Empirical Modelling of Requirements

Meurig Beynon *Steve Russ*

{*wmb,sbr*}*@dcs.warwick.ac.uk*

Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK

September 1994

Abstract: We propose a modelling process, based on observation and experiment, which is well-suited to reactive systems, and which provides an integrated environment for the requirements, specification and design phases of a development. The modelling method depends on application-specific knowledge which can be modified on-line by the intervention of the modeller. The process offers immediate experience of the model behaviour, allows for the concurrent refinement of a requirement according to multiple viewpoints, and assists in the decomposition of a system requirement into component requirements. We present arguments for the principles of the process, and illustrate the application of the process with extracts from a vehicle cruise control model built using our methods.

Keywords: requirements modelling, requirements specification, spreadsheets, observations, experiments, agent-oriented modelling, agent-oriented programming, concurrent design, reactive systems, complex systems.

1. Requirements and Programming for Reactive Systems

1.1 The Challenge for Requirements from Reactive Systems

The earliest requirements for the stored-program electronic computer were for well-established tasks in areas such as code-breaking, ballistics modelling, numerical analysis etc. These were tasks which had previously been carried out by skilled human operators with the aid of mechanical or electromagnetic devices. The new computing machines were essentially replacements for previous devices, and some of the skills of their operators. In such cases it is known in advance that the requirement is satisfiable. The programming to meet such requirements would be equivalent today to the construction of small programs in a high level language, and is what we shall call 'one-person programming', using the phrase introduced by Harel in [11]. In many cases the programming called for by these requirements was therefore relatively straightforward. However, requirements soon

escalated, programs became 'software systems', and the complexity of both provoked the comment by Brooks in 1986 that, "The hardest single part of building a software system is deciding precisely what to build" (p.17 of [6]).

Since then the challenge for requirements has become even more formidable. Dramatic advances in technology have led to the availability of systems not only of unprecedented power but also of a radically new functionality. These are complex systems comprising distributed software and hardware components and offering facilities inconceivable, or infeasible, prior to the advent of electronic computing devices. As well as providing new methods of meeting existing requirements, these systems are inviting entirely new requirements. Often these are tentative and provisional requirements in the face of the uncertainty as to how this new range of computing systems can best be used. Some such requirements may be unsatisfiable for practical or physical reasons. A particular kind of complex system, so-called *reactive systems*, are embedded, concurrent, real-time systems; these are described by Harel in [11] as "posing some of the greatest challenges in systems engineering", and he concludes that whatever general framework is eventually adopted for system modelling, "reactive behaviour will be one of its most crucial and delicate components". We believe that some of the deepest challenges of reactive systems are closely related to the effective management of their requirements. In the same paper Harel maintains that in spite of numerous "methodologies" for development, "The availability of a solid, general-purpose framework within which one can conceptualize, capture, and represent a system model seems to be far more important right now". We shall show in this paper how our proposal for such a framework also addresses the challenge of requirements in a powerful and flexible fashion.

1.2 Motivation and Objectives of this paper

The well-known problems of the conventional development cycle of requirement, specification, design, program and testing, are exacerbated for reactive systems. The behaviours of such systems are rich in observational and temporal patterns and can be exceptionally complex, therefore their initial description, in either a requirement or a specification, is likely to be imperfect. But machine-oriented specification usually excludes any subsequent influence from real world experience, and so detailed knowledge of the imperfections may be long delayed, perhaps until testing. Prototyping can never be rapid enough. Consequently our objectives here are to present a new modelling process well-suited to reactive systems, and to show that it:

- integrates the requirement, specification and design phases of a development;
- allows for on-line intervention in a way that imitates immediate experience of the world;
- allows a provisional requirement to be elaborated and revised from different viewpoints concurrently;
- assists in the decomposition of a system requirement into requirements for the system components.

We shall illustrate our general approach, and each of these objectives, with examples from a vehicle cruise control simulation built using this modelling process.

1.3 Generalised Programming for Complex Systems

Our approach to modelling, which is described in the next section, can best be understood as complementary to a generalised, behavioural view of programming which has motivated our

research for some years. On this view, programming at a behavioural level, actually *is* modelling of our particular kind, when the latter is viewed prescriptively rather than descriptively. But this behavioural view of programming is not one which abstracts away from experience. In our approach both programming and modelling can well be described as *empirical* because they are explicitly based on observation and experiment. In this section we shall outline this view of programming by considering the similarities between one-person programming and the development of reactive systems. (In the previous section we highlighted their differences with regard to requirements.) It will be helpful in our discussion to make reference to three particular viewpoints which will recur prominently in our main example of a vehicle cruise control system in §2.3, these are:

- what the system is going to do (user's viewpoint);
- constraints on the context for operation (analyst's viewpoint);
- the resources or devices that are available (implementer's viewpoint).

We begin by asking how we might regard one-person programming as a 'system'; this will lead to seeing how the development of reactive systems can naturally be viewed as programming.

At an abstract level a software system has much in common with general engineering systems, for example, it involves: the formulation of requirements and their satisfaction subject to limited resources, problems of the design and decomposition of systems, issues of efficiency and maintenance of operation, etc. As a general context for both one-person programming and reactive systems we shall therefore adopt a system comprising several components, each with some capacity to respond to its environment and to initiate, or record, changes of system state. In one-person programming the components of such a 'system' are the user and a conventional, purpose-built computer. This context is so familiar that important and ever-present features of it are easily overlooked. The following two features are especially instructive because they cannot be assumed for reactive systems and yet they are essential for building such systems successfully:

(i) all state-changing components (including the user) need to be 'programmed';

(ii) all state-changing components (including the computer) are presumed to operate reliably.

For obvious reasons we are very conscious of the need to program the computer and of our own ability to respond reliably, but less so of the fact that the user needs to be programmed and that we are presuming the computer to be reliable.

The first feature will lead us to a generalised notion of programming. Consider the elementary task of writing a program for the computer so that the user inputs two numbers and the computer outputs their sum. The program prescribes a sequence of operations for the computer but it cannot be used without a user who knows how to behave appropriately. The user must be able to enter inputs and read the output, and do these things in the right order and at suitable speeds. No matter how 'user-friendly' the interface there must be a significant protocol of action, observation and interpretation for the user: in effect, both user and computer are programmed. The reciprocal nature of the relationship between user and computer is characteristic of the interaction that pervades complex systems. Input is the means by which the user influences the state of the computer, and output the means by which the computer changes the environment of the user. In this context, the user comes to the system both as client (with the requirement) and as essential participant, a component in the system which is designed to meet that requirement.

Thus it is reasonable for an analyst, viewing one-person programming from outside the perspective of the user, to regard the activity as that of a system. To satisfy a given requirement with

such a system means to prescribe appropriate behaviours to each component (user and computer). This leads us to a definition of abstract, or generalised, programming as the prescribing of the behaviour of the state-changing components of a system so as to guarantee an appropriate overall behaviour of the system. In general, there will be a system-component hierarchy. In the case of one-person programming it is clear that each of the components are themselves complex systems. At each level the system requirement needs to be decomposed into the component requirements. The decomposition continues until the components can be readily identified and realised in a physical form. This is the realm of the implementer or engineer.

This generalised perspective on programming, which we have derived from one-person programming, also suits reactive systems very well. The latter may contain many state-changing devices, each capable of generating, and responding to, stimuli in different ways. Thus programming such systems may involve specifying the responsiveness of sensory devices, prescribing the speed of mechanical operations and the characteristics of electronic devices, as well as formulating correct procedures for interaction with possibly many users. A consequence of defining programs in terms of the real-world environment in which they are to operate, is that we can initially eliminate the accidental features of particular computing devices and languages.

The second feature of one-person programming which we mentioned above was the reliability of the computer. Having pointed out that the user also needs to be prescribed, we now draw attention to how little we contribute to the prescription of the computer. The high-level program that we give to the computer is only a fraction of the prescriptions that have been built in by manufacturers in layers of software and hardware which ensure (nearly always) the expected sequences of compilation and execution at appropriate speeds. The user is thus protected from all the vicissitudes of mistimings, mismatched characteristics and unreliability which, in general, may attend the assembly of disparate system components.

Before we can consider the requirements for the components of a reactive system, there may therefore be a problem in even identifying the components, that is, the reliable state-changing components we can use as programmable devices. From the perspective of modelling we shall call such devices or components in a system the *agents*. Typically an agent responds to the stimulus of state change in its environment by making further state-changes according to given protocols. (We are using the language of stimulus and response in relation to agents, in a similar way to that in which Deutsch uses it in relation to objects in [8].) In the simple user-computer system the user's stimulus is generally provided by observations of the screen state. We generalise this notion of observation to refer to any stimulus to an agent of the system. In specifying a reactive system, there is an essential role for on-line experiments to establish the reliability of component, or agent, responses. Reliability is closely connected with the constraints for the context of operation so these will affect the kinds of experiments to be performed. And since synchronisation of actions is crucial, guarantees about component responses must also reflect timing considerations. Thus our analysis suggests that to prescribe system behaviour abstractly we need to have identified all the relevant :

- observations of the system;
- agents that can be active in the system;
- stimulus-response patterns (protocols of agents);
- dependencies between system parameters;
- timing characteristics of agent interactions and reactions.

The determination of the above characteristics calls for systematic experiments with the system concerned, and for judgements on the regularity and reliability of the behaviours observed. This is close to conventional engineering procedures. We first satisfy ourselves about the expected behaviour of each component through experiment and observation, then organise components into a system whose behaviour is guaranteed subject to assumptions about the reliability of components, and the circumscription of the context for operation. In specifying a complex reactive system, it is inconceivable that a designer can identify appropriate prescriptions for agents, and gain confidence in the reliability of their behaviour, without first carrying out experiments with the components of the system. It is generally only realistic to perform these experiments in a virtual setting, that is, a modelling environment, since they may involve devices that have yet to be constructed, and scenarios that may be hard to configure in practice.

In one-person programming the viewpoints of user, analyst and implementer are usually conflated. The user probably knows what they want the system to do, is very familiar with the context for operation, and the only resource is the computer and interface. With reactive systems these three aspects need to be separated and each given careful attention. They may be represented by different people or teams of people, there will generally be conflicts of interest, and detailed communication and negotiation will be necessary. To conduct such interactions powerful methods of modelling are again called for which will embrace both software and hardware components, and allow for continuous intervention and experiment.

User, analyst and implementer are each being regarded here as an application expert. In the example of the vehicle cruise control used throughout §2.3 the user would ideally be an experienced driver, the analyst a person with expert knowledge of the operational environment, and the implementer an automotive engineer. This essential need for application experts illustrates our belief that there can, in principle, be no general 'method' for the development of complex systems. In particular, the relationship between system requirements and the requirements for the components involves deep, application-specific, knowledge. We claim that there are two aspects to a successful development of a complex system: the application-specific knowledge, and the general framework for the representation and use of that knowledge, together with knowledge of the requirement and its context. This general framework is the arena for the modelling process which forms the core of any development, it is this to which we now turn attention.

2. Modelling Requirements for Reactive Systems

2.1 The Principles of the Empirical Modelling Process

The features of our empirical modelling process match closely the characteristics of generalised programming which we have described in the previous section. Our models must begin from some knowledge of the application domain, and reference to a provisional requirement, but they are intended to be built as a prelude to the precise formulation of a specification. They exhibit behaviours that are appropriate to the application but which are not preconceived: changes in behaviour are possible which we had not anticipated, and which could not have been derived in advance. The principles of our modelling process reflect to some extent the way in which we as humans typically perceive, and interact with, the world. The process has four mutually complementary, and inseparable, features:

- it is *state-based*: the current state of the real-world context is always represented;
- it is *agent-oriented*: identification of the state-changing agents active in the system determines the model structure;
- it is *definition-based*, that is, it uses spreadsheet-like definitions to express the dependencies perceived between observations in the world;
- the fundamental entities are *observations*: these are associated with agents, they define the state, and correspond directly to the variables which are the components of the definitions.

In our modelling framework, observables are represented by variables. In observing a behaviour, there are certain indivisible relationships amongst the observables. For instance, the moment at which the minute-hand passes midnight may also herald a new year, or be the moment at which a savings policy matures. Such dependencies are modelled by unidirectional constraints, and expressed using systems of definitions, or *definitive scripts*.

The semantics of a definitive script resembles that of a spreadsheet, in that it typically represents one state in a real world behaviour that is immediately experienced. For instance, in a spreadsheet that represents the current state of a financial account, it is in general impossible to predict the form and effect of the next transaction. The use of definitive scripts as the fundamental method of representing perceived system state is consistent with the idea that directly experienced behaviour is more primitive than circumscribed behaviour. A script is automatically updated exactly like cells defined by formulae in a spreadsheet. The user, or the machine, may make re-definitions to effect a state transition. User intervention in this way corresponds to an on-line 're-programming' of the model. This explains how such models can both be continuously responsive and yet also behave in a way that is not preconceived. Thus our models are constantly open to revision in the light of new experience of the world, and experience (e.g. via visualisation) of the current state of the model. This power to model immediate experience distinguishes our approach from most other computational frameworks, many of which are based solely upon abstractions for describing circumscribed behaviour.

The correspondence between our computer model M, and the real-world system S it represents is unusually close. There is a rich and precise correspondence between the states of M and the states of the system S, so that we can simulate many different modes of interaction with S and its components through interacting with M. This virtue derives in part from the explicit state-based nature of our model. But, more significantly, the association between S and M is based upon the correspondence between observations of S and variables in the model M. As a result, we can simulate the effect of adopting a new perspective on S, of choosing to view parts of S, and interact with S in an unrestricted manner. This will be illustrated in the discussion of the vehicle cruise control system in §2.3.

In modelling the behaviour of a system of agents, the first step is the description of those observables that are bound to an agent (state variable), those that it is conditionally privileged to change (handle), and those to which it responds (oracle). This description is represented using a special-purpose notation known as LSD. In animation from an LSD description we take account of the enabling conditions that must be satisfied before the values of observables can be changed, and the perceived events that serve as stimuli for agent action. The animation is executed in the computational framework of the Abstract Definitive Machine (ADM), an environment in which the modeller can act as a superagent to impose, dynamically, appropriate scenarios for action and

interaction upon agents. The overall development of this modelling process is presented as a general schema in Figure 1.

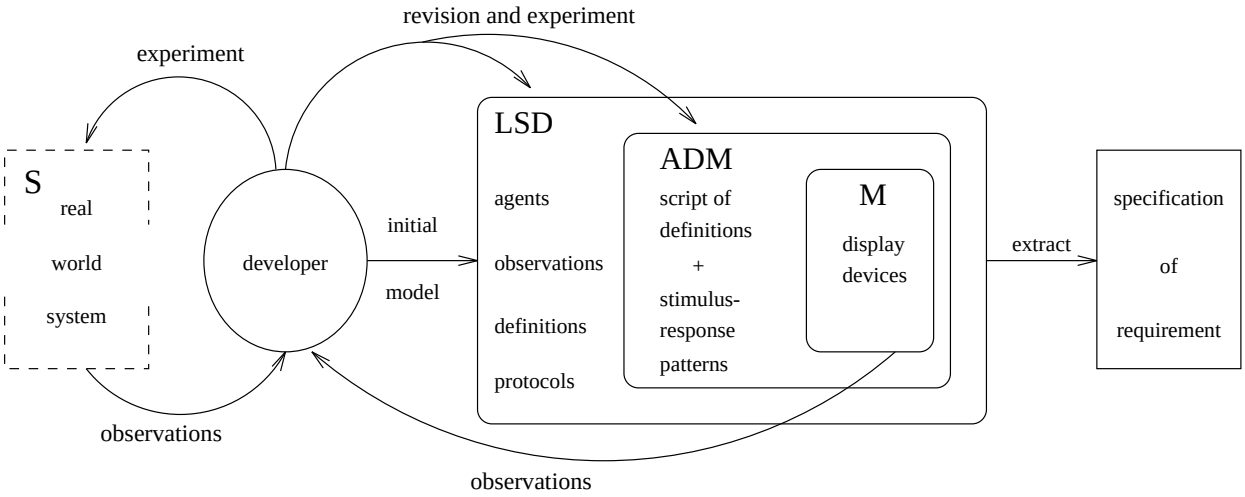

Figure 1: The empirical modelling process

The 'developer' of Figure 1 has to combine the role of an application expert and a modeller. The developer makes an initial model by identifying relevant agents, and the observations and behavioural protocols which are associated with them. This forms a partial description in the LSD notation. When supplemented as described above, this leads to a set of definitions which can be evaluated by the ADM. The resulting 'value' is a state of the model, and is automatically displayed by whatever physical devices are available. Re-definitions entered by the developer, or automatically, allow for animation of the current behaviours of the model. Initially the developer seeks to reproduce identifiable behaviour from the agents described and this is gradually refined by experiment and observations into the required behaviour. These experiments and observations can always be made on the model, but may be restricted in the real world system depending on how much is accessible or in existence. The refinement is made at two levels indicated by the branched arrows at the top of the diagram. Temporary re-definitions are made directly to the ADM, more permanent revisions are made to the LSD script. In principle the extraction of the specification in a textual form, derived from the LSD script, can occur incrementally during the development process as commitments are made. For a fuller discussion of our overall approach to computation and modelling than is possible here see [2],[4].

2.2 Comparison with other Modelling Methods

Many mathematically-based, or formal, modelling methods make an early commitment in which features of the application domain are represented by abstract objects and operations. This necessarily means preconceiving the possible future states of the model; revision in the light of experience requires a new model. Formal models, when they can be constructed in such a way as not to abstract from the essential behaviour of a system, have highly desirable properties, such as the precision and reasoning power which they offer. But the mathematics required to underpin them is only available where aspects of the application are well understood, or can be put under controlled

conditions. Many complex systems which we wish to model are ones drawn from contexts of social interaction, economic and biological systems etc. The mathematics, or scientific theory, adequate to describe such systems in detail does not yet exist, and in many cases may not, in principle, be possible. Even where it does exist, the fact that formal models do not directly reference states in the real world renders them severely limited for the general modelling of reactive systems.

With object-oriented modelling methods the situation is more subtle but there are clear and significant commitments. Following such a method it is necessary to be able to decompose a system into distinct, coherent objects. According to Davis (p. 61 of [7]), "For the purposes of requirements an *object* is defined as:

- a real world entity
- related to the problem domain
- with crisply defined boundaries
- encapsulated along with its attributes and behaviour
- whose behaviour and attributes must be understood in order to understand the problem at hand."

In some applications a decomposition into objects with such properties is no problem, and is subsumed by an observational viewpoint. But the understanding of the behaviour and attributes of objects, to which Davis refers, implies a level of commitment and knowledge about the problem that may not be realistic initially. In general, we believe our emphasis on observations as primitive is much more flexible, and allows for a greater openness to the immediate experience of the real world. For example, the indivisible propagation of state changes which is implicit in our spreadsheet model may need to cross 'object' boundaries. Also, by dealing with observations rather than objects, we can take account of synchronisation issues crucial in the semantics of assemblies and components. By expressing how changes in observations are coupled together – as in a mechanical linkage – we can represent the indivisible operations associated with each computational agent. The distinction between our model and an object-oriented model is best appreciated by considering the design of an entirely new engineering product, where we have first to conceive the functions for sub-assemblies and the observations that are essential in interpreting their cooperative behaviour. In contrast, an object-oriented model presumes sufficient knowledge of the system decomposition and interaction to allow the encapsulation referred to in the quotation above.

Some software development frameworks for modelling concurrent systems are superficially similar to that described in §2.1. For instance, animation plays an important role in systems using Statecharts [10] and Actors [12]. Such methods of animation are used in analysis and design activity following a preliminary system specification. It is less clear to what extent they are used, or could in principle be used, for exploring a requirement in order to develop a system specification. The same comment applies to the application building techniques in [17] which adopt spreadsheet principles similar to ours. The significance of stimulus-response analysis is explored in the context of object-oriented design by Deutsch [8], while active object systems like SAOS [15] have some resemblance to the definitions and protocols in an LSD script.

2.3 Requirements for a Vehicle Cruise Control System: a Case Study

2.3.1 Overview

A vehicle cruise control system has been the focus of several software specification studies [5],[8],[9]. The examples in this section are drawn from a vehicle cruise control simulation

developed within in our framework and which we regularly use for demonstration purposes. The model was not built primarily as a simulation: it was built to explore our modelling methods. The simulation effects are an inevitable by-product of the modelling process in which realistic observation and experiment in the model is the chief instrument for development and refinement. In the following sections we give a condensed description, and a few extracts, to illustrate the application of our method to requirements analysis. Fuller technical details of the modelling involved in this example can be found in [3] and [16].

In keeping with the objectives set out in §1.2, we shall adopt as a provisional requirement: "the vehicle should have a cruise controller, set by the driver, that maintains the set speed under varying load conditions in a manner that is safe, efficient and comfortable". As explained in §1.3, it will be useful here to consider three conceptually distinct 'views' of the modelling process: those of the user (driver), the analyst (who manages the environment and the context for operation), and the implementer (or engineer, who will specify the components of the system). To simplify the exposition we shall assume that each application specialist can act in the role of developer (see Figure 1). The table in Figure 2 summarises the roles of the three developers with reference to their contribution to the overall specification and the real world experience that informs it.

Developer	*Domain of experience*	*Nature of specification*
user	driving and use of a cruise control	user requirement e.g. a safety cutout
analyst	operational environment and physical laws	analyst requirement e.g. operation of cutout
implementer	engineering devices and their characteristics	implementer requirement e.g. devices for cutout

Figure 2

In practice it is important that these views are represented within the same framework, and that they can mutually interact, but it is convenient also to distinguish them for the sake of understanding the progress of the process, and how it can assist in the decomposition of a requirement.

2.3.2 The User's View

The user is concerned with the possible observations and actions of a driver and therefore introduces agents for the driver, the vehicle, and the engine, each with associated observations (e.g. the position of the brake, which is under control of the driver and belongs to the vehicle). The description of the agent 'driver' in the notation LSD is shown in Listing 1. This makes use of a 'protocol' section that consists of a set of guarded actions, each of which takes the form of an enabling condition and an associated sequence of variable redefinitions or agent instance invocations. Each guarded action is viewed as expressing a privilege to act: if an enabling condition holds, a particular action may be performed. In interpreting a protocol for animation purposes, application-specific assumptions are invoked in the ADM (see §2.1) to model the way in which an agent exercises its privileges for action. There is no general principle to decide which

action to perform when there is non-determinism (i.e. two or more enabling conditions hold), nor is it always appropriate to presume that a privilege that is enabled will be exercised.

```
agent driver {
    const     maxCruiseSpeed = 70.0          /* miles/hour */
              minCruiseSpeed = 20.0          /* miles/hour */
    handle    engineStts, cruiseStts, cruiseSpeed
    oracle
              engineStts, cruiseStts,
              cruiseSpeed, measSpeed, brakePos
    derivate
              brakePos = user_input (brakePos_Type)
              accelPos = user_input (accelPos_Type)
    protocol
              engineStts == esOff → engineStts = esOn      /* on */
              engineStts == esOn → engineStts = esOff      /* off */
              cruiseStts != csOff → cruiseStts = csOff     /* switch off cruise controller */
              cruiseStts == csOff→ cruiseStts = csOn       /* switch on cruise controller */
              cruiseStts == csMaintain → cruiseStts = csOn /* return to manual control */
              cruiseStts == csOn → cruiseStts = csMaintain /* resume to cruiseSpeed */
              cruiseStts == csOn →                         /* maintain the current speed */
                 cruiseSpeed =| measSpeed |; cruiseStts = csMaintain
              cruiseStts != csOff ∧ cruiseSpeed < maxCruiseSpeed
                 → cruiseSpeed = |cruiseSpeed | +1         /* increase cruiseSpeed */
              cruiseStts != csOff∧ cruiseSpeed > minCruiseSpeed
                 → cruiseSpeed = |cruiseSpeed| −1          /* decrease cruiseSpeed */
    }
```

Listing 1

The user accesses the current state of significant observables (the status of the cruise control, the speedometer etc.) through a visual interface (see Figure 3). The user can interact with the display as a screen object, e.g. to redesign the interface, or observe the dynamic characteristics of the system by introducing an agent to simulate the dynamics.

The sensitivity of the accelerator and brake windows is defined by the modeller in such a way that the buttons and sliders initiate appropriate transitions under the control of the user. This user-interface provides an environment for the user that closely matches the perceptions and privileges of the driver. The user can thus experience the system as conceived through the model. In particular, the user may revise and experiment with the interface content and style, and make experiments with the perceived function of the controller for safety, efficiency and comfort. In principle, the display could be augmented with a physical driving seat on a platform subjected to forces corresponding to those in the model.

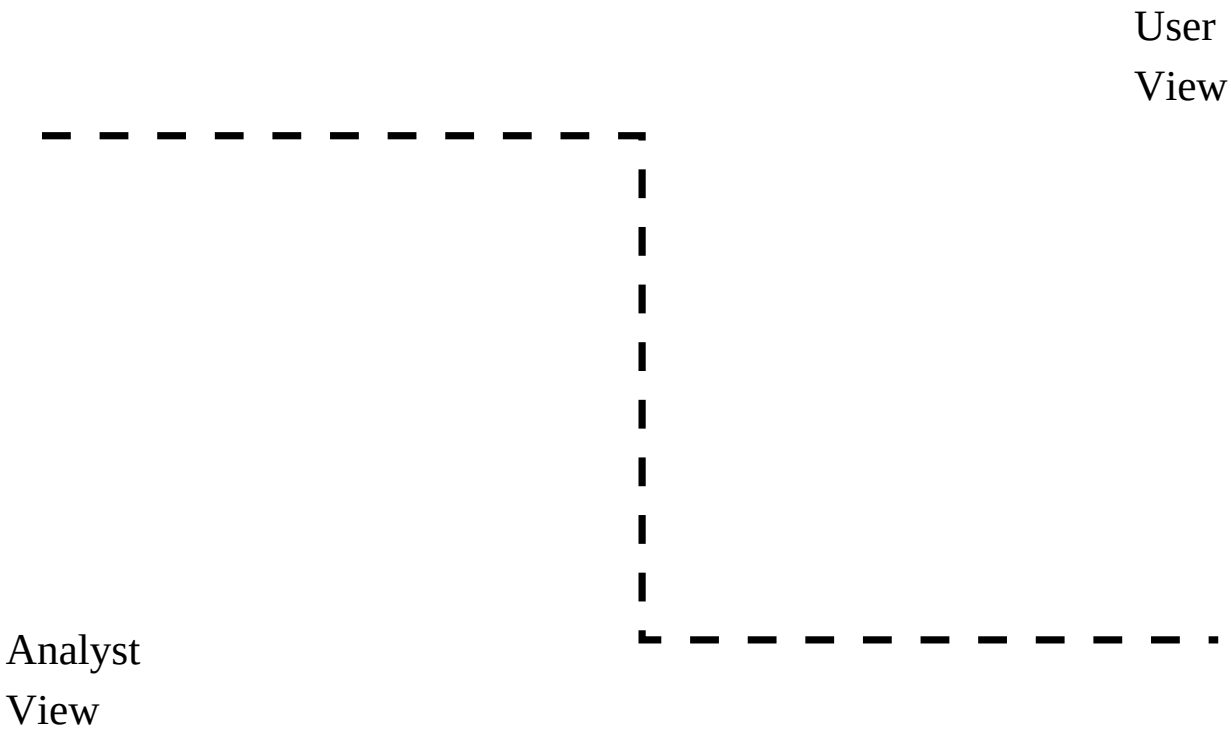

Figure 3

2.3.3 The Analyst's View

In the spirit of Jackson System Development [14] we emphasise the thorough modelling of the application domain prior to the consideration of functionality. This is the particular focus of the analyst, who has first been briefed by the user on the application domain and the provisional requirement. In this case the context must include a road with varying gradients, wind, a vehicle and time. These ingredients constitute the environment within which the cruise controller operates, and the analyst must ensure they are modelled with a level of physical faithfulness and sensitivity that will ensure a reliable context for the implementer's components, and therefore useful feedback

to the user. The analyst will be responsible for the representation of general physical laws (e.g. Newton's laws and air resistance). The 'vehicle' agent description in Listing 2 illustrates this, and makes use of 'derivate' variables which express indivisible relationships, corresponding here to physical laws.

```
agent vehicle {
    const
        mass = 1500          /* total mass of car and contents [kg] */
        windK = 10           /* wind resistance factor [N m^-2 s^2] */
        rollK = 100          /* rolling resistance factor [N m^-1 s] */
        gravK = 9.81         /* acceleration due to gravity [N m^2 s^2] */
        brakK = 150          /* braking constant [N m^-1 s] */
    state
        actSpeed             : analog              /* actual speed */
        accel                : analog              /* acceleration */
        windF                : analog              /* wind resistance force */
        rollF                : analog              /* rolling resistance force */
        gradF                : analog              /* gradient force */
        tracF                : analog              /* engine traction force */
        brakF                : analog              /* braking resistance force */
        brakePos             : analog (0.0, 1.0)   /* normalised */
        accelPos             : analog (0.0, 1.0)   /* position */
    oracle   brakePos
    derivate
        windF = windK * actSpeed^2
        rollF = rollK * actSpeed
        gradF = gravK * mass * sin (gradient * π/200 )
        brakF = brakK * actSpeed * brakePos
        tracF = enginePower / actSpeed
        accel = (tracF – brakF – gradF – rollF – windF) / mass
        actSpeed = integ_wrt_time (accel, 0)
    }
```

Listing 2

The equations of motion of the vehicle are represented by the derivate definitions in which the variables represent analogue quantities. Such variables illustrate a process of idealisation involved in the use of definitive scripts, whereby ideal entities (such as points and lines) are represented on the display to a degree of approximation that can in principle be chosen with arbitrary accuracy. In the animation, the values of these variables are updated by integration with a step-size that, conceptually, is a parameter that can be chosen to be arbitrarily small. Other aspects of the animation, such as the specification of the speed transducer, also involve integration – in this case to reflect the process by which the actual speed of the vehicle is converted into a sampled signal.

In this context, the step-size of the integration is a design feature of the model, chosen to satisfy the requirements of the engineering model with respect to feedback control for the cruise speed maintenance. It is on the basis of such explicit relationships that our animation may claim to have explanatory power.

2.3.4 The Implementer's View

The implementer has to choose suitable components, and determine realistic properties, so they can be used to meet the overall requirement. For the vehicle dynamics, such agents as a speed transducer, a throttle manager and a cruise cutout are introduced. For the interface, other factors such as driver attributes, the speedometer, the ergonomics of the controls, need to be taken into account. Assurance about meeting system requirements is based upon the faithfulness of the component behaviour in the model. The role of the implementer in choosing appropriate models of the component devices is therefore crucial. What is 'appropriate' depends on physical constraints, the user's requirement, and the characteristics of the display and other output devices in the model. This is illustrated by the LSD description for the speed transducer (Listing 3).

```
agent speed_transducer {
    const
        wheelDiam = 0.45                          /* wheel diameter [m] */
        wheelCirc = π * wheelDiam                 /* wheel circumference [m] */
        countPeriod = 0.2                         /* counter/timer period */
        maxCountVal = 65535                       /* 16-bit counter */
    state
        measSpeed       : analog
        pulseRate       : analog                  /* wheel revs / sec [s^-1] */
        countVal        : analog                  /* counter/timer value [s^-1] */
    derivate
        pulseRate = actSpeed div wheelCirc        /* integer division */
        countVal = (pulseRate * countPeriod) mod maxCountVal
        measSpeed = (countVal * wheelCirc) / countPeriod
    }
```

Listing 3

The speed_transducer agent presumes that a transducer on a wheel emits one pulse per revolution; the speed is measured by a counter/timer that estimates pulse-rate. With this model of the speed transducer, it is possible to assess the effect of quantising the measured distance into multiples of the wheel circumference and of finite counter length. So the accuracy is determined by the size of the counter (in this case 16 bits) and the pulse rate. The process of incorporating such features into the model introduces sets of definitions with parameters associated with the components. It is from these definitions that we can extract detailed requirements for the components. The implementer

can also investigate on-line the effects of, for instance, unreliability in the counter's response to the wheel pulses, and the precautions that could be incorporated as a safe-guard.

2.3.5 Integration of Views in the Development

It is important to notice that we have described three views of the *same* modelling process which assumes a rich pattern of mutual interaction throughout its development. There is, for instance, interaction between the viewpoints adopted. There are abundant examples of such possible interactions. We can experiment with agent protocols and adapt system behaviour (e.g. modifying the 'look and feel' of a user-interface on-line): this calls for user-implementer interaction. We can explore the neighbourhood of a solution under small modifications (e.g. how degradation of the speed transducer affects the feed-back mechanism): this calls for analyst-implementer interaction. We can incrementally enrich the observational framework to reflect more subtle characteristics of agent activity (e.g. whether wind resistance forces could take account of the vehicle shape): this calls for user-analyst interaction. In this sense, the user, analyst and implementer are all co-developers. There is also the orthogonal interaction, open to each developer and illustrated in Figure 1, of experiment and observation with both the real world system and the model. We are thus able to monitor the relationship between the system behaviour as partially realised by the implementer and the system as conceived by the user. When the user and analyst believe that an acceptable and realistic requirement has been identified, the current state of the model can be 'frozen' and regarded as a specification. The model that has evolved, and been viewed up until this point as a description of a system, can then be interpreted as a prescription for system components.

3. Conclusion

The modelling process that we have described here is intended to form the major part of an overall development process that would have wide application for the specification, design and programming of complex systems. By addressing, for the most part, our original objectives as set out in §1.2, we have focussed on the impact of our development process on the investigation and evolution of requirements. It will have become clear that our development approach is indeed a process, rather than the traditional cycle with feedback. We view it in the spirit of a high-level concurrent design process between cooperating developers, and it is in this sense that it "integrates the requirement, specification and design phases" (§1.2). Further details of applications to such multi-agent cooperative tasks appear in [1].

As explained in §2.1, and illustrated many times in our case study, our modelling methods allow for the direct intervention of the modeller in imitation of real world experience. It is this facility which supports the cycles of observation and experiment which are central to the whole process and which give us confidence in the reliability of the components, and the systems, which we are seeking to specify and construct. The provisional, informal requirement for the vehicle cruise control given in §2.3.1, is elaborated and refined from the viewpoints of user, analyst and implementer in the succeeding sections. The final, precise and detailed requirement for the components and their interactions is embodied in the specifications of the user, analyst and implementer. We have not explicitly extracted the formulation of these requirements but we have indicated in more detail in the case of the speed transducer (§2.3.4) how this can be done. This same example illustrates how the requirement for a component of a system can be extracted from the system requirement, after

negotiation between all developer viewpoints concerned, and in the light of their experiments and experience.

Two further features of the modelling process, highly relevant to requirements, should be emphasised. It is a remarkably flexible process, allowing on-line re-definitions during animation of the model. This helps us to explore, and to meet, modifications of the requirements with the minimum of additional work, and to respond rapidly to the future evolution of the requirement. The fact that observations are fundamental in the model means that arbitrary enrichment of the possible observations, and consequent modes of interaction, can be achieved by modification to the model rather than its reconstruction. In addition, the physical faithfulness that we strive for in the component models lends an explanatory power to the overall model. It behaves in a 'realistic' manner not for pragmatic, or arbitrary reasons, but because it embodies, at least to some extent, elementary physical knowledge (such as Newton's second law). Of course, the imperfect state of our knowledge, quite apart from other shortcomings of our models, is a limitation on their faithfulness which may be beyond our control. But the easy modifiability of the model allows for the incorporation of new, or revised, knowledge, in a piecemeal fashion close to the way in which we ourselves assimilate revisions to our knowledge.

The recent survey of methods and tools in requirements engineering by Hofmann [13] contains a summary of eight major and representative approaches to requirements. The tools available for these methods are not described in detail, but they appear to be limited essentially to editors (including graphical editors), databases and code-generators. In this context the single most distinctive feature of the modelling process we have presented is its capacity, throughout its use in a development, to accommodate real world experience, and to offer direct experience from the model. This may well be compared with the situation of an engineering team designing a large complex structure, such as a ship. There will be a vast body of theory, data and experience at their disposal, as well as their combined intelligence, imagination and creativity. They may have distributed machine support to integrate these resources, and generate and record negotiated designs. But, in addition to all this, they will probably build a model ship, and subject it to controlled experiments in a wave tank. The role of the direct experience of our model in building a reactive system is of a similar kind. It is one source of knowledge, among others, but being represented within the same overall framework it is maintained in an interactive relationship with other sources, and the requirement that is being refined and finalised.

As we showed in §1.3, a deeper understanding of programming arises from viewing it in its real world context. The behaviour of software in context requires knowledge of the implementation of the relevant interfaces and surrounding components. It is only in terms of such behaviour that software can be useful, and that we can properly formulate a detailed requirement. This is why we believe the traditional logical separation of requirements and implementation, while undoubtedly valuable for many purposes, should not translate into a practical separation in development. It is also why experience plays an integral part in our modelling process, thereby making it a truly empirical process.

The relationship between formal models and our empirical models is complex, and is a current topic of our research. In particular, we are investigating ways of deriving formal specifications from our models. A complementary area of research involves devising suitable 'metaphors' for the representation of a wider range of experience than is currently available within our models.

References

[1] V. Adzhiev, W.M. Beynon, A.J. Cartwright, and Y.P. Yung. A computational model for multi-agent interaction in concurrent engineering. In *Proc. CEEDA '94*, pages 227–232, University of Brighton, 1994.

[2] W. M. Beynon. New paths for programming in theory and practice. Unpublished report of University of Warwick, 1992.

[3] W. M. Beynon, I. Bridge, and Y.P. Yung. Agent-oriented modelling for a vehicle cruise controller. In *Proc. Eng.Sys. Design & Analysis Conf.*, number 47-4, pages 159–165. ASME, 1992.

[4] W.M. Beynon and S.B. Russ. The interpretation of states: a new foundation for computation. Technical Report 207, University of Warwick, 1992.

[5] G. Booch. Object-oriented development. *IEEE Transactions in Software Engineering*, 12(2):285–292, 1986.

[6] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[7] A. M. Davis. *Software requirements Analysis and Specification*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[8] M. S. Deutsch. Focusing real-time systems analysis on user operations. *IEEE Software*, pages 39–50, September 1988.

[9] M. S. Deutsch. Enhancing testability with scenario-oriented engineering. In *6th Intl. Conf. on Testing Computer Software*, pages 1–12, Washington D.C., 1989.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:232–274, 1987.

[11] D. Harel. Biting the silver bullet: towards a brighter future for system development. *IEEE Computer*, January 1992.

[12] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[13] Hubert F. Hofmann. Requirements engineering: A survey of methods and tools. Technical Report 93.05, Institut für Informatik der Universität Zürich, March 1993.

[14] Michael Jackson. *Structured System Development*. Prentice Hall, 1983.

[15] T. Minoura, S.S. Pargaonkar, and K. Rehfuss. Structural active object systems for simulation. In *Proc. OOPSLA '93*, pages 338–355, 1993.

[16] Y. P. Yung and W. M. Beynon. Agent-oriented programming for visualisation. Unpublished report of University of Warwick, 1991.

[17] C. Zarmer, B. A. Nardi, J. Johnson, and J. Miller. Zen and the art of application building. In *Proc. 25th Hawaii Intl. Conf. on System Sciences*, number 2, pages 687–698, 1992.