheets are significant in this context:

- the intuition upon which the interpretation of the spreadsheet depends is procedural in nature; the values of cells in the spreadsheet designate observed values of external variables that can attain many values, unlike mathematical variables
- the functional dependencies in the spreadsheet directly reflect observation of the external system it models; the correspondence between predictions from the spreadsheet model and observed relations between parameters in the external system can be tested by experiment.

1.4. One-person programs vs reactive systems

The discussion of first and second factors points to issues that discriminate between one-person programming and reactive systems engineering. The present theoretical foundations of computing are only sufficient to simplify one aspect of the programming

task: specifying the first-factor operational interpretation. This is only part of programming in practice: it is also necessary to establish the conventions that specify the relationship between a program and its context. In practice, this knowledge representation task, associated with analysis of the requirement, has to be accomplished without the benefit of adequate foundational principles.

Many one-person programming problems can be resolved by dealing with all issues of external interpretation off-line. When the requirements analysis for a program is simple and the physical substrate in which the program is to be realised is well-understood, specification methods based on conventional mathematical foundations are readily applied. These conditions are rarely fulfilled for reactive systems; interaction between first and second factors has an essential role. This analysis suggests that Harel's optimism about the "vanilla" approach is unjustified; there are reasons to believe that new mathematical principles are needed to overcome the challenges of reactive systems.

Harel's paper itself bears out this analysis in many respects. His reference to the progress that was made in one-person programming in the period 1950-1975 is significant, since this is precisely the period over which practical declarative programming systems evolved. There have been many practical developments in software and hardware since 1975, but none that has bridged theory and practice in the same way as functional pogramming. The division in AI between logicism and unprincipled programming to which McDermott refers [14] has its counterpart in software engineering; it centres upon the potential scope of classical formal methods in software development.

Harel's "vanilla" approach entails the development of a conceptual model of a reactive system that is successively refined into a prescription for the components of the physical system. Harel stresses the importance of a mathematical model of the behaviour as a means of ensuring executability of models. In other respects, second-factor issues – the need for visualisation, the role of testing, the importance of a state-based behavioural model, are the dominant concerns. In effect, conventional mathematical semantics is invoked to guarantee an unambiguous operational behaviour, but there are no comparable principles to guide the engineer in matching system behaviour to observation or intention.

The value of a conceptual model for a reactive system depends on many factors. Smith's thesis indicates that models based upon conventional mathematical foundations are inappropriate for this knowledge representation role. Harel's account gives little insight into how good conceptual models can be devised. The engineer needs to relate observation and conception of the system and the conceptual model e.g. to enhance the model to cope with new constraints, to predict the effect of failures in particular components, or to diagnose which components are faulty after the physical system has been constructed. In this context, the need for guidelines for refinement is a crucial concern. The merits of Harel's conceptual model may derive as much from unrecognised peculiar qualities of his chosen visual formalism (i.e. statecharts) as from the "vanilla" approach *per se* – an idea for which further evidence is adduced below.

2. What is computation?

Connecting observation of physical systems with a conceptual model is the central problem of reactive systems engineering. To understand how this might be done in a principled manner, we must re-examine the fundamental ingredients of the computational process itself.

Traditional one-person programming is concerned with sequential interaction with a conventional computer. In such programming, basic characteristics of the computer as a computational device are taken for granted. The computer can be instructed to change its internal state and the state of its environment (e.g. through changing the screen display) in a reliable and replicable manner. The conventions by which these changes of state are to be interpreted by the user have already been fixed at some level of abstraction. For instance, the computer may be able to display strings of text, or draw families of points and lines.

Observation of the externally accessible state is simplified by the conventions for interaction and interpretation. The user adapts to the approximate nature of the screen display e.g. to extract ideal points and lines from patterns of pixels. In understanding physical systems through scientific experiment, simultaneity of observations is centrally important; in sequential interaction with the computer, indivisible changes in the observed state are contrived by restricting the points of entry for the user, whether as observer or experimentor. For instance, the computer may take several minutes to generate a line drawing in an incremental fashion, but the user is not intended to interpret the drawing until it is complete.

One-person programming involves two kinds of state-based activity. There are the changes to the internal state of the physical computer that are specified in a first-factor account of the program. At some level of abstraction, how these changes occur is beyond the programmer's concern; what matters to the programmer is that they do occur reliably. The internal state changes have the characteristics of primitive machine operations: they are indivisible and can neither be observed or interrupted in execution. What the programmer deems to be an indivisible machine operation is a matter of discretion, depending upon how closely the computational activity has to be observed.

The other kind of state changes involved in one-person programming are those that communicate internal state to the user and allow the user to change the internal state. The nature of these state changes is influenced by second-factor issues – i.e. concerning the relation between the program and the external environment – that are hard to describe formally. These include:

- the physical capabilities of the computational device: how many pixels there are, whether it can generate sound, whether there is a mouse or a windowing system

- the sensory capabilities of the user: whether the user is blind or colour-blind, whether the user can handle the keyboard or control the mouse

- the knowledge that the user brings to bear in interpreting the state-changes on the screen: what conventions for interaction are assumed (e.g. the mouse must be in the window before you type), and for interpretation (e.g. the string "324.3" denotes a specific decimal number).

In practical programming, the informal nature of the relation between first and second-factor aspects of a program is frequently exploited. In debugging a program, the programmer typically adapts the interface so that the internal state of the computer becomes apparent. If a computer program for drawing a picture has an error, the user may be able to detect this by observing how the display is generated, despite the fact that this generation process depends upon the graphical interface rather than the program. A the-

oretical framework that does justice to practice should account for such abuses. Conventional formal specification techniques for generating a program with a specified IO behaviour typically limit the scope for investigating second-factor concerns by restricting what the programmer can know about program execution.

Reactive systems force us to examine the relation between state change in physical systems and its computational interpretation more closely. There are many agents that can act concurrently to change the system state and these can be of several different kinds: human, electronic or mechanical. The manner in which agents react to changes in state is not constrained by simple protocols for sequential interaction; simultaneity of observations and indivisibility of actions must be explicitly considered. Principled methods of interrelating first and second-factor aspects are needed to replace the informal conventions that operate in one-person programming.

Computations performed on unconventional computers are instructive in exposing issues that conventions typically hide. Children are sometimes taught to multiply the single digit n by 9 by displaying all 10 fingers and turning down the n-th counting from left to right. This separates the fingers into two classes: the number of fingers on the left and the right are the two digits in the decimal representation of the answer. In this context, the fingers themselves define a computational device with a trivial state-changing behaviour. "Turning a finger down" *is* multiplying by 9. The potential subtlety of interaction between computational agents is vividly illustrated by analysing what implicit knowledge and capability is expected of the child.

As another example, consider a folk-dance routine for computing the greatest common divisor:

```
To calculate GCD(m,n), take m woman and n men
NB m and n must be positive

Match up men and women in pairs
        until no more pairs can be formed
If everybody has a partner
        stop the dance and count the number of pairs
        [This'll be GCD(m,n)]

Otherwise
        there's either a spare man or a spare woman
        NB of course there may be more than one!
If there's a spare man:
        send the men with partners out of the room
        NB without their partners
If there's a spare woman:
        send the women with partners out of the room
        NB without their partners

Repeat the dance, forming new pairs etc
```

In this mode of computation, the role of particular observations of the physical system in distinguishing relevant state changes is evident. Within reason, we are not concerned with what partners do over and above carrying out the instructions specified in the dance e.g. we assume that they do nothing to generate more men or women. There are certain hidden assumptions about how the dance instructions are to be interpreted. The

physical models poses implicit constraints on the usefulness of the program: e.g. there is an absolute bound on the number of women and men that can be involved and there are feasibility constraints upon synchronisation and evaluation.

State change is all around us, but not all this can be construed as computation. To interpret the behaviour of a physical system in computational terms, we must first identify patterns of state change that occur reliably. This process primarily involves conceiving appropriate observations of the system, specified by protocols for measurement such as are required in experimental science. This observation is the basis for interpretation in the second-factor sense. We shall argue that, in interpreting the behaviour of system X (e.g. a Macintosh computer) as computation, we are appealing to an exact correspondence between observations of system X and of an independently specified physical system Y. By this thesis, computation is defined by a correspondence between two sets of experimental observations: the set of observations of X that define the *program model* and those of Y that define the *reference model*. Notice that the term program model encompasses observations that are typically regarded as external to programmable components; for instance, it includes relevant observations of input-output devices, such as the position of the hands of a clock.

How is the "times 9" computation to be understood in these terms? Self-evident as it appears, the fact that whenever the 3rd finger is turned down there are 2 fingers to its left and 7 to its right must be regarded as an experimental observation in which we can place great faith, but for which we have no further explanation. The protocols for observation include "counting fingers": it must be presumed that a child can count up to 7 fingers before its fingers are required for a second multiplication (or other less educational activity). The answer is 27, and this is the same answer that we could derive from an independent activity: collating 3 sets of 9 objects and counting the total number of objects. To be more pedantic, we might say that the total number of objects can be separated into two groups of 10 with 7 remaining.

Analysing computation in this depth is a way of revealing an essential similarity between reactive systems engineering and one-person programming. Both involve establishing a correspondence between a program model that describes the behaviour of the physical computing system and how it is to be observed and a reference model that describes what we expect of these observations. The difference between the two activities lies in the status of these models.

In one-person programming much of the program model is preconceived and is prespecified by convention, whilst the reference model is typically of such a familiar form that it can be specified in declarative terms, as a functional relationship for which an appropriate input-output state-based observation model can be readily conceived. Multiplying by 9 is an example where this applies. The reference model takes other forms when non-functional requirements are involved.

In reactive systems specification, much of the program model has to be constructed from first principles on the basis of particular characteristics of the physical components, whilst the reference model is typically defined in the early stages of development by behavioural fragments or *scenarios*. The program model and the reference model are developed in parallel; to do this in a principled manner, it will be necessary to describe how observations of the models correspond. The use of the spreadsheet principle in formulating a simple correspondence of this nature has been described above; our objective is to generalise this principle to multi-agent systems.

3. Reactive systems specification: a principled approach

3.1. The development process overview

Observation is the fundamental concept in our view of computation. We also believe it to be fundamental to a principled approach to the second factor aspects of programming for reactive systems. The programming principles for realising input-output relations based on the conventional theory of computation deal with a particular kind of state-changing activity where only the initial and final observations are significant. A more general theory of computation has to account for many other kinds of observation.

Our general approach to reactive system specification will first be described without reference to the role of observation. This account will hide the characteristic features that distinguish our program model from a "vanilla" conceptual model, such as is described in Harel [10]. As explained in more detail in [3], the principles that guide the process of construction put special constraints upon our program model. This is what we should expect if the model is to reflect second-factor concerns (cf [18]).

Figure 1 depicts the relation between models that is involved in our view of reactive systems specification. R is the reference model that describes the observations we would ideally like to make. C is the program model. When C has reached its final form, it is a model that reflects the actual capabilities to receive and respond to stimuli of the components of the physical system at an appropriate level of abstraction and to an appropriate degree of accuracy.

Three types of activity are involved in the process of constructing the reactive system S:

1 the programmer conceives or modifies the "reference model" R that defines the total behaviour to be realised in terms of desired observations of the system S

2 programmer develops a program model C on the computer that represents the behaviour of the physical system in terms of observations pertaining to those components of the system that have currently been conceived

3 the components of the system S are built by using the model C prescriptively to determine their physical characteristics and how they are programmed to act

The program model C resembles an conventional engineering plan for an object to be built (cf a design drawing). Rather than being a set of documents, it is a program with which the designer can interact to simulate possible state changes in the reactive system as so far conceived. The most important characteristic of this model is its close correspondence to the physical system as it is conceived at each stage of development. The representations of state in the program model will involve variables whose values are represented to the user visually and that can be interpreted as observations. The correspondence between spreadsheet variables and observations of a physical system is the archetype.

The reference model is the most informally specified; it is intended to represent the requirement in all its aspects. It will typically include particular scenarios for system response; perhaps more formally specified abstract specifications of behaviour, and aesthetic preferences on the part of the designer that can only be expressed through in-

teraction with the program model. The model may be influenced during the development process both by interaction with the program model and by the results of experimenting with possible components of the system S.

1, 2 and 3 are not to be viewed as phases of the development, but as modes; interaction between the three activities is to be expected. In its final form, the program model describes the physical devices of the system with sufficient accuracy to be used prescriptively; it is not an abstract description of system behaviour that is unrelated to the engineered system. In conventional one-person programming, activity in modes 1 and 2 is meta-level activity that is often performed off-line, whilst 3 corresponds to the construction of the program itself.

3.2. Linking specification to observation

There are three aspects to our work on reactive systems specification outlined above:

- practical studies in prototyping using the methods
- concepts and techniques that have been developed for this purpose
- foundational issues: what – if any – are the principles behind our approach.

In the context of this paper, as indeed for our program of research as a whole, it is the third of these aspects that we regard as most important. The reasons for this emphasis are clarified in [3]. The history of object-oriented programming in particular suggests that expressive modelling techniques for addressing first and second-factor concerns independently are not enough to overcome the complex problems of reactive system engineering. Ideally, a prescription for new programming methods to address these problems should be supported by proposals for adequate theoretical foundations. This section briefly summarises the experience that has informed our ideas about reactive systems engineering and that best illuminates potential foundations.

The programming techniques that provide the basis for our work are:

- definitive (definition-based) state-transition models that exploit a wide-ranging generalisation of spreadsheet principles
- agent-oriented models, over a language LSD in which to specify the values bound to agents, those they can observe, those they can conditionally redefine, and the indivisible functional dependencies associated with their actions.

The practical advantages and characteristic features of these particular techniques has been well-documented elsewhere [3,4,5,6,7]. They have proved particularly useful in describing interaction and visualisation.

Our research so far has included several case studies on concurrent systems simulation. Published specifications include simulations of telephone operation, of an electronic cat-flap, and of a protocol for the arrival and departure of a train. Most recently, we have developed animations of a vehicle cruise control system and of Harel's digital watch together with its accompanying statechart [10]. Of particular interest is the fact that the states in statecharts can be related to persistence of LSD agents.

In the context of this paper, the most interesting aspect of definition state-transition models and agents is that they are intimately related to observation of physical systems. A spreadsheet expresses the relation between real-world observations in a manner that is outside the scope of pure declarative models. The process of developing a program model for a reactive system described above is motivated by the desire to provide an environment for the designer that precisely captures observation of a complex physical system as the spreadsheet does.

Observations have a fundamental role in guiding the construction of the program model:

- the program model is defined by observations of reliable state-changing devices
  The programmer has to conceive state-changing activities that are as reliable and replicable as machine operations on a conventional computer. Only physical objects that exhibit such behaviour can serve as computational agents. There is a role for experiments to determine appropriate characteristics of physical objects, analogous to the tests of materials that an engineer might perform.

- observations determine the communication of state-changes between agents
  The programmer has to identify what each agent observes of the state of its environment and how these observations change indivisibly in that agent's view when actions are performed. The anthropomorphic view of agents reflects the need to represent system activities through the eyes of the programmer.

- the process of interpretation involves matching of experimental observations
  The programmer must conceive the relation between the program model and the reference model.

A theory of observation is a plausible basis for a principled method of constructing a program model for a reactive system. In such a theory the programmer's task is seen as identifying the observations of a physical system that must be analysed and prescribed in order to guarantee a particular behaviour. The following principles of observation lead us to think that the abstractions behind definitive state-transition models and agents are of fundamental interest. In a physical experiment, once the context and conventions for observation and action have been specified, experiment alone determines:

- whether observations are indivisibly linked through functional dependency

- what groups of observations are subject to change independently.

This provides a physical basis on which to specify definitive scripts and agents.

Conclusion

This paper proposes a new perspective on computation and programming. Computation is viewed as interpreted state-change that can be associated with physical systems of diverse kinds. In this framework, the problems of one-person programming and reactive systems can be regarded as having essentially the same nature. General principles that govern observations of physical systems are a plausible basis for a theory of computation that can deal effectively with both kinds of programming task.

References

1. P America *OOP: a theoretician's introduction* EATCS Bull 29, 1986, 69-84
2. J Backus *Can programming be liberated from the Von Neumann style?* CACM 21(8), pp.613-641, 1978
3. W M Beynon *Programming principles for the semantics of semantics of programs* Computer Science RR#205, Warwick University 1992
4. W. M. Beynon, S. B. Russ, M D Slade, Y P Yung *Programming as modelling: new concepts & techniques* Proc ISLIP'90, Queen's Univ. Kingston, 1990
5. W M Beynon, M T Norris, R A Orr, M D Slade *Definitive specification of concurrent systems* Proc UKIT'90, IEE Conf Publications 316, 1990, 52-57
6. W M Beynon, Y P Yung *Definitive Interfaces as a Visualisation Mechanism* Proc GI'90, Canadian Inf Proc Soc, 1990, 285-292
7. W M Beynon, S B Russ *The development and use of variables in mathematics and computer science* IMA Conf Series **30**, OUP, 1991
8. G Birtwistle, O-J Dahl, B Myrhaug, K Nygaard *Simula Begin* 2nd ed., Studentliteratur, Lund, Sweden, 1979
9. Chambers 20th Century Dictionary, 1972 edition
10. D Harel *Biting the Silver Bullet: Towards a Brighter Future for System Development* IEEE Computer (to appear Jan 1992)
11. C A R Hoare *Communicating Sequential Processes* Prentice-Hall Int. 1984
12. P Hudak et al *Haskell: A Non-strict, Purely Functional Language* Report Version 1.1, Aug 1991
13. W Kent *Data and Reality* North-Holland, 1978
14. D McDermott *A critique of pure reason* Comput Intell **3** (1987) 151-160
15. D Parnas *On the Criteria to be used in Decomposing Systems* CACM **15** (1972), 1053-1058
16. A Pnueli *Applications of Temporal Logic to the Specification and Verification of Reactive Systems* LNCS 224, Springer-Verlag 1986, 510-584
17. B. C. Smith *The owl and the electric encyclopedia* A I **47** (1991) 251-288
18. B. C. Smith *Two lessons of logic* Comput Intell **3** (1987) 214-218
19. W. A. Woods *Don't blame the tool* ibid, 228-237

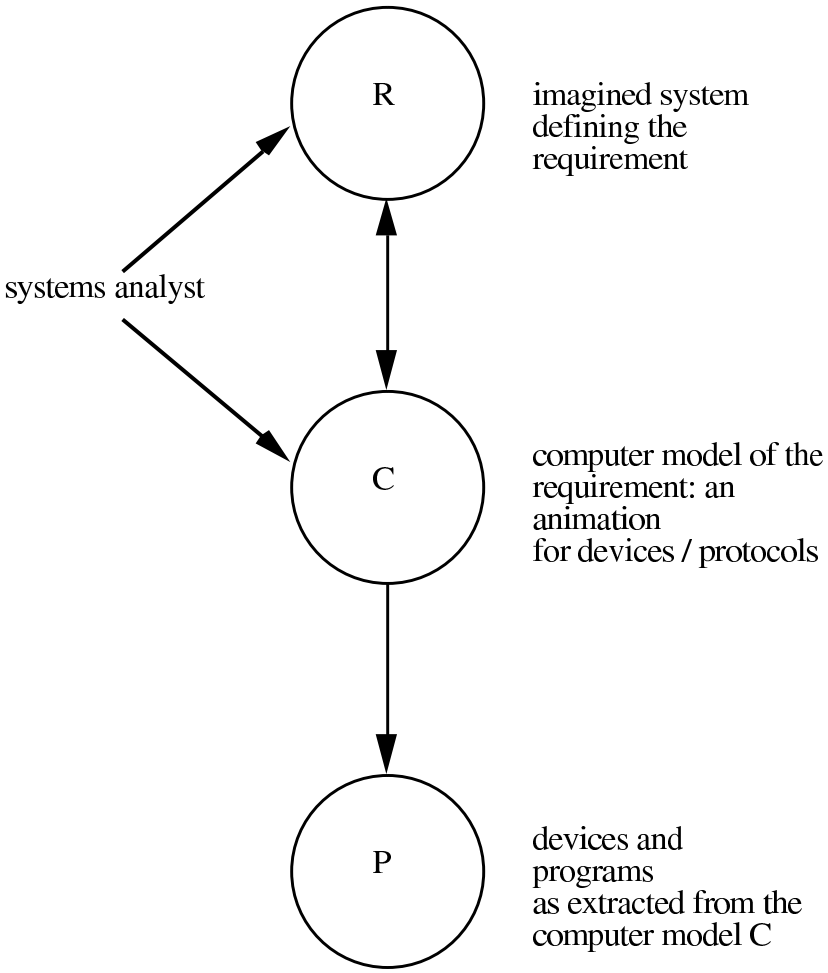

Figure 1: Modelling in Requirements and Testing for a Reactive System