Do interactive systems need specifications?

Anthony Hall

Praxis Critical Systems,
Bath, UK

Abstract The obvious advantages of prototyping and incremental development for interactive systems lead some people to believe that specifications of such systems are unnecessary or even harmful. I question whether there really is a conflict between specifications and prototypes. In fact the two schools have more in common than is usually supposed. Both specifications and prototypes can be understood as theories about the system to be developed. Both have important and complementary roles in development.

1 Introduction

Interactive systems seem to be particularly fruitful subjects for prototyping and incremental development. Such approaches are increasingly popular and their attractiveness, compared with the known problems of traditional waterfall development methods, seems to throw into question the whole notion of specification and design. If we can build the system in small increments and validate each increment with its users, why write a specification at all? Why have a design, when it is only an approximation to the code?

I suggest that, valuable though these evolutionary approaches are, it is a mistake to think that they can replace specifications. On the contrary, I believe that the two approaches are complementary and it is only by using both that we can build interactive systems of any size. This paper tries to substantiate that claim.

First, I summarise the main benefits of prototyping and incremental development. Then I look at some of the problems that can arise if they are used to the exclusion of other methods. In contrast, I describe the main ways in which writing specifications can contribute to system development and assurance.

The central part of the paper proposes a unification of the two approaches: I propose that we can view each as a particular type of theory about the system under development. This suggests that, rather than being opposed, the two approaches are trying to solve different aspects of the same problem and indeed that their similarities are deeper than their differences. From that perspective, I show where the contributions of each approach are most valuable, and how they can complement each other.

We put this idea into practice on our projects, and have developed major operational systems using it. I describe how the idea works out in practice, and what problems we found in using it. This leads me to some research issues which I believe need to be addressed and which I hope will stimulate discussion at the workshop.

2 The case for evolutionary development

One of the fundamental difficulties of creating computer systems is the IKIWISI principle. I find it hard to envisage what it is I want, but I am sure that *I'll know it when I see it*. This principle describes the behaviour of users, who find it far easier to assess a working model than to review a technical document. It accounts for the frustrating experience of delivering a system only to have the user say "That's about right. There's just one small thing...." It suggests that document-based approaches, epitomised by the traditional waterfall lifecycle, are bound to fail.

The IKIWISI principle works just as well for developers, although this is less commonly acknowledged. Most developers are much more comfortable trying out a program to see what it does than thinking about whether a specification is right or not. We can be cynical and think this is just a preference for hacking over serious thought, but we have to recognise that there is a powerful psychological fact which we need to take into account if we want our development methods to be effective.

The best tool we have for overcoming the problems caused by IKIWISI is prototyping. Rather than tell the users what they are going to get, we show them. The hope is that what we show them is realistic enough, so they will explore its behaviour and tell us whether it is really what they want.

Prototypes are not only useful for showing users what they will get. They can also be used by developers to explore potential designs to see whether, for example, they are feasible and will perform adequately. It is important to realise that the kind of prototype that a designer needs is quite different from the kind a user needs. The designer may be interested in performance, for example, and will need a prototype which has a realistic processing load but does not need to have a usable interface. The user, on the other hand, will want the user interface to be as realistic as possible but is not concerned about how much processing is going on behind the scenes.

Fortunately, the need for prototyping is now accompanied by powerful means for achieving it. Technologies like user interface development kits, databases, languages like Visual Basic and the powerful hardware they run on mean that we can build complicated user interfaces with relatively little effort.

This combination of an obvious need and the technical means for meeting it has led to widespread interest in the evolutionary approach to development. The evolutionary process starts with a prototype: this is evaluated and improved, and a second prototype is built incorporating the lessons of the first. Eventually these prototypes will evolve into a full-blown system.

This process clearly works with, rather than against, the IKIWISI principle. It is also sometimes claimed that it is more responsive to change: that as requirements change, the prototypes will evolve to meet the new requirements. Indeed, so appealing is this approach that some people believe that it is a complete solution to the system development problem, and that there is no need for documents such as specifications or designs.

3 Is this enough?

Evolutionary development is clearly very attractive. It addresses some of the most difficult issues which arise when we try to satisfy user needs and deal with a changing world. Before we throw away all our previous ideas, however, we should look at the potential problems of adopting such an approach.
I am especially interested in operational systems—those which are used to support applications like railway signalling, air traffic control and fly by wire aircraft. All these are highly interactive and need to be prototyped and evaluated in the way I have described. However, there are many other issues to be addressed when building such systems.

3.1 Complexity

A key aspect of such systems is that they can be very complex. I am told that if you listen to cockpit voice recordings from fly by wire aircraft which have had difficulties, one of the most common things you hear is: "*Why did it do that?*" Indeed, how often have you asked just that question of your own word processor? Such a question arises when the user no longer has a clear mental model of what the system is doing. I think it is complexity which causes the difficulty in constructing such a mental model.
Complexity can arise from several sources. One of them is simply that the system may have a huge number of possible states. The tracks, points and signals for a reasonable railway junction probably have something like 10^{20} possible states—and that's before we consider the trains. No amount of prototyping is going to explore a significant fraction of that state space.
A more intractable source of complexity is, of course, concurrency. While it is fairly easy to expose the user to the main behaviours of, say, a single-user word processing system, it is literally orders of magnitude more difficult to explore a multi-user system (For an example, see [1]). It is even more difficult if the external environment is an unpredictable source of events. If the events and responses are time critical, the task gains yet another dimension of difficulty.
Of course we can—indeed we must—build prototypes of such systems. But we cannot expect these prototypes to tell the users everything about how the systems will behave in real life. Even if the prototype were the complete system, the evaluations carried out by the user could only explore a minuscule fraction of its behaviour. Faced with a new combination of circumstances, the system might do something quite unexpected.

3.2 Partiality

An evolutionary approach to system building has the attraction that it produces something looking like a real system early on in the development. However, it is important to remember that any such early version must necessarily be only a partial system. It will fail to display some of the characteristics of the final system. It may have only a limited set of functions available, or support only a single user, or differ in some other way from a full implementation. This may seem obvious, but it is easy to forget that it has important consequences. For example, a prototype may have only some of the functions available: it will then be impossible to detect harmful interactions involving other functions which are not included in the prototype.

In some applications such incompleteness may not be very important. I think it is annoying that different features of my word processor don't work together properly, but it is no worse than that. I may be prepared to pay that price for having the features in the first place. However, if we are building critical systems controlling trains or aeroplanes, such unexpected interactions could be disastrous.

Experience shows that errors in computer systems do frequently arise from incompleteness in the behaviour of the system. Such incompleteness, sometimes called "missing design", means that the system has simply not been programmed to cope with particular combinations of circumstances. As systems get more complex the scope for missing design increases.

There are, broadly, two styles of evolutionary development. Both are useful, but neither of them completely addresses the problem of missing design.

Throw-away prototypes

The first style is the "throw away" prototype, which is built to explore particular aspects of the system's behaviour. Once the prototype has been evaluated it forms part of the definition of what the real system is meant to do. For example, Sutcliffe [7] advocates a "requirements specification comprising the concept demonstrator, a set of analysed design rationale diagrams expressing the users' preferences for different design options, and specifications as text, graphics or more formal notations depending on the requirement engineer's choice."

Using the prototype as part of the definition of what the system is to do seems to me to be problematic. It begs the question of what it means to build something "like" the prototype. This is a problem even if the prototype is a fully functional system: I once talked to a hardware engineer who had been told to design a new disk drive. The new drive was to "exactly like the current drive, but faster". The trouble was, he had no idea what "exactly like the current drive" might mean.

There are two complementary difficulties: determining what behaviour the current drive has; then determining which parts of that behaviour are important, and which are just incidental. Failure to solve the first means that essential behaviour will be missed; failure to solve the second means that inessential behaviour will be preserved, constraining the designer and limiting the scope for improvement.

This is a problem not just for the designer, but for the customer too. Suppose the designer brings along a disk drive which, he claims, meets the requirement: how is it to be tested? Does it have to be indistinguishable from the current drive? Clearly not, for at the very least it is going to differ in speed. But does it, for example, have to behave in the same way when data are cached? Who knows?

My conclusion is that prototypes cannot be used as a *definition* of what the system is to do. They are an essential step on the way to arriving at such a definition, but they are not in themselves enough.

Incremental builds

The second kind of evolutionary development is incremental. Rather than being thrown away, each intermediate product is a version of the final system. The question of using it as a definition of the system does not arise: it *is* the system. However, this does not remove the problem of partiality. It is just as likely that, when new functions are added in later increments, they will interact in unexpected ways with the features of the earlier increments.

The incremental build approach may not even be feasible. If, for example, the final system is to be concurrent, distributed and have high reliability requirements then there needs to be a strong architecture in place before any functionality at all can be built. Such an architecture depends more on the non-functional requirements than on functionality seen directly by the user. Experiments like Sutcliffe's show that it is hard to gather non-functional requirements from users by working with prototypes.

3.3 Premature commitment

There is one other danger in prototyping solutions. That is precisely the fact that we are showing the user *solutions*, not talking about the *problem*. Some assumptions must already have been made about what the users' requirements are, and what kind of solution they are looking for. Prototyping on its own makes it difficult to question these assumptions. For example, air traffic controllers currently use flight progress strips to record details of flights. If we try to develop a system to replace flight progress strips, we will naturally start by prototyping different ways of presenting the information on the computer. But what we are unlikely to do is to rethink the whole question of what information is needed, when it is needed and how it might best be made available. To do that we need to do more than just present trial solutions: we need to think more deeply about the problem.

4 Requirements and specifications

Thinking about the problem is called Requirements Engineering. The outcome of a good requirements engineering exercise is a definition of the problem to be solved, and the constraints that must be met by its solution. This definition is a requirements specification. A requirements specification is much more than a sketch of the proposed system: it defines everything the implementer needs to know in order to produce a solution to the problem. Good specifications have some valuable characteristics:

- A good specification is *abstract*: it concentrates on the essence of the problem at the expense of detail. This helps to master complexity.
- It is, in a useful sense, *complete*: it covers all the important properties that the solution needs to have.
- It is, far more than a prototype can be, *free of bias* towards a particular solution.

4.1 The power of abstraction

One of the most powerful tools for writing specifications is abstraction: ignoring some of the details of the problem to concentrate on its essence. Abstraction is, of course, a different form of partiality: like a prototype, an abstract statement about a system misses things out. But it is partiality along a different dimension, and it has different benefits and costs.

Abstraction is most useful when we have the problem of managing complexity. If we have a system with a huge number of states, we don't enumerate them all: instead, we impose some structure which allows us to understand them. We build a model in which the concepts are at a higher level than the individual elements of the problem, and thus the model is small enough and regular enough to be comprehensible. Instead of thinking about individual track circuits and points, we think about routes; instead of thinking about individual train positions, we think about whether the route as a whole is free or occupied.

4.2 Completeness

To say that a specification is abstract is to say that it does not describe certain aspects of the problem. However, within the scope of what it does describe, a specification can be complete. That is, it can describe every possible behaviour of the system. For example, one can use quantifiers to say things like "*all* controlled signals on unset routes are red". This may not be as immediately appealing as seeing a prototype force signals to red when routes are unset, but it gives one much greater confidence that the system will *always* do the right thing. One reason is that the rationale for the signals being red is explicit in the rule, so the mental model of the system gives us an answer to "Why did it do that?". The second reason is that the rule clearly covers every case: it won't suddenly stop working if there are more than three routes, or if a new kind of route is implemented.

Another important property of specifications which is not shared by prototypes is that it is possible—indeed, quite easy—to say what the system will *not* do as well as what it will do. Important properties such as safety and security are often of exactly this form: "A route will *never* be set if conflicting routes are set". No amount of experimenting with a prototype can establish negative properties of this sort.

A third reason for writing specifications is to capture non-functional requirements for the system. Properties such as performance, capacity and reliability are not easily determined by experimenting with prototypes [7].

In contrast with a prototype, which is an example of what a system should do, a specification is a definition of what it must do. It is therefore a sound basis for a contract to implement the system. The relationship between an implementation and a specification is one of satisfaction: the implementation does or does not satisfy the specification. In contrast to the similarity relation, satisfaction can be determined unambiguously. If my hardware colleague had been given a specification of the current disk drive, he would have had no difficulty deciding what the new one should do. For that purpose, the specification is actually better than the real thing.

A good specification, therefore, serves everyone:

- Users can decide whether or not it defines what they want.
- Implementers can design and build a system which satisfies it.
- Verifiers can check whether or not the system does indeed satisfy it.

4.3 Describing requirements

It is possible, and indeed desirable, to avoid commitment to a particular solution when writing requirements. The key idea, described by Jackson [4], is to define the effects that are to be brought about in the real world. Some of these effects may be achievable directly by the system to be built, but others may not be. A requirement for a railway signalling system is to prevent collisions between trains; on its own, however, the signalling system cannot do that. It relies on a lot of real-world knowledge about the behaviour of train drivers and the braking distance of trains.

Jackson has shown how to relate the specification of a system to the requirements it is intended to meet. In order to do this one has to prove that the specification, taken together with relevant facts about the real world, entails the requirements. Such reasoning needs the specification to be written down explicitly. It also needs the relevant real-world properties to be made explicit. For example, the correct behaviour of an interactive system might well depend on its users behaving in particular ways. Making these assumptions explicit is one of the benefits of having specifications.

The "missing design" that is such a common cause of errors really means that the world is behaving in some way that the designer did not expect. This may be because the designer expected the world to behave in a different way; more often, it is because they never thought about the question at all. In either case, making the assumption explicit will go a long way to avoiding the problem and making the system behave consistently in all circumstances.

5 A Synthesis

Evolutionary development and specification-based development are frequently thought of as opposites. Proponents of one suppose that the other is wrong. I do not believe this at all. Both approaches emphasise the importance of the real world and the user. Both approaches try to establish, early on in the development, that it is going in the right direction. They have more in common with each other than either has with, say, so called structured or object-oriented approaches. One way of understanding more about them is to think of prototypes and specifications as being different kinds of theory about the system to be developed. We can then ask what each kind of theory is good for, and how we can best use them both.

5.1 Theories and Refutations

Popper [6] pointed out that the defining characteristic of a scientific theory is that it is refutable. If it is wrong, there is a way of showing that it is wrong. Furthermore, that is the best that can be done: there is no way of ever showing that a theory is right.

If we think of specifications, prototypes, designs and other artefacts of the development process as theories, then we can evaluate them by their degree of refutability. A powerful theory is one that says a lot: it could be refuted very easily. A weak theory does not say very much: most phenomena are consistent with it and it is hard to refute. Structured analysis typically gives rise to weak theories: it is very hard to argue with a data flow diagram, because it means so little.

Prototypes are often strong theories, because of the IKIWISI principle. If a prototype exhibits undesirable behaviour, it is easy to criticise it: that criticism is a refutation of the theory “the system should look like this”.

Good specifications are also strong theories in this sense. If the specification says “A route can be set whenever its subroutes are clear”, a signalling engineer will be able to tell you that it is wrong. Similarly, if you are relying on a description of the world which says “track circuits are activated whenever there is a train on the track”, anyone who has experienced leaves on the track will be able to tell you how wrong you are.

5.2 Getting the best of both worlds

Clearly, specifications and prototypes are very different kinds of theory. I believe that both kinds are needed. That is because each is good at describing different aspects of the system. A prototype will quickly reveal defects in the user interface. On the other hand, even if it makes unwarranted assumptions about the real world it is unlikely to be falsified provided it behaves satisfactorily in the obvious cases. Conversely, a specification can often reveal inconsistencies but, provided it is consistent and complete it may be hard to find anything wrong with it, even if what it describes would be completely unusable.

The question for developers, therefore is not “Shall we write a specification or shall we build a prototype?”. In my opinion specifications are always necessary, because it is only specifications that really define the system. The question is: “What kinds of specifications should we write, and how should we attempt to falsify them?” And, in particular, “What aspects of the specification should we prototype?”

These are open questions, and I do not offer a definitive answer. I will describe some of our experience and explain why I believe it may have some general validity.

Three-level model

Like others, [5], we find it useful to structure user interactions with a system into the usual lexical, syntactic and semantic levels. Lexical specifications describe the atoms of the interaction: keystrokes, mouse movements and so on. Syntactic specifications describe the dialogues between the system and the user: “select an object, then select an action”, for example. Semantic specifications define the meaning of the interactions by describing their effects in the real world. They provide the users’ conceptual model of what the system does.

This structure helps to manage the complexity of interaction specifications.

Use appropriate notations

Each of these levels is best described in its own notation. I think that one of the big mistakes that developers and theorists make is to try to use a single notation or a single development method for everything. I have seen attempts, for example, to describe the physical appearance of a screen using a Z model of the pixels. This is nonsense: the only sensible way to describe the appearance of a screen is to draw a picture or show the screen itself. Conversely, as I have discussed at length above, pictures or prototypes are quite inadequate for describing, say, safety properties of a signalling system.

Use appropriate verification methods

Just as different notations are appropriate for the different levels, so are different verification methods. In my opinion, prototyping is the only practical method for checking the lexical level. It is only by actually trying it out that you can tell whether, for example, it is easier to use a touch screen or a mouse to select particular trains on a track layout. Prototyping is also the best method of verifying the syntactic level, although here there are other methods that can be used to supplement it. For example, there are several different sequences of actions one might choose for a signaller to select a route through a junction, and the best way of finding which the signallers prefer is to let them try each one. In addition, though, once a method has been chosen then its syntax can be specified by a state-transition diagram. This allows it to be checked for completeness and unambiguity.

Prototyping at the semantic level is a necessary technique, but here it can only be one technique among others. It is the semantic level where the complexity of the state explosion, concurrent access and timing criticality prevents complete exploration. The number of states needed to verify the lexical representation is typically about 10^1. The syntax might generate 10^3. An exploration of the semantics might need 10^{20}. It is certainly necessary to check that the obvious things expected by signallers actually happen when they set routes. But since they can try out only a tiny number of cases, other techniques such as formal reasoning must be used to try and falsify the specification. For example, one can propose "challenge theorems"—properties that one believes should be ensured by the specification—and try to show that they are true. It is particularly important to use such techniques to establish negative properties such as the impossibility of setting conflicting routes.

6 The Synthesis in Practice

We have used these ideas in developing an air traffic information system. Our experience has been published [3], and has also been used as the source of case studies for the Amodeus project. [2]

6.1 Overall approach

The basis of our approach was that our biggest problem was to define the system at the conceptual level. Our central specification was therefore a formal model of the system state and behaviour, written in VDM. This specification was supported by user interface specifications for each of the types of user. In addition there were interface control documents defining the interface with other systems, and a concurrency definition defining the concurrent aspects of the system's behaviour.

We adopted the three-level model and the user interface specifications were confined to the syntactic and semantic levels. The semantics was defined by relating each dialogue to an operation specification in the VDM. The syntax was defined by state transition diagrams, and the lexis by pictures of the screens.

The granularity of this relationship was different for different interfaces. At one extreme, there was a page editing facility whose formal definition consisted of a single operation "Edit Page". At the other, there was one device (called a CERD) whose operation was so complex and critical that every keystroke was separately specified in VDM.

We built prototypes of each of the user interfaces. The purpose of these was primarily to evaluate the lexical and syntactic definitions. In the case of the CERD, these were so closely tied to the semantic definitions that we were necessarily verifying the semantics as well. The prototypes were strictly throw-away: the knowledge we gained from them was incorporated into the written specifications. We found many useful facts from these prototypes. For example, we discovered that the layout we had proposed simply did not work in the actual furniture of a controllers workstation, because it was physically too difficult to select the buttons accurately. We also found that the keystroke sequences we had proposed did not accord with the controllers' mental model: we had not recognised that they always thought of sequencing flights in terms of putting one flight behind another, rather than in front of another. However, we also needed a clean and complete specification to describe exactly which flights would be on the screen and where they would be: the combination of passage of time, flights landing, and controller actions simply presented too many possibilities to work through exhaustively with the prototype.

6.2 Effectiveness of the approach

Overall this approach was highly successful. The system is in use and has measurably better reliability than comparable systems. It is liked by its users. Perhaps most significant in this context is that an abnormally small proportion of its faults are specification errors—there is almost no "missing design". I believe this is because we did use an effective combination of methods to specify the system and to validate that specification.

We did have some problems using this approach. Some of these arose simply because we did not do our job well enough; others, however, were more fundamental and point to the need for further research and development.

One problem was that the specifications of the user interfaces were not always complete, because although we specified the user side of the dialogue completely, we omitted some of the system actions. The effect was that the implementers made wrong choices about, for example, what colours should be used for certain conditions.

A related problem is the difficulty of tying together the different specifications. Indeed this is probably the root cause of the first problem. If we had had a more formal relationship between semantic actions and user interface behaviour, we would have avoided some of the difficulties.

The third problem was more subtle, and was in fact discovered by the Amodeus investigations [2]. It turned out that the CERD specification allowed some undesirable behaviour: it could discard messages without the user being aware of it. The problem illustrates many of the points in this paper. First, it depended on a particularly unfortunate coincidence of concurrent events. Simple prototyping would never have found it. However, we failed to find it even though we also had a formal specification of the CERD interface. That is at least partly because we did not have a formal model of the environment and the users, so implicit assumptions about their behaviour went unchallenged.

In spite of these problems—indeed, because there were so few of them—I am convinced by this experience that the overall approach here is a practical and effective way to develop interactive systems for critical applications. I hope that the problems we did find can be addressed by better practice and more research.

7 Where should we go from here?

There are many areas of research where advances would help us to develop interactive systems in a more cost-effective way.

I suspect that we will always need different notations for different aspects of system behaviour. Research in multi-paradigm specifications [8] is relatively recent but promises to address this issue. We need to understand better how to divide the different aspects of the problem, how to specify each one and how to put the specifications back together again. Tool support for the combined notations would of course help enormously.

We need a closer integration between specifications and prototypes. A seamless generation of prototypes from comprehensible specifications is the ideal: there is a long way to go.

We also need to understand more about how to use prototypes. I have suggested that prototyping is most effective at the lexical and syntactic levels. It is also essential at the semantic level, but there it can never be a complete solution. We need to understand how to combine validation by prototyping with other aspects of validation such as proofs of consistency and completeness.

We need to understand more about the behaviour of users. It is only recently that the role of environment descriptions in requirements has really been understood, and of course users are a key part of the environment of any interactive system. I would like to see a synthesis between the user modelling work and some of the work in requirements engineering.

8 Summary

There has traditionally been a big difference of approach between those who believe in the importance of rigorous specification and those who believe in evolutionary development. I believe that this difference is founded on a mistaken belief that the two are antithetical. In fact, they are addressing different aspects of the same problem. That problem is to establish, before we go too far in building a system, just what system we really need to build. No one technique can solve that problem, and only by unifying the approaches can we make progress. That unification depends on understanding the roles and limitations of both techniques.

References

1. Abowd, G. D., Dix, A. J.: Integrating status and event phenomena in formal specifications of interactive systems. Proceedings of the 2nd ACM SIGSOFT

Symposium on the Foundations of Software Engineering, Software Engineering Notes 19 (5), 44–52

2. Buckingham Shum, S., Blandford, A., Duke. D., Good, J., May, J., Paternó, F., Young, R. M.: Multidisciplinary modelling for User-centred system design: An air-traffic control case study. In: Sasse, A. Cunningham, J., Winder, R. (eds): People and Computers XI. Proceedings of HCI'96. London, Springer Verlag. pp. 201–219.
3. Hall, A.: Using formal methods to develop an ATC information system, IEEE Software 13 (2), March 1996, 66–76
4. Jackson, M. A.: Software Requirements and Specifications, ACM Press Addison Wesley, 1997
5. Knight, J. C. Brilliant, S. S.: Preliminary evaluation of a formal approach to user interface specification. In: Bowen, J. P., Hinchey, M. G. (eds): ZUM'97: The Z Formal Specification Notation. Springer 1997 (Lecture Notes in Computer Science, vol 1212, pp. 329–346)
6. Popper, K.: The Logic of Scientific Discovery.
7. Sutcliffe, A.: A technique combination approach to requirements engineering, Proceedings of the 3rd International Symposium on Requirements Engineering, January 1997, IEEE Computer Society Press, 65–74.
8. Zave, P., Jackson, M.: Where do operations come from? A multiparadigm specification technique, IEEE Transactions on Software Engineering, 22(7) July 1996, 508–528