What Does Industry Need From Formal Specification Techniques?

Anthony Hall
Praxis Critical Systems

Abstract

In this paper I examine what industry really needs from formal specification techniques. I first describe the background to our use of formal techniques. I then look at the role of formal specifications in development and consider what are the important questions that need to be asked. I illustrate these with some practical experience on industrial projects and conclude with some lessons we have learned about for formal specification techniques and about the tools which support them.

1. Background

Praxis Critical Systems is a systems and software engineering company whose business focus is the development of systems where the cost of failure is high. Our main markets are aerospace including avionics and air traffic control, railways, finance, medicine and similar applications. To develop systems with the high integrity needed for these applications we deploy a strong quality management system, the best technical methods and a comprehensive risk management approach. Formal methods are among the powerful engineering techniques we use in our day to day work.

2. What are the important questions?

Given the large number of competing formal methods it seems natural to ask "What is the best formal method". This is not really a useful question on its own, however. Before we ask about the best solution, we need to understand what problem we are addressing. I conjecture that much fruitless debate in this area, and some misguided exercises in formal specification, arise because there has not been a clear enough understanding of the problem to be solved. I suggest that before carrying out any formal specification we need to ask, and answer, the following three questions.

- *What* am I going to specify?

  Before we can write a specification, we need to know what it is we are trying to define. As I shall argue shortly, there are many possible answers to this question, and I believe that failure to distinguish clearly between them can lead to much confusion.
- *Why* do I want to specify it?

  Specification is not an end in itself. It is only a means to some other end, such as the deployment of a working system. We always need to ask whether specifying something will contribute to that end (and, if it will, whether it is the most effective means of doing so). I will suggest some areas where we have found that formal specification is useful, and examine why this is so.
- What *use* am I going to make of the specification once I've got it?

  Different formal notations are good at different things. For example if I want to animate my specification I need an executable notation; if I want my specification to be easily readable by people then I may want an expressive notation which is less suitable for animation or proof. I therefore need to consider, just as with any document, the intended audience for the specification and the use that will be made of it.

3. What is it useful to describe?

When building a system we need to understand many different things, ranging from the environment that the system will be used in to the detailed code that performs critical functions. All of these are candidates for specification. Although there are no universally agreed artefacts that are relevant to all systems, we find it useful to distinguish the following things to be specified.

- Domain Knowledge

  Parnas[1] and Jackson[2] have both pointed out that understanding the behaviour of the system's environment is a crucially important part of building any system. If we are building an air traffic control system, for example, then we certainly need to understand the behaviour of aircraft.
- User Requirements

  Systems are built with a purpose: to achieve some effect in the real world. The purpose of an air traffic control system, for example, is to prevent aircraft from colliding with each other while maintaining an expeditious flow of traffic. Note that these requirements are often not directly related to the system at all – they describe desired behaviour in the system's environment.

- System Requirements

  In order to achieve the desired effects in the real world, the system must itself exhibit certain behaviour. For example an air traffic control system must be able to track aircraft and manage flight plans. System requirements differ qualitatively from user requirements in that they define only behaviour which the system itself must exhibit: they can be given to suppliers as a definition of what they must provide.
- System Specification

  Typically, system requirements do not prescribe every detail of what the system is to do. In response to the requirements the supplier may provide a more detailed specification of the system's behaviour. This specification should be free of any design information – that is not of interest to the system's users – and it differs only in degree of detail from the system requirements. There is no hard and fast line between them.
- Design Structure

  The design structure of a system is entirely different from its specification. It defines the components of the system and how they interact. It is of direct interest only to the supplier and should not be relevant to the system's users.
- Subsystem specifications

  Subsystem specifications define the external behaviour of each of the system's components. If the subsystem is a code module, it defines exactly what behaviour the code must achieve. In some cases the subsystems are large developments in their own right which are contracted to another supplier.
- Process behaviour and interactions

  One particularly complex kind of component in a software-based system is a process: a component which has its own autonomous behaviour. Different processes work concurrently, often on different machines, and it can be particularly difficult and important to characterise their interactions.
- Code

  One way of showing that a module meets its specification is to characterise the behaviour of the code mathematically. Such low-level specifications are used in critical systems for development and assurance of code.

The following sections discuss the role of formality in the specification of each of these characteristics.

4. Domain Knowledge

Facts about the environment of a system are traditionally the subject of "systems analysis". Techniques such as context diagrams are used to identify the relevant actors in the domain and their interactions with the system. Important facts about entities in the domain are typically captured in entity-relationship diagrams and in domain-specific notations such as acceleration and braking formulae.

Formal notations can be used to supplement entity-relationship diagrams since they are, of course, much richer notations and can express far more complex properties than simple cardinalities. Typically these properties are expressed as state invariants in notations like VDM and Z. It is crucial to appreciate that what is being specified here is knowledge about the real world, not desired behaviour. Failure to appreciate this point can lead to serious errors. For example in specifying the state of an air traffic control system people are tempted to write an invariant that states that aircraft separations are maintained. This is a dangerous confusion between what *is* true and what one would *like* to be true.

Specifying domain knowledge can be very beneficial provided it is done accurately. Knowledge of the behaviour of the real world is frequently used to justify preconditions on operations and to show that only certain event sequences are feasible. Such restrictions can greatly simplify the implementation of the system.

5. User Requirements

User requirements for systems are typically couched in very high-level terms, and usually there is no great need for them to be precise. The most important characteristic of a user requirement statement is that it should be comprehensible to the end users. These requirements are frequently quantitative ("increase traffic by 10%") and often involve time ("land an aircraft every 90 seconds"). The best way of defining user requirements is usually by scenarios describing how the world should look when the system is working, and by quantified changes in real world measures.

For all these reasons we have not found formal methods to be useful or necessary in specifying user requirements.

6. System Requirements

As we move towards characterising the system to be built, precision becomes more important. The requirements for a system need to draw a sharp distinction between those systems which are acceptable and those which are not. They need to define all the properties which are important to the user. These typically include:

- state transitions;
- allowable histories;
- the transfer function of a control system;
- a Formal Security Policy Model;
- critical safety properties.

All of these are in principle specifiable mathematically. However, there are some problems in using conventional formal methods for carrying out such specifications. One issue is modality: typically system requirements are not all or nothing, but are prioritised in some way such as mandatory versus desirable. This sort of modality is not easily expressible. A second issue is to do with the way that system behaviour is characterised. For example we have recently developed a formal security policy model (FSPM) for a highly secure system, as well as a formal top level specification (FTLS). Whereas the FTLS was readily expressible in Z, since it defined a set of operations on the system, the FSPM was more difficult to express. Rather than defining particular operations, it was necessary to characterise certain properties of all operations (for example that they should not display secret material) without saying exactly what the operations were. This can be done in Z, but not by using the established strategy, and the relationship between the FSPM and the FTLS is fairly subtle [6].

7. System Specification

There is no difference in principle between a system specification and a statement of system requirements, but in practice they are at very different levels of detail. Broadly speaking, system requirements say what a system must do; the specification says what it will do. However, because the specification is never complete, it still permits a variety of different behaviours all of which satisfy the specification.

The main aspects of system behaviour which are typically specified include:

- abstract functionality;
- concrete interfaces;
- concurrency;
- performance;
- availability, reliability and maintainability.

Notations used in a specification must offer

- precision: this, of course is one of the main reasons for using formal notations.
- expressiveness: all the different aspects need to be expressible, and the specification should be as close as possible to the "natural" way of defining what is wanted.
- complexity management: any realistic system has a specification running to several hundred pages, and it is essential to structure it in a manageable way.
- verifiability: it must be possible to show that the specification is well formed and that a system which satisfies the specification will also satisfy the system requirements.

There are three different audiences for a system specification:

- users, so they can evaluate what they are going to get;
- implementers, so they know what they have to build;
- testers, so they know what the system should do.

The reasons for using formality in a system specification are:

- to achieve clarity;
- to achieve expressiveness; in particular to allow the specification to be written in user oriented terms, stating what the system will do rather than how it will do it and to allow the use of logical constructors such as "and" rather than programming constructors such as ";";
- to allow for analysis, in particular to allow formal demonstration that the requirements are met;
- to allow for refinement into design and code.

In practice, clarity and visibility are, in my opinion, the most important characteristics. The specification should make it absolutely obvious to the users what they are going to get. Furthermore by using the specification as a basis for testing, it becomes clear exactly how much of the system has been implemented and tested. This really works in practice: some years ago we developed a system, CDIS, using formal specification [3]. During integration of this system into its operational environment the customer's project director reported that "CDIS performed impeccably as expected". I believe that the "as expected" is a result of the clarity and precision of the specification and our use of it throughout the development.

Unfortunately, the different uses of a specification demand different characteristics in the specification language. In particular, expressiveness runs counter to the ability to execute the language, to carry out refinement and to carry out proof. Therefore these aspects have to be traded off against each other. I do not suggest that there is always one right choice: it depends where the greatest risks are. If the greatest risk is that the specification will define behaviour which is not what the user wants (and in my experience it often is) then expressiveness must take precedence over proof and refinement. However, if the greatest risk is that the subsequent development will be wrong, or that there will be some subtle inconsistency between the specification and, for example, a security requirement then a language which is better suited to proof and refinement may be more valuable.

Different aspects of a system require different specification notations. Indeed formal notations are only appropriate for some aspects – most notably abstract functionality – but where they are appropriate they are by far the best methods. This means that different languages will be needed for the specification of a single system. There will of course be points at which the specifications are talking about the same thing, and there the meaning is that the system must simultaneously satisfy all the different specifications. On the other hand I do not believe that one needs to imagine that there is a single underlying semantic model defining the whole system behaviour. One should recognise that different specification languages are needed precisely because they are talking about different things, and one should think of relating the specifications at the points of contact, rather than unifying them in some grand model.

Our projects typically use different specification methods for abstract functionality, user interface and concurrency (and other notations for performance and other aspects). We have used, for example, Z for abstract functionality, pictures and state machines for the user interface and CSP for concurrency. Typical points of contact are the fact that particular buttons on the user interface invoke particular operations characterised by Z schemas, and that these same operations appear as actions in the CSP model.

8. Design Structure

The high level design of a complex system is typically concerned with issues such as distribution of processing over different machines, communications between components and so on. We have found that the high level design is usually driven by the non-functional requirements such as resilience, performance, safety and so on and is not directly related to the functionality of the system at all. Looking at the architecture of CDIS, for example, one can immediately see that it is a highly resilient distributed system but one has no idea whether it is for air traffic controllers or, say, management of the electricity supply.

Just as there are different aspects in a system specification, there are different aspects to a system design so it is better to talk about design structures than to think of a single structure. Typical structures are:

- distribution structure – how functionality is allocated over machines;
- process structure – how functionality within a machine, and inter-machine communications, are assigned to concurrent processes;
- transaction structure – how the processes co-operate to carry out processing of units of work;
- calling hierarchy – how the functionality within a process is allocated to the layers of software and hardware.

Designing these structures is a creative task and we do not have good formal criteria for judging whether one design is better than another. Without such criteria there is little that formality can contribute to the high level design. Formal notations certainly do help us at the next level of detail, however, in defining the components and their interactions.

9. Subsystem specifications

The creative task of design is to identify the components and to allocate functionality to each component. Having chosen our components, we need to define them precisely: that means giving a precise specification of the interface that each component offers. Creating and maintaining these interface definitions is a key to controlling the development of a complex system. We have successfully used formal specifications for this purpose. However, the definition of these interfaces and their relationship to the system specification is not as straightforward as textbooks on formal methods sometimes claim. First, there is some subtlety in deciding what exactly is being specified; second, the conventional account of refinement does not match what is really needed in large systems.

9.1. Subsystems versus components

The problem of specifying components is illustrated in Figure 1.

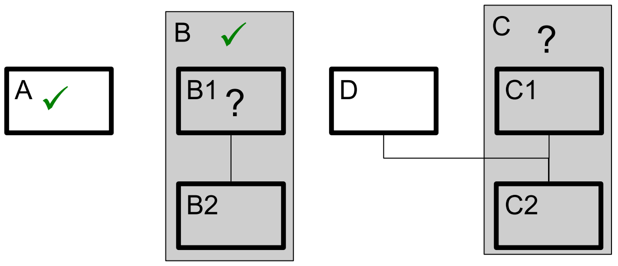

Figure 1: Specifying Components

If A is an isolated component at the bottom of the hierarchy, then it is straightforward to specify its interface. However, consider the subsystem B, made up of components B1 and B2. While we can certainly specify the behaviour of B as a whole, it is not so straightforward to specify the behaviour of component B1. To do this we need to define how B1 uses B2. This could be done either by specifying B1 as a functor which transforms the behaviour of B2 into the behaviour of B as a whole, or by describing the actual program that B executes using, for example, the refinement calculus. Either of these is a more complex undertaking than specifying a single component like A. The situation becomes more complex when there are shared components such as C2. Now it is not even straightforward to specify C, since its behaviour can be influenced by the external module D causing changes in the state of C2.

9.2. The Problem of Refinement

In notations like Z and VDM there is a well-developed theory of refinement which allows us to move from an abstract specification of a system to a more concrete design. Unfortunately this theory bears little resemblance to the real practice of design. That is because it assumes that the underlying structure of the design is the same as the structure of the specification, as shown in Figure 2.

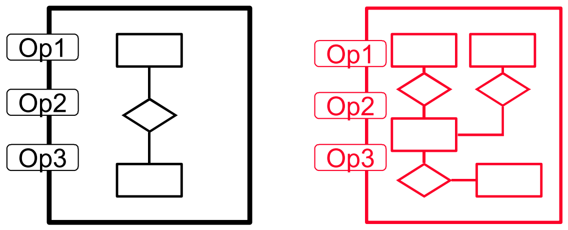

Figure 2: The Z/VDM Model of Refinement

In reality, however, the functionality is allocated to components in a much more complex way, as shown in Figure 3. Conventional refinement does not tell us how to relate the specification of Op1, say, to the various components which implement it in the design.

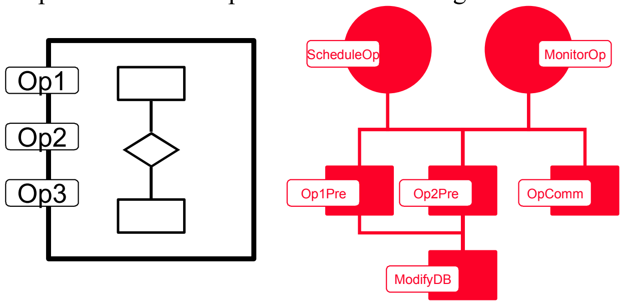

Figure 3: Allocation to Processes and Modules

9.3. Formality and Testability

In spite of these limitations, the specification of components is extremely useful in practice. First, it gives a clear definition to the implementers of the component. Second, it makes the components highly testable. An independent study on CDIS [4] looked at the number of defects in components that had been designed formally and informally. The number of defects found by system testing was similar in both cases; however, the number of defects found in the delivered system was lower in the formally designed components, showing that testing of these components had been more effective. This suggests that a benefit of formalisation is not so much in eliminating errors as in making them more visible.

10. Process behaviour and interactions

One of the most difficult aspects of design is management of concurrency. Distribution of processing across different machines and different processes on a single machine introduces many possibilities for errors like deadlock, livelock and race conditions. Fortunately there are well-developed notations and theories which allow us to develop and analyse concurrent designs to check for such problems. We have recently used CSP to validate the design of a fairly complex distributed system.

We started by writing a CSP specification of the intended behaviour of the system. In this specification, each external CSP action is an operation in the Z specification. The behaviour was described by processes which were based on an abstract view of the overall design. In addition we modelled the effect of failures of individual machines either through hardware or software faults. We then tried to show that this specification was correct by showing that it satisfied some critical properties: for example that it was deadlock-free, and that it satisfied some security requirements. We used the FDR tool to demonstrate these properties as refinement rules. In order to make this practical we had to simplify the model: we reduced the number of instances of identical processes, and we grouped operations into classes which and treated each class as a single operation. The FDR analysis proved extremely useful in showing that there were several flaws in our original specification, in particular in our proposed handling of failures.

We then developed a more detailed design of each of the components in the system. Each of these designs was validated by showing that it refined the corresponding part of the specification. Unfortunately machine checking was only practical for the simpler components; the state space of the more complex components was too large for a refinement check to be practical. Real systems still need more power from tools such as model checkers.

11. Code

It is of course possible to develop code in a fully mathematical way from formal specifications. We have not usually applied this degree of rigour. However we have found that developing code from formal module specifications in VDM or Z is a straightforward activity. Furthermore, the use of formal specifications in this way leads to exceptionally simple and well structured code [4]. Similarly, it is straightforward to develop the code of Ada tasks directly from CSP specifications. For highly critical applications, we use the SPARK annotated Ada subset and use the SPARK analyser to check for information flow errors in the code. There is evidence that this approach actually reduces the cost of critical code because of its ability to find obscure errors early in the process [5]. It is also possible to use the analyser to support fully formal development from pre and post conditions, and to prove the absence of run-time errors in the code.

12. Summary and Lessons Learned

12.1. Effectiveness of Formal Methods

We have found the use of formal methods highly effective. The key benefits come from their application early in the lifecycle, where the cost of errors is high. Formal methods do not eliminate errors, but they do highlight them and make them easier to find in reviews and tests. We have some experience of using formality retrospectively, to validate developments that have been carried out informally, and this is less effective than using proper methods from the outset. Formal methods are only appropriate for some aspects of development, and they have to be used in conjunction with other methods. Where formal methods are appropriate, they are not just effective but they also reduce costs. In a recent project we have found that proving properties of the Z specification was an effective and efficient method of detecting errors: it found more errors, at a lower cost per error, than unit testing for example. On the other hand proofs later in the lifecycle, based on the code, were less effective.

Overall there is considerable evidence that use of formal techniques can greatly reduce defect rates in delivered products [4].

12.2. Lessons for Methods

Different methods are needed for different aspects of development. There are many aspects of a system and its environment to be specified, and each aspect makes its own demands on the notation. All notations offer partial descriptions, and these partial descriptions need to be consistent. However, this does not mean that methods have to be integrated; it is better to think of relating different notations at the points where they are describing common phenomena.

Formal methods can be effective without necessarily carrying out formal analysis. The most important benefit of a formal notation is the clarity and precision it offers, not the analytical power it conveys. This needs to be borne in mind when selecting methods and when designing formal specification notations.

Since the main benefits of formal methods stem from their use early in the lifecycle, we need to find ways of making them more accessible to end users. The use of prototypes and of domain-specific notations are examples of how we might do this.

If we want to carry out development formally, we need to find much more powerful methods of refinement. These need to take into account the complexity of real systems and the big change in structure that takes place between the system specification and the system architecture.

12.3. Lessons for Tools

Tools are very important, but I do believe that they should be subservient to methods and not vice-versa. It is more important to develop expressive notations than to have good tools for handling obscure specifications.

Tools should be seen as aids to analysis, more than as methods of assurance. The real benefit of a model checker or proof tool is not when it tells you that your specification is correct, but when it demonstrates the flaws in your thinking. This is an important consideration for tool writers: they need to think far more about the error cases than the correct ones. This is similar to situation with compilers: there are far more compilations of incorrect programs than there are of correct ones, and good error reporting is an important requirement of compilers.

If formal methods tools are to be used in mainstream development they have to be fully automatic. Model checking can be applied by ordinarily capable engineers, but it is not reasonable to expect most engineers to carry out sophisticated proofs.

Unfortunately real systems are too big for the current generation of tools. We need huge improvements in the capacity of tools like model checkers if they are to be used routinely in real system development.

13. References

1. See, for example, C. L. Heitmeyer, R.D.Jeffords, B. G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology*, 5:231–261, July 1996.
2. M. Jackson, *Software Requirements and Specifications*. Addison Wesley, 1995.
3. A. Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software*, March 1996: 66–76.
4. S. Lawrence Pfleeger, L. Hatton. Investigating the Influence of Formal Methods. *IEEE Computer*, February 1997: 33–43.
5. M. Croxford, J. Sutton. Breaking Through the V & V Bottleneck. *Proceedings Ada Europe 1995*. Springer Verlag Lecture Notes in Computer Science 1031, 1996.
6. R. Barden, S. Stepney, D. Cooper. *Z in Practice*. Prentice Hall, 1994