Formal Methods and Safety Critical Systems in Practice*

Victoria Stavridou†

Andrew Boothroyd‡

Peter Bradley‡

Bruno Dutertre†

Linda Shackleton§

Robert Smith¶

October 13, 1995

Abstract

This article describes the state of the art in the practice of formal methods for safety critical computer systems. The work was done as part of the SafeFM [14] project and covers a tutorial introduction to specification and verification methods, standardisation activities, and examples of practical use as well as a unique survey of companies in the safety critical systems sector.

1 Introduction

The SafeFM project aims to provide guidelines on a cost effective approach to using formal methods in the development and assessment of high integrity systems. The project is sponsored under the SafeIT initiative by the UK Department of Trade and Industry and the EPSRC and is a collaboration between AEA Technology, the Flight Systems Division of GEC Marconi Avionics and the Formal Methods Group of Royal Holloway, University of London. We believe that SafeFM is at the cutting edge of realistic and practicable use of formal methods in the high integrity system sector.

*Work partially sponsored by EPSRC Grant No GR/H11471 under DTI Project No IED4/1/9013

†Department of Computer Science, Royal Holloway, University of London, Egham Hill, Egham, Surrey TW20 0EX, United Kingdom (email *bruno@dcs.rhbnc.ac.uk, victoria@dcs.rhbnc.ac.uk*)

‡AEA Technology, Thomson House, Floor 4, Risley, Warrington, Cheshire, WA3 6AT (email *andrew.boothroyd@aea.orgn.uk, peter.bradley@aea.orgn.uk*)

§Work carried out while with GEC-Marconi Avionics Ltd, Instrument Systems Division, Airport Works, Rochester, Kent, ME1 2XX

¶ESI Parque Tecnológicó de Zamudio,#204, E-48016, Bilbao, Spain (email *bob.smith@esi.es*) (Work carried out while with GEC-Marconi Avionics Ltd)

The focus of the project is on improving current practice rather than the theory of formal methods for high integrity systems. In particular, we aim to provide guidelines for a cost effective methodology for the specification, development and dependability assessment of complex, safety related systems. Maintaining the practical focus of the project, our work builds upon existing practices in the requirements specification, development and assessment processes. Our approach is centered on the integration of the development and assessment lifecycles thus supporting existing and emerging safety critical system standards.

In the context of identifying existing best practice, we have carried out an extensive survey and review of the role of formal methods in industrial practice. It is the objective of this article to document these findings. This seems a timely undertaking given the level of activity in this area.

We begin by identifying the kind of system that is typically involved in safety critical applications and surveying methods currently used for requirements specification and verification. The industrial perspective is introduced by a review of some practical formal methods applications. We consider possible links between risk assessment methodology and formal methods. We then survey existing and evolving standards that have a bearing on the industrial use of formal methods in the safety critical sector. This work is complemented by the results of a unique survey amongst companies in the sector, indicating the kinds of methods used as well as the levels of involvement in formal methods technology.

2 The nature of safety critical systems

Safety critical control systems can be diverse and complex. They are composed of various elements which have to cooperate. The software in such systems can be a critical component as computers are in charge of an increasing number of complex tasks. The performance, reliability and safety of the whole controller can depend largely on the qualities of the software.

A first issue in the development of software for safety critical applications arises from the diversity and complexity of the specifications. Properties of the controlled physical process and of the interface between the system and the physical environment influence the requirements and hence, a large number of different aspects has to be taken into account. A second issue is ensuring system safety. Safety requirements and reliability constraints have consequences on different aspects of the design such as the architecture, which themselves can have an important impact on the software. Another issue is the provision of evidence of the dependability of the system. This requires means of measuring, predicting and evaluating dependability.

In most cases, safety critical systems are embedded. They usually perform a complex control task and interact with the external world through sensors and actuators in order to control a physical phenomenon. They also often communicate with human operators, receiving commands and delivering information.

The equipment under control (EUC) is the physical process controlled by the computer system (e.g. a nuclear reactor) interacting with a control system which aims to reduce inherent risk (e.g. a trip system for a nuclear reactor). The environment can be defined as anything external to the equipment under control and the control system.

The exchange of data between the EUC and the controller is performed through an interface which may itself fail and which can thus be a source of further problems. Defining interface behaviour is a prerequisite to system conception. Furthermore, the interface often introduces new requirements; new functionality has to be added to the control task in order to deal with the vulnerabilities of the interface components.

So a typical specification consists of a triple <EUC, controller, interface>.

2.1 Equipment under control

In order to design a safety critical system, it is necessary to model in some way the physical process it controls. The variety of applications and disciplines involved makes it difficult to provide a model which can address this complexity in a uniform manner. However, all models can be roughly described by

- a set of variables representing pertinent aspects of the system, and
- a set of relations between these variables and of properties of individual variables.

The variables model the possible states of the EUC. The variables can be either discrete or continuous, reflect physical quantities (such as impact pressure, mach number, wing position) or give qualitative information (the plane is on wheels). Many systems are supervised by human operators who send commands (flight mode selection, stick position) and require information on the status of the system. Such commands and status reports can also be described as supplementary variables of the EUC.

Both functional and safety requirements are expressed in terms of these variables. Since the system has to control in some way a physical process, the functional requirements should be expressed in terms of the variables of the EUC. And since a hazard is a potentially dangerous situation occurring in the physical world, the safety requirements should also be expressed in terms of these variables.

Along with a set of variables, the description of the EUC must often include properties over these variables. Such properties can describe the evolution of the state, indicate the range of different parameters (e.g. the speed limits of a train or their braking characteristics[23]), etc. or relate different environmental variables (e.g. definition of Mach number in terms of pressure). As with the variables, the properties can be either quantitative (e.g. equations) or qualitative (e.g. if no gas then no flame in [73]).

These properties are useful during different phases of system conception. The requirements rely on understanding the possible behaviour of the EUC. Testing is also guided by this knowledge: test cases are devised to simulate the actual behaviour of the EUC, to be *real operating scenarios* [36].

2.2 Interface

Safety critical systems communicate with their environment through an imperfect interface. This interface allows a particular system to access a set of input and output variables but these variables are not, in general, identical to the variables used to model the EUC.

The input interface often includes unreliable components, which can be sensitive to external perturbations (such as electro-magnetic interference) and can fail. Even when sensors behave correctly, they can introduce a 'measurement error'. Another difficulty is that some parameters may not be directly measurable; they may only be estimated through indirect information (e.g. calibrated airspeed and true airspeed are obtained from indicated airspeed).

Similarly, the output interface can include unreliable components (such as electro-mechanical actuators) and therefore introduces uncertainty.

All these limitations of the interface have significant impact on the requirements of the system. First, the requirements should describe the relation between accessible variables (input and output) and the actual EUC's parameters the system has to monitor and control. Second, the unreliability of the interface often imposes supplementary requirements. New functionality to deal with the possible failures of the interface components is added to the initial control task of the system. Typically, the system is in charge of detecting failures in its interface components and has to respond to these failures appropriately. In fact, it appears that these new requirements are often much more complex than the control task itself and that a large part of the system is devoted to failure detection and fault-tolerance [61, 11].

Communication with the human operator can also be a significant function of safety critical systems. The operator can supervise the system by selecting commands and control modes and by receiving information from the system. This aspect should not be neglected. For example, inadequate error reporting (cryptic error messages) has been shown to have catastrophic consequences in a computerized radiation therapy machine [58]. Giving precise and readable error messages (or any relevant information) is an important contribution to overall safety that should be present in the requirements.

2.3 Current practices

The separation between physical variables and input and output variables of the system, is advocated by some authors [71] as a way of clarifying the requirements and making them reviewable. According to Parnas *et al*, the requirements document should include a specification of the functional requirements in terms of environmental variables, and a description of the relations between these variables and the inputs/outputs of the system. It appears that this separation is not always adopted and that requirements are often expressed in terms of the interface variables [11, 23].

However, it is clear that the ultimate goal of a safety critical system is to act on a physical environment and the interface is only the necessary intermediary to achieve this ultimate goal. The unreliability of this mediator can only complicate rather than facilitate the expression of requirements.

3 Requirements specification

Reducing the risk associated with a system can be achieved by reducing the probability and severity of an accident of the EUC. This requires identification and analysis of potential accidents and evaluation of their consequences. Primary safety requirements result from this analysis; they aim at maintaining the physical process under control in a safe state.

One task of the developers of a safety critical system is then to ensure that the safety requirements are satisfied. Architecture decisions are often guided by safety requirements. For example, redundancy may be necessary to protect the system from random hardware failures. Our main concern is the impact of the safety requirements on software specification and development.

A first issue is the derivation, from the safety requirements, of constraints or new functional requirements of the software. For example, redundancy of hardware may have significant repercussions on the software. Complex synchronization algorithms may have to be implemented. Detection of random hardware failures and protection against their consequences can also add complex requirements. A second issue is the measurement of the dependability of the software.

Capturing and analysing these requirements is the first phase of system development. It consists of producing the initial specifications of the system to conform to the client's requirements. These first specifications form the starting point of the software development process and are also an important means of communication between the client and the developers.

Errors in the initial specifications are difficult and expensive to correct if propagated to the design or implementation phases. In safety critical applications, they can also have a major impact on safety. Therefore, methods to provide correct initial specification and early detection of errors are of great importance.

The different approaches to requirements specification can be classified according to the degree of formality, precision, and unambiguity they offer, from informal descriptions in a natural language, through to structured semi-formal notations, to formal methods.

3.1 Informal specifications

The least precise method of describing systems requirements is simply to write them in a natural language, possibly decorated with diagrams, equations, and so on. As pointed out in [84], such a textual specification suffers from several problems:

- It is monolithic.

  You have to read the entire specification to understand it. This is due to the difficulty of structuring and partitioning textual specifications in understandable individual portions.

- It is often redundant.

  The same information is often present in different parts of the documents and this makes ensuring coherence difficult.

- It is ambiguous.

  A description in natural language can be interpreted differently by different people.

- It is difficult to validate.

  Since there is no support for performing rigorous coherence and other validity checks, informal review is the only way of validating the requirements. This can be technically challenging because of the many possible interpretations.

Despite these weaknesses, textual descriptions of requirements are still used in industrial contexts (cf. the SafeFM case study [11], or companies 4, 7 in Appendix).

3.2 Semi-formal specifications

In order to improve the quality of specifications and subsequent design process, semi-formal methods such as Yourdon [84], Core[68], SSADM [60], HOOD [43] have been proposed. These methods emphasize structuring of specifications. They often offer diverse graphical representations, such as data-flow diagrams, entity-relationship diagrams, mixed with natural language, pseudo-code notations, decision tables, and so on. All these notations allow hierarchical and structured system description.

The structuring mechanisms facilitate review and understanding of the system. Structured approaches also tend to reduce redundancy. Each of the different diagrams or notations gives a representation of one facet of the system. The validation of the requirements is also facilitated by the possibility of performing automatically some consistency checks (e.g. that every defined type is used somewhere).

These semi-formal methods are widely used in the software industry, and are supported by many CASE tools. The answers to the questionnaire (cf. Appendix) show that the majority of the participating companies use structured methods for requirements capture and validation. Some companies however feel that the supporting tools are poor.

Although structured specifications have many advantages over informal textual notations, they still present several limitations:

- The mixing of natural language with other notations allows scope for ambiguity.
- Some aspects particularly important in safety critical applications such as timing are dealt with in a very crude manner.
- The validation process is informal since it is not possible to prove properties of the requirements specifications.
- The semantics are not clearly defined.

3.3 Formal methods

The term *formal methods* has no universally accepted definition. In this review, we call formal method any notation with a clearly defined semantics which can be used to describe and reason about systems and their properties. This definition is very general and covers a wide spectrum of formalisms such as:

- High-level programming languages with formal semantics.
- Different models of parallelism and concurrency.
- "Classical" formal methods like Z and VDM.

In section 6, we present a sample of formal methods which have been deployed in industrial applications. For a more comprehensive review, see [7].

Formal methods are currently the most rigorous and precise approach to specifying and designing a given system and experience in industry suggests that formal methods can bring benefits to software development. Several cases of industrial applications borrowed from [23, 10] are presented in section 6.

Despite these examples of successful use, formal methods are not yet widely employed in the software industry. The questionnaire shows that just one company routinely uses a formal method. Despite several significant advantages, such reluctance to adopt formal methods is understandable given the practical problems that may be associated with their introduction. We briefly describe these issues below.

3.3.1 Benefits

As discussed in [23, 5] the benefits associated with the use of formal methods include:

- Requirements and specifications are unambiguous.

  The main reason for this is the clearly defined mathematical semantics of formal specifications. Communication between people involved in requirements analysis, specification construction, design and implementation via the formal specifications is clear and accurate. Therefore, errors due to misunderstandings are reduced.

- Analysis of safety requirements becomes easier.

  It is possible to use formal methods to make safety analysis much more thorough and precise. In [37] it is shown how fault trees, a major safety analysis technique, can be given formal semantics. Subsequent analysis of the system design allows both allocation of safety properties to various system components and verification that such components meet their safety specifications. Such a precise, well defined linkage between safety analysis and system development is not possible with informal or semi-formal approaches.

- Implementations based on formal specifications are easier than those based on informal ones.

  This is because formal specifications usually present precise tasks for implementation, whereas informal ones cannot do so. Such precision results from thorough problem-domain formal analysis.

- Correctness proofs can be carried out.

  Correctness proof is a powerful approach to verifying properties of implemented software systems against their specifications. It is especially appropriate for safety critical applications. Since formal specifications adopt mathematical notation, correctness proofs are made possible.

- Validation of requirements specifications becomes easier.

  Because of the precision of formal requirements specifications, every task specified can be precisely interpreted to allow clients to judge the correctness of the specifications more easily.

3.3.2 Limitations

From the industrial point of view [5], the limitations of formal methods include:

- Tools and environments to support the use of formal methods are not available.

  Tools to support the use of some formal methods do exist. However, none of them is powerful enough to support the whole formal methods activity, such as consistency checking of specifications, specification refinements, correctness proofs and so on.

- Formal specifications may not be accessible to clients.

  Clients are not necessarily accustomed to the mathematical notations used. To overcome this difficulty, one particular company [36], for example, had to rewrite a formal specification in natural language in order to communicate with railway signalling engineers.

- Formal methods cannot model all properties of a real world system.

  For instance, human-computer interfaces and some dynamic properties of systems, amongst others. In addition, many formal methods lack ways of describing time constraints.

- Correctness proofs are resource intensive.

  This is because of the intrinsic difficulty of performing correctness proofs automatically, so they have to be done manually, which is expensive.

- Development costs may increase.

  One reason for this limitation is the high level of initial investment needed to apply formal methods effectively.

- Formal specifications can still have errors.

  As mentioned previously, formal methods can help reduce errors due to misunderstandings. However, this does not mean they can guarantee that people will not make errors in formal specifications (e.g. syntactic errors and semantic inconsistencies). Almost no formal method in existence can provide automatic support for semantic consistency checking for formal specifications (a few tools supporting type checking do exist).

- Ways of incorporating formal methods into the whole life cycle of software development are not well defined.

  This is primarily a process management issue which requires an existing well defined framework in which formal methods can be embedded. Even when such a framework exists and all the process interdependencies are understood, many challenging technical issues still remain (construction of good quality specifications, refinement and effective verification amongst others).

4 Formal verification methods

Formal methods make it possible to prove or disprove facts about a system. Two entities are involved: the specification of a system, and a set of properties of the system to be proved. The specification has to be checked to verify that the properties hold.

Two categories of verification methods can be distinguished.

- In some cases, it is possible to construct from its specification, a model of the system. Different verification techniques can then be applied. If the property is expressed as a logical formula, the problem is to decide whether this formula is true on the model. This technique is called *model-checking*. A different method also can be used, where the property and the system are expressed in the same formalism. It consists of comparing two models, one obtained from the system specification and the other constructed from the property.

  The advantage of such model-based methods is that they can be fully automated. However, they can only be applied where it is possible to construct a model from the specifications. This requires that every specification possesses a finite model and that a construction algorithm exists. The most expressive specification languages do not fulfill these two conditions.

- The second category includes all the rewriting methods, based on axioms and deductive rules. In this approach, verification consists of proofs obtained by applying sequentially the deductive or rewriting rules.

  These deductive methods are purely syntactic; no model needs to be constructed. They do not suffer from the limitations of the model-based approach and can be used for very expressive formalisms. The inconvenience is that these methods are generally impossible to automate. Most of the proofs have to be carried out manually (in some cases with assistance from theorem proving tools).

4.1 Model based methods

4.1.1 System modelling

The first mathematical models of programs and computation are probably due to the work of Hoare and Dijkstra. They describe sequential programs as relations between an initial state – the initial value of the program variables – and a final state – the result of the computation. If the set of all <initial state - final state> pairs could be enumerated, an exhaustive search would suffice to prove any property of programs. Of course, for any interesting program, such an enumeration is impossible because the set of pairs is infinite.

Paradoxically, the situation is different in the *a priori* more complex domain of parallel programming. Finite models of parallel systems have been proposed. Such models can be constructed and proofs by exhaustive search are possible. These finite models are generally abstractions of systems. They concentrate on some logical aspects which are critical. For example, many models of communication protocols describe the synchronizations only; the values of the messages are not represented. But, due to the complexity of the synchronization algorithms, such partial models are still extremely useful.

Models for parallel systems cannot be as simple as models for sequential programs. The difficulty lies in describing the complex interactions between different sub-systems. Furthermore many applications (e.g. communication protocols) never terminate; there is no notion of a final state. Parallel systems are better described as reactive systems which continuously interact with their environment. Models must be able to represent each component as well as to describe the composition of several components.

The model of labelled transition systems is commonly used (see [3] for example) for this purpose. The possible interactions of a system with its environment are represented by a set A of symbols called events or actions. The sequences of actions the system can produce are described by an automaton. More formally, a labelled transition system is a triple (Q, A, T) where Q is a set of states, A is a set of events or actions and T is a subset of $Q \times A \times Q$. Elements of T are the transitions of the system. The notation $q \xrightarrow{a} q'$ stands for $(q, a, q') \in T$. It means that, when the system is in state q, it can evolve to the state q' by executing the action a.

The composition of two sub-systems is modelled by the product of two labelled transition systems. The product of (Q_1, A_1, T_1) and (Q_2, A_2, T_2) results in a new labelled transition system, whose states are elements of the cartesian product $Q_1 \times Q_2$. The actions and transitions of this new system can be defined in various ways, depending on the chosen semantics.

This model is very general and many other formalisms like Petri nets and process algebras [64, 9] are similar to transition systems. A Petri net can be considered as a transition system where the states are the markings of the net and each transition corresponds to the firing of a transition of the Petri net. The semantics of a process algebra associates a labelled transition system to each term of the algebra. Other models describing sequences of events such as linear temporal logic [72] have also a clear link with automata.

Extensions of state-transition systems have been proposed in order to describe real-time constraints. These extensions include timers which represent elapsed time. Transitions are enabled depending not only on states but also on the value of the timer. Special actions can manipulate the timers [69, 2].

Transition systems may be infinite. In this case some problems are not decidable. In practice, only finite transition systems (i.e. where Q and A are finite sets) are considered. Even in that case the size of the system is still an issue. For realistic applications, the number of states is so large that the model cannot be constructed. This is due to the explosive effect of the product operation. Composing n systems of k states yields a new system which can contain up to k^n states.

This problem can be partially solved by using equational representations. The principle is to represent each state and each event by vectors of variables – generally boolean variables – and to describe the transitions by equations. If states and events are respectively boolean vectors $(x_1, \dots, x_n)$ and $(y_1, \dots, y_m)$, the set of transitions can be represented by a boolean equation

$$f(x_1, \dots, x_n, y_1, \dots, y_m, x'_1, \dots, x'_n) = 1.$$

There is a transition from a state $q = (x_1, \dots, x_n)$ to a state $q' = (x'_1, \dots, x'_n)$ labelled by $a = (y_1, \dots, y_m)$ if and only if $(x_1, \dots, x_n, y_1, \dots, y_m, x'_1, \dots, x'_n)$ is a solution of the equation.

Such equational techniques supported by a compact representation of boolean functions [16] allow very large transition systems to be dealt with [17, 21]. However, these methods have their limits: a compact representation of a given logical function is often difficult to find, and in some cases no representation compact enough exists [59].

4.1.2 Model checking

All logics consist of a syntactic part – a set of formulas – and a semantic part that defines the meaning of formulas. Classically, the semantics describe mathematical structures on which formulas are interpreted. For a given structure and a given formula, the semantics also define the truth or falsity of the formula for the given structure. If a formula is true on a structure, it is said that this structure is a model of the formula.

The model-checking problem is simply to decide whether or not a given structure is a model of a given formula. For this decision problem to have solutions it is required that the model be finite or at least be finitely represented[1]. This approach has been comprehensively studied in the area of parallel and concurrent systems where finite models exist. To verify that a parallel program satisfies a property described by a formula f, a finite structure modelling the program is constructed and a model-checking algorithm is used to determine if this structure is actually a model of f.

In practice, the structure is usually a labelled transition system. This system can be given by the list of its states, actions and transitions or in an implicit equational form. The properties are themselves expressed in one of the numerous temporal logics. Different logics can be used but, as a general rule, the more expressive the logic, the more complex the model checking. For the latter reason, a particular logic, CTL is generally employed. This logic was introduced in [20], and it was proved that for this logic, model checking is of polynomial complexity. Roughly, the model checking algorithm consists of an exhaustive exploration of the automaton. The advantage of CTL is that the number of times a state has to be visited is bounded by the length of the formula. The verification algorithm is then very efficient.

Recent work has extended model checking to timed automata, allowing real-time aspects to be taken into account[2]. This of course requires the use of a temporal logic other than CTL, capable of describing timing constraints. However, the complexity of the algorithm is still reasonable and the method seems very efficient.

Despite the low complexity of the model checking algorithm, CTL is powerful enough to express many interesting properties. Model checking with CTL or variants of CTL, has been successfully applied in different domains such as protocol verification [76] and parallel program validation. The most impressive results have been obtained in the verification of large circuits, where model checking was combined with a powerful equational representation of circuits [17, 21].

It is also interesting to note that model checking has been applied to the verification of high level specifications of safety critical systems [4]. The functional requirements

[1]In the sense, for example, that a finite state automaton finitely represents an infinite regular language.

were expressed using the tabular notation of the SCR project [39]. A transition system was constructed from this specification. Safety requirements were written in CTL and model-checking was used to verify that the functional requirements satisfied the safety requirements.

The efficiency of the model checking approach has led to numerous implementations of model checkers. We can cite, for example, EMC [20] or XESAR [76]. These tools can give a counter-example in case a proof fails. This information can be useful for debugging the system design.

4.1.3 Comparison methods

It is sometimes difficult to describe a required property using a temporal logic like CTL. Such logics are suitable for expressing simple global properties such as "the system will reach such a situation" or "there will be no deadlock". Other properties are represented by very complex formulas or even worse cannot be written at all. A typical example is the property "the event a is observed at every even instant" which classical temporal logics cannot express [26]. Other more useful properties relating to the ordering of different events or actions can be very difficult to express. Rodriguez [76] gives the example of a communication protocol where messages are numbered from 1 to 4, and the property to verify is "messages are received in the order of their number". The specification of this property using a variant of CTL requires the conjunction of twelve non trivial formulas.

Such properties are much more easily described by automata. It can then be interesting to verify that a system, modelled by a transition system, behaves in accordance with a required property also described by a transition system. The verification consists of comparing two automata in order to decide whether their respective behaviours are compatible.

Many notions of compatibility between two transition systems can be defined. Language equivalence is a simple one: two transition systems are equivalent if they generate the same sequences of events.

When non determinism has to be considered, language equivalence is no longer sufficient. A deterministic automaton and a non deterministic one which generate the same language would be considered equivalent. To take non determinism into account, other compatibility relations are used. The simplest one is *bisimulation*. A definition of this equivalence can be found in [64].

Bisimulation is an important notion from which many other equivalences are derived. For example, *observational equivalence* [64] assumes that the systems under comparison can execute invisible actions. Two sequences of invisible actions cannot be distinguished. The behaviour of a system can be *abstracted* by replacing any chain of invisible transitions by a single one. Two systems are observationally equivalent, if their respective abstractions are equivalent for the bisimulation.

More general abstractions can be defined by considering that certain sequences of transitions cannot be differentiated by an observer [9]. Each abstraction defines a new equivalence relation between transition systems.

Equivalences not derived from bisimulation have also been proposed (for example, the *refusal equivalence* of CSP [42]). Sometimes, equivalences are not suitable and pre-order compatibility relations are used (this is the case when one wants to verify if one of the systems "implements" the other).

Some commercial implementations of these verification methods are available [8]. Efficient algorithms able to compute abstractions and verify the bisimulation equivalence exist. But the main drawback of these tools is the limited size of the systems they can manipulate.

4.2 Deductive methods

Deductive methods are based on a purely syntactic approach. Proofs are done by rewriting formulas of a particular language using inference rules (sometimes called rewriting rules).

This formalisation of the proof process was introduced by logicians to study the way mathematicians establish new theorems. Deductive methods are then encountered in logic based formal methods such as VDM [53] and Z [81], diverse temporal logics [62], algebraic specification formalisms like OBJ [33, 32] and process algebras like CCS [64]. Such methods are very general and can be applied to a large number of formal notations.

A deductive system contains a set of axioms and a set of deductive rules. The axioms cannot be proved, but describe basic facts from which new theorems can be established using the inference rules. In this framework, specifications of a system are formulas or sets of formulas of the language. To verify that the system satisfies a property f, one has to deduce f from the specifications and the axioms.

The difficulty is finding a sequence of application of the rules that leads to the formula f. In a few cases, proofs can be automatic and algorithms to decide whether or not a formula is a theorem exist. But most of the interesting logics are undecidable; in this case, proofs cannot be fully automated. However, tool assistance is available. Tools range from theorem provers based on heuristics to simple proof assistants which have to be completely directed by the user.

4.2.1 Deductive systems

Formally reasoning about a particular domain assumes a language and a semantics. For formal logics, the semantics defines the truth or falsehood of the formulas of the language for a given interpretation structure. A formula is said to be *valid* if it is true in any interpretation; it is said to be *satisfiable* if it is true in at least one interpretation.

A deductive system contains a set of basic formulas called axioms and a set of inference rules. The axioms describe a certain number of basic facts which are postulated. They represent basic properties of the domain.

A formula f is provable if there exists a finite sequence of formulas, ending with f, such that each formula is an instance of an axiom or follows from previous formulas by application of one of the inference rules. This sequence is the proof of f.

Three important properties are associated with deductive systems:

- *Soundness.* A deductive system is sound if every provable formula is valid. This is a vital property which justifies the use of syntactic manipulations in order to establish semantic properties. In practice, one often needs to enrich an existing sound theory by adding new axioms. It is important that such extensions preserve soundness [35, 13].

- *Completeness.* A deductive system is complete if every valid formula is provable. Although completeness is a desirable property, most interesting systems are not complete. For example, axiomatisations of ordinary arithmetic are incomplete[75]. Thus, any formal notation which includes a theory of the natural numbers is incomplete.

- *Decidability.* A deductive system is decidable if there exists an algorithm which can always decide the validity or invalidity of a formula. If a deductive system is decidable, automatic proofs are possible. Several temporal logics are decidable [26, 2] as is Presburger arithmetic[80] (a fragment of arithmetic without arbitrary multiplication). The existence of a decision procedure does not however guarantee automatic proofs.

It is also necessary that the procedure be of reasonable complexity. In the case of temporal logics, for example, the algorithms are so inefficient that they are impractical. In most cases, it is then impossible to prove properties automatically. Proofs have to be carried out manually. Even when proof assistants or theorem provers can be used, they require manual intervention. This is a resource intensive activity and it appears that when proofs are performed in a software design process, verification is an intricate task[23].

4.2.2 Theorem provers

Software tools have been developed to facilitate the task of performing formal proofs. They range from proof checkers which just apply the inference rules chosen by the user, to very complex theorem provers which rely on heuristics for discovering proofs. In this section we briefly describe three commonly used theorem provers.

OBJ

OBJ is a declarative language primarily designed for algebraic specification of programs and abstract data types [33]. The semantics of the language is based on abstract algebras and equational logic.

Specifications in OBJ consist of sorts, signatures and equations. Sorts are symbols representing different abstract types, signatures are sets of symbols denoting constants and functions, and equations describe relations between the different functions.

The equations can be treated as rewrite rules: an equation $t = t'$ is interpreted as the substitution rule "replace t by t'". These rewriting rules are used as an execution mechanism but they also allow properties to be proven [32].

Using this rewriting technique, OBJ is able to check if a given set of equations E – the specification – implies another equation e – the property to verify. If the rewrite system derived from E is *canonical* (any term t can be reduced in a finite number of rewriting steps to a unique canonical form t') and terminating, the proof is automatic. In other cases, the user has to guide the rewriting mechanism by selecting appropriate rules.

HOL

HOL is a variety of higher order logic based on type theory [35]. The language is similar to that of classical predicate calculus, with two principal extensions:

- Variables are allowed to range over functions and the arguments to functions can themselves be functions.

- Every term is typed.

The HOL theorem prover is implemented in ML [22] which is also used as a meta-language for manipulating formulas and proofs. The system is based on a few primitive inference rules which are particular ML functions.

A collection of types, constants, symbols, axioms and theorems is called a *theory*. Elementary theories like booleans, lists and numbers are pre-defined.

The properties to be proven are *sequents* of the form:

$$h_1, \ldots, h_n \vdash f,$$

where $h_1, \ldots, h_n$ are formulas called assumptions and f is a formula called the conclusion. If such a sequent has a proof, it is a *theorem* meaning that f is a consequence of the assumptions.

The proofs are entirely directed by the user and no heuristics are used. It is possible to proceed in a backward manner, starting from the *goal* property to be established, and successively decomposing it into simpler *subgoals*. Forward proof is also possible. Different ML functions called *tactics* and *tacticals* allow this process to be controlled.

Boyer-Moore

The Boyer-Moore prover is based on a first-order quantifier-free logic that permits recursive definitions [12, 13]. The syntax of the language is similar to Lisp.

The prover begins with an initial theory defining the Boyer-Moore logic. The user extends this theory, in a sound manner, by adding definitions and theorems. A definition defines a new function symbol and is only accepted if the system can prove that the function terminates. A theorem is accepted if the system can prove it using inference rules, from axioms, definitions and previously proved theorems.

The proofs are automatic. The prover applies sequentially a set of heuristics to a given formula in order to derive the proof. A formula is accepted as a theorem if the heuristics can simplify it to a value other than false. For complicated theorems, the user can guide the prover by a judicious choice of lemmas to be proved and by the order of their presentation.

4.2.3 Examples of applications

Many applications of deductive methods and theorem provers can be found in the domain of hardware verification [18, 32, 63].

Examples of use in industrial contexts are not very common. However, some software products of significant size have been formally verified (see section 6). These efforts were mainly due to the criticality of the applications. One is a railway control system [23, 36] specified and proved using Hoare's Logic and the B-method. Another is a shutdown software system for a nuclear power plant [23]. We can also cite the utilization of VDM to specify the safety properties of a system controlling the storing of explosives [67].

In all these applications, very little or no tool support was used. This is partly due to the absence of tools for the methods used at that time. It seems also that very expressive notations like VDM and Z render proofs and proof automation very difficult. Experience in developing a proof theory for VDM has shown the complexity of this task [30].

Theorem provers and proof assistants have however been used in other interesting applications. HOL has been used to mechanize proofs of CSP specifications [19]. The verification of concurrent programs can be supported by the Boyer-Moore prover [34]. The formal verification of a complex fault-tolerant algorithm for clock synchronization is presented in [78].

5 Formal methods and standards

The industrial use of formal methods in safety critical applications is on the increase, and it is generally accepted that formal methods have potential benefits. However, even there, the use of formal methods is still the exception rather than the rule. A significant and perhaps the most influential motivating force for the use of formal methods is likely to come from standards. Many safety-critical standards are now referring to formal methods as one of the techniques that should be used when the highest integrity of software is required. In this section we give some examples of such standards.

5.1 RTCA DO-178

The US *Radio Technical Commission for Aeronautics* (RTCA) produced a guideline on software consideration in airborne systems and equipment certification (DO-178A) in 1985. This does not explicitly recognize formal methods as part of accepted practice. However a completely rewritten guideline for the newly named *Requirements and Technical Concepts for Aviation* (RTCA DO-178B) [77] has now been approved since 1st December 1992. A very brief subsection entitled *formal methods* gives a general introduction to formal methods and mentions three levels of rigour: formal specification with (1) no proof, (2) manual proofs and (3) automatically checked or generated proofs.

It is now possible for a manufacturer following the DO-178B guideline to make use of formal methods in the context of aircraft certification, although it is incumbent on the manufacturer to justify its use. This less than enthusiastic endorsement is despite significant lobbying for a more rigorous approach to be adopted for the most critical software in aircraft systems.

5.2 UK HSE

The UK *Health and Safety Executive* issued an introductory guide and some general technical guidelines [44] concerning Programmable Electronic Systems (PES) in safety related applications in 1987. Two pages are devoted to software development and a further two to software change procedures. No mention is made of formal methods; it simply states that software should be of high quality, well documented, match its specification and be maintainable. It does list the necessary phases of software development and includes in these requirement specification, software specification, design, coding and testing, and system testing. It goes on to state that modifications to the software should be strictly controlled. The efforts of HSE are now mainly concentrated on the IEC standards.

5.3 IEC

The *International Electrotechnical Commission* (IEC) has been very active in this area. Following on from the 1989 Standards[46, 47], a further substantial undertaking of Technical Committee 65 has produced a draft Standard on the functional safety of safety-related systems. The standard is structured in three parts: Part 1[48] contains general requirements while Parts 2 and 3 refine these to electrical/electronic/programmable electronic systems (E/E/Pes)[49] and software[50] respectively. The standard is generically designed so as to be applicable across a variety of sectors. It classifies systems at integrity levels 1 to 4, 1 being the lowest.

Part 3 of the Standard makes specific mention of formal methods (CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z) and formal proof. Formal methods are recommended for the modelling, design, development and verification of software at integrity levels 2 and 3. For integrity level 4, the Standard deems formal methods as highly recommended which places an obligation on developers either to use them or to detail the rationale for not using them in the quality plan agreed with the assessor.

In addition, Part 2 recommends the use of formal methods for safety requirements specification and the design and development of E/E/Pes at integrity levels 3 and 4.

5.4 ESA

The *European Space Agency* (ESA) has issued guidelines for software engineering standards [27]. This suggests that "formal methods *should* be considered for the specification of safety critical systems" in the Software Requirement Document. The word "*should*" indicates strongly recommended practices in the document whereas "*shall*" is used for mandatory practices. A short section suggests that formal proofs *should* be attempted if practicable. Thus the use of formal methods is strongly recommended, but not mandated by the document.

5.5 UK RIA

The *Railway Industry Association* (RIA) consists of a number of interested organizations and industrial companies in the UK and has produced a consultative document on safety-related software for railway signalling [74]. It is a draft proposal that has yet to be ratified. It makes extensive references to the IEC65-WG9 standard [46]. Formal methods are mentioned briefly in several places in the document. In general, formal techniques such as proofs of programs and mathematical modelling are only recommended for the two highest integrity levels (graded from 0 to 4).

5.6 MoD 00-55 and 00-56

The UK Ministry of Defence has published two interim standards concerning safety. 00-55 [65] on the procurement of safety critical software in defence equipment, is split into two parts, on requirements (Part 1) and guidance (Part 2). Issue 1 was made available in 1991. The 00-56 standard [66] is concerned with safety management requirements for defence systems containing programmable electronics. Both standards have been extensively revised and are due to appear in the summer of 1995.

00-55 mentions and mandates formal methods extensively and has, therefore, caused much discussion and argument in the defence software industry as well as the software engineering community in the UK. The standard is currently in interim form. MoD set 1995 as the target date for the introduction of fully mandatory standards, but has now withdrawn a specific introductory date.

00-55 mandates the production of safety-critical module specifications in a formal language notation. Such specifications must be analysed to establish their consistency and completeness in respect of all potentially hazardous data and control flaw domains. A further fundamental requirement is that all safety critical software must be subject to validation and verification to establish that it complies with its formal specifications. This involves static and dynamic analysis as well as formal proofs and informal but rigorous arguments of correctness.

5.7 AECB

The Atomic Energy Control Board (AECB) in Canada commissioned a proposed standard for software for computers in the safety systems of nuclear power stations [1]. This was prepared by David Parnas who is well known in the field of both software safety and formal methods. His report formalizes the notions of the environment, the behavioural system requirements, and their feasibility with respect to the environment. It is based on the IEC standard 880 [45].

Since then, a new report which is much further from IEC880 and more focused on documentation has been prepared and is due to appear as an AECB "INFO" report shortly. This report is not used as a standard itself by AECB, but is used in the evaluation of standards and procedures submitted by Canadian licensees. These mandate the use of formal methods.

5.8 IEEE P1228

The P1228 Software Safety Plans Working Group, under the Software Engineering Standards Subcommittee of the IEEE Computer Society, is preparing a standard for software safety plans [51]. This is an unapproved draft that is subject to change. The appendix of an early draft dated July 1991 includes headings of "*Formal/Informal Proofs*" and "*Mathematical Specification Verification*" under techniques being discussed for inclusion. However a more recent version of the draft omits all mention of formal methods so it likely that the standard will make no specific recommendation concerning formal methods.

6 Industrial applications of formal methods

This section briefly describes some industrial applications of formal methods and aims to be indicative rather than exhaustive of where formal methods are currently being used. The examples are grouped according to the particular method used, and for each category, a brief introduction to the particular method is given. Further instances of inudstrial use can be found in the industrial survey appendix.

6.1 VDM

VDM (Vienna Development Method) is a denotational model-based approach [6, 53]. In VDM, specifications are constructed around abstract states which are models defined in terms of data objects such as sets, maps and lists. Operations on these state-like objects are specified by pre- and post-conditions. The pre-condition is a predicate over the initial state of the operation and can be used to limit the cases in which the operation has to be applicable. The post-condition is another predicate which specifies the relationship between the initial and final states of the operation.

Some industrial-scale projects using VDM for requirements analysis and specification are as follows:

- **Air Traffic Control**

  As part of its advanced programme to expand and develop the air traffic control system over the south-east of England, the Civil Aviation Authority is building a major new operation centre, the Central Control Function (CCF) facility, at the London Air Traffic Control Centre. Praxis Systems Ltd. has developed the CCF Display and Information System (CDIS) which forms part of this programme. CDIS is a vital component of the data entry and display equipment used by air traffic controllers. Praxis used VDM to do an abstract specification of the whole system and for some parts of the design [23].

- **Nuclear Power Plant**

  Rolls-Royce and Associates have been applying formal methods (mainly VDM) to the development of software for safety-critical systems, and nuclear power plants in particular, for a number of years [40, 41]. Their approach has proved successful and has produced many positive conclusions based on practical experience.

- **Control of Ammunition Stores**

  The formal methods group at RHUL examined an ammunition control system (ACS) currently used by MOD and the explosives regulations which it implements. They then constructed a formal model of the system and formalised the associated safety requirements. They also proved some properties of the specification using VDM [67].

6.2 Z

Z is a formal notation for system specification construction [81, 82, 54]. In Z, a specification is decomposed into pieces called schemas. Each piece can be linked with a commentary which explains informally the specification. Schemas are used to describe both static and dynamic aspects of a system.

Some industrial-scale projects using Z are as follows:

- **SSADM Infrastructure and Toolset**

  Praxis System Ltd. has developed a Computer-Assisted Systems Engineering toolset to support the use of the CCTA standard development method SSADM. Z was used to develop a formal specification of the toolset infrastructure. In the early stages of requirements analysis [15], Z was also used for concept exploration. Later, the project team developed guidelines for the use of Z with Object-Oriented Design methods.

- **Customer Information Control System**

  IBM's Customer Information Control System (CICS) is a large (slightly more than 500,000 lines of mixed language code), on-line transaction processing program which runs on some 30,000 installations around the world. IBM Hursley Labs Ltd. began to investigate new ways to improve future releases of CICS in 1981. IBM Hursley had responsibility for all aspects of CICS, including the development and upgrade of future releases. In cooperation with the Programming Research Group (PRG) at Oxford University, IBM Hursley used Z to specify several modules of the next commercial release of CICS [25]. CICS/ESA 3 Release 1 involved some 268,000 of new and modified lines of code, of which 37,000 were completely specified using Z, and about 11,000 were partially specified using Z.

- **Reusable Software Framework**

  In early 1987, Tektronix began an R&D project to design a reusable software framework for oscilloscopes [24, 31]. Results from this project were incorporated in 1988 into a specific product development effort of a successor family of oscilloscopes to a then current product. Product development involved a number of parallel but related development efforts involving three divisions and the research labs and frequent meetings with all the product engineers (40-50 people in total) were held. Z was used for constructing non-executable prototypes of designs and as a communication medium among the various members of the development team with different roles, backgrounds and expertise.

6.3 RAISE

RAISE (Rigorous Approach to Industrial Software Engineering) is a systematic development method [28, 70], which is a combination of aspects of VDM with well-researched areas of algebraic specification techniques and CSP [42]. It provides two notations, the RAISE Specification Language (RSL) and the RAISE Development Language (RDL). LaCoS (Large Correct Systems) is intended to be a demonstration of the feasibility of using formal methods for industrial software development, based on the RAISE method and tools developed in an ESPRIT I project [56, 57]. LaCoS companies using RAISE include Bull SA, Insiel Espacio SA, Lloyd's Register of Shipping, Matra Transportation SA, Spece Software Italia SpA and STC Technology Ltd. RAISE has been used successfully by the LaCoS group on two applications:

- **The Bell and LaPadula security model**

  The purpose being to investigate modelling security in RSL.

- **The Safe Monitoring and Control System**

  This system is a distributed controller, for use in mining and other industrial applications. RSL has been used to formulate the requirements.

6.4 B

B is a formal software development process for the production of highly reliable, portable and maintainable software which is verifiably correct with respect to its functional specification [29]. The method uses the Abstract Machine Notation (AMN) as the language for specification, design and implementation within the process. The method is supported over the entire spectrum of activities from specification to implementation by a set of computer-aided tools. B was used in the following industrial project:

- **SACEM Railway System**

  In 1988 GEC Alsthom, MATRA Transport and RATP started working on a computerized signalling system for controlling RER trains in Paris. Their objective was to increase traffic movement by 25% while maintaining the safety levels of the conventional system. The resulting SACEM system (partly embedded hardware and software) was delivered in 1989 and has since been controlling the speed of all trains on the RER Line A in Paris. The SACEM software consists of 21,000 lines of Modula-2 code. 63% of the code is deemed as safety-critical and has been subjected to formal specification and verification. The specification was done using B and the proofs were done manually using automatically generated verification conditions for the code and Hoare's logic [23].

6.5 FDM

The Formal Development Methodology (FDM) uses the Ina Jo language for expressing specifications and requirements, and Ina Mod for writing program assertions [55, 38]. Specifications are expressed in terms of a state and operations that may modify the state. Operations are described as a state transition relation. Correctness requirements are described by using "criteria" and "constraints". A criterion is an invariant on states. A constraint relates two successive states and thereby excludes some state transitions.

An application of FDM is as follows:

- **NIST Token-Based Access Control System (TBACS)**

  TBACS is a smartcard access control system with cryptographic authentication [23]. TBACS was developed by the U.S. National Institute for Standards and Technology (NIST) to replace traditional password-based systems. TBACS was formally specified and verified (with respect to a security model) using FDM.

6.6 CSP

CSP (Communicating Sequential Processes) [42] is a method designed for describing interacting processes. A process, which stands for the behaviour pattern of an object, is described in terms of the limited set of events selected as its *alphabet*, where the *alphabet* of an object is the set of names which are considered relevant for a particular description of the object. Communication between the various processes governs the interaction.

A case of using CSP in industry is as follows:

- **An Operating System Interface**

Praxis Systems Ltd. formally defined the interfaces between various components of a distributed factory control system using CSP [23]. A particular component for which Praxis had design authority was subsequently specified using VDM and the relationship between the VDM and CSP specification was established.

7 The industrial survey

The purpose of this section is to provide a review of existing specification practices, development methods and tools used in industry at present.

The information collated in this section is based on the results of a questionnaire answered by a number of major industrial software development companies. In addition, the section contains a summary of the status of software development in over 20 individual companies which are part of the GEC group. This information on its own in no way provides a complete picture of current practice in software engineering. But when compared against recent similar surveys it provides a set of consistent results.

The results of the questionnaire are summarized in tables 1 to 4. Table 1 presents the application domain of each company and gives the standards they follow. Table 2 gives an overview of the methods and tools used during the different development phases (requirements capture, software design and implementation). It also lists the main programming languages used in each company. Table 3 describes software testing procedures. Table 4 gives the different experiences of each company with formal methods and summarizes their feeling about formal methods.

This survey has shown that there are some common features used in the development of high integrity systems across many companies in a number of industrial sectors.

In particular the Yourdon structured analysis and design method features frequently. Object-oriented analysis techniques are beginning to be used more frequently now. In nearly all cases the methods are not used as standard, and in-house procedures are developed to suit the applications.

Ada is the most commonly used language and some form of subset is usually used. The size of software produced ranges from 1,000 to 100,000 lines of code.

Very little use of formal methods has been made and in some cases there has been no preparation for their use to date. Where formal methods have been used, the majority of the work has been case study exercises on a reasonably small scale often without tool support. Formal methods have yet to gain widespread acceptability. The extent to which it is practical to formalise the software lifecycle needs to be established. The practicalities of using them on large systems need to be addressed.

This review has highlighted a common core of current best practice which includes the use of Yourdon, HOOD, independent test teams, static and dynamic analysis of the source code and Ada subsets. The survey of software development within the GEC group showed that all the companies surveyed will, at some point in the future, need to consider an approach to using formal specifications in the development of their systems. The results of the survey also show that there are common points in the development lifecycle into which formal methods could be integrated. However the variety of development approaches must be taken into account so as to minimise disruption to current practice yet maximise the benefits to be obtained from using formal methods.

Company	Applications	Standard(s)
1	Avionics	IDS 00-56
2	Train Control	
3	Avionics	DO-178A
4	Command/Control	
5	Train Control	RIA 23 based on IEC WG 9
6	Air Traffic	IDS 00-56 and DO-2167A
7	Command/Control	
8	Avionics	DOD-STD-2167A
9	Avionics	DO-178A

Table 1: Applications and standards used

Company	Method(s) used	Tool support	Implementation language(s)
1	Yourdon[84]		Ada subset
2	B[29]	B-toolset	Ada, Modula II automatically generated
3	Core[68], Yourdon Jackson[52], PDL	Core Workstation Teamwork	Pascal, assemblers Ada
4	Informal, RTM Yourdon, SSADM[60]	Teamwork	Coral, Ada, C
5	Yourdon, Ward/Mellor[83]	SELECT	Assembler, C
6	RTM, OOA based on Schlaer/Mellor[79]	Teamwork	Ada automatically generated
7	Core, Yourdon HOOD[43]	IPSYS	Full Ada
8	Core, Yourdon, HOOD	Teamwork IPSYS	SPARK Ada subset
9	Yourdon	Teamwork	Ada subset

Table 2: Methods used during development

Company	Independent Teams	Tool Support
1	Sometimes	Yes
2	Yes	Yes
3	Independent reviewer	Yes
4	Yes	No
5	Yes	Yes
6	Yes	Yes
7	No	No
8	At functional level	Yes
9	Yes	

Table 3: Verification and validation

Company	Formal methods experience	FM views
1	Case studies	Tools are deficient
2	B used routinely	Good tools (B-toolset) Real-time aspects not managed Need skilled staff
3	Case studies	
4	None	
5	Formal specifications given by customer	Use of B envisaged
6	Case studies with Z and VDM	No further use envisaged
7	None	
8	Case studies	HOL and B envisaged
9	Verification of an algorithm with HOL Many case studies	

Table 4: Experiences with Formal Methods

8 Conclusions

In this paper we have presented the results of a substantial review exercise undertaken by the SafeFM project. The results of the review are manyfold. First, and according to the practical bias in the SafeFM investigation, we have identified the pertinent aspects of typical safety critical systems. This is necessary if we are to target our technology effectively. This has allowed us to identify ways of making more effective use of formal methods technology in the traditional application areas. The results of the industrial survey are very important for ensuring that the procedures and guidelines for integrating formal methods into current best practice are applicable in a wide range of companies and sectors. We found that the proposed SafeFM research programme is on target in terms of addressing existing practical problems of formal methods applications. These views are based not only on subjective assessment but also on objective information provided by a substantial number of industrial correspondents involved in the development of safety-related software and hardware.

Importantly, we have been able to identify possible novel ways of using the technology to solve some perennial problems in the application domain. For instance, we have already made inroads in combining formal methods with risk assessment methods such as fault tree analysis. Furthermore, it looks likely that a similar approach may be developed for other methods such as event trees, common mode failures identification and sneak circuit analysis.

Our overall assessment is that through both standardisation and technical necessity many companies are making the first cautious moves in the direction of formal approaches to some aspects of safety critical system development. Driven also by an increased awareness for quality management procedures in software production, the industry is moving towards the adoption of formal methods solutions where their analytical potential can aid the development of complex systems.

A Industrial Development Approaches for High Integrity Systems

A.1 Company number 1

General Methodology and Lifecycle Adopted

Real time embedded avionics systems are developed. The average size of software in their safety critical applications is 1,000 to 5,000 lines of Ada. IDS DEF STAN 00-56 is used to identify software components which are safety critical. Architectures generally adopted for safety critical applications include redundancy, dissimilar systems and fault tolerance.

Feasibility and Project Definition

Formal methods are not generally used although there have been several instances of use recently. Rapid prototyping is used in the project definition phase. They feel that the prototyping methods used would not interface well with a formal specification.

Requirements Capture

Several methods are used for requirements capture including Yourdon. Where formal specifications are used, syntax and semantic checkers are used. Proof checkers are also sometimes used depending on the method. They feel that these tools are not usually adequate. A weakness with these methods and tools is that the top level requirement is in English and there are no means of achieving automatic validation. The requirements specification is verified by formal review. Standard methods are normally used but methods can be chosen for a particular project if the standard method does not fit. System functionality is partitioned between hardware and software after the top level requirements capture. User interface and other system interfaces are defined by various interface documents.

Software Design

The requirements specification is usually transformed to the design using manual techniques. Compliance between the requirements specification and the design specification is demonstrated by formal review.
Where formal methods are used the transformation from formal to semi-formal methods is accomplished by using formal functions as a basis for the semi-formal modules. Syntax and semantic checkers are also used. Deficiencies in the methods and tools used are that they do not usually allow sufficient flexibility and there is no checking of the requirements coverage.

Implementation

The specification is transformed to code manually. The programming language generally used is Ada although some constructs are not allowed. The code is validated against the design specification by static and dynamic analysis.

Testing

Independent VV&T teams are sometimes used. The system is validated against its requirements and design specifications using test harnesses and predetermined results. Testbed and the VAX Performance Coverage Analyser are also used to ensure good coverage of the tests. Where a formal specification is used test data and expected results are determined from the formal specification. The test tools used do adequately perform their specified functions but there is no automatic check against the requirements.

Management Issues of Formal Methods

They have responded to proposals mandating IDS DEF-STAN 00-55 and have proposed a development environment including a toolset based on what is currently available. However, they do expect a greater number of tools to become available to provide the tool support that will be required to provide a cost effective approach.

A.2 Company number 2

General Methodology and Life Cycle Adopted

This company develops safety related real time software systems that are concerned with the protection of train movements, for both train separation and train routing. The average size of the safety related software in each system is in the range of 5 KLOC to 20 KLOC of Ada or Modula II.
Their general approach to the development of safety critical systems is as follows:

1. Specification using functional analysis, modelling and prototyping.
2. Formal design with proof of specification consistency and refinement corrections.
3. Unit and integration testing with branch coverage measurement.
4. Functional testing in simulated environment.

This approach follows the V lifecycle which has been modified to accommodate formal methods. The software is designated safety critical by a preliminary hazard analysis. Architectures are usually based on redundancy combined with information coding.

Feasibility and Project Definition

Formal methods are not used in this phase; functional analysis (SADT and finite state machines) is used instead. The ASA tool is used for prototyping.

Requirements Capture

Requirements capture begins with functional analysis and ends with formal specification using the Abstract Machine Model, based on the B Method. In general, the same method

is used for non safety critical systems except that the specification is not modularised. The B Toolset is used as a proof assistant to validate the consistency of the specification. It is felt that these tools perform their specified function well. The strengths of the method and tools are that they are efficient and easy to learn. The weaknesses are that real time aspects cannot be handled. There does not appear to be any limit on the size of problem that can be managed successfully using the method and tools.
Functional analysis performed from the supplier's point of view is used to partition the hardware and software. The constraints are performance and costs. The requirements specification is validated by prototyping with customer participation and by proving the consistency of the formal model. Methods such as SADT, finite state machines and B Method are used as standard but linking the methods is an in-house approach.

Software Design

The requirements specification is transformed to the design specification using stepwise refinement. Compliance between the requirements specification and the design specification is demonstrated by the refinement proof contained in the B Method. Transformation and verification between the different design stages is done in the B Method. Proof obligations are generated automatically by the B Toolset and the prover helps to do the proofs. The same method is used for non safety critical components of software except that the proofs are not done. The B Toolset adequately performs the specified functions. No semi-formal methods are used and so there is no transformation from semi-formal to formal methods. One of the main weaknesses of the method and tool is that it requires highly skilled personnel for effective use.

Implementation

The specification is transformed automatically into Ada or Modula II by the B Toolset. Some language constructs are forbidden as is the use of interrupts. The language and compiler are both adequate. Formal proof can be used to validate the code against the design specification.

Testing

Formal design allows a sensible reduction of testing especially unit tests. Functional validation remains mandatory. The set of tests may be produced during specification and prototyping. An independent verification team is used from a different department. Logiscope is used to dynamically analyse the code and to measure path coverage.
No static analysis tool is used, although when formal proof is used, the assertions produced by the B Method are used as input to their own in-house tool. A weakness of this method is that formal proof takes a long time and is not useful when the code has been automatically produced.

Management Issues of Formal Methods

Proposals have been submitted which ask for formal methods but not in connection with DEF STAN 00-55. They propose to use the B Method for all such work. They feel that the use of formal methods will increase progressively over the next few years and they are working to promote them. The cost benefit has been measured using design times and schedules.

A.3 Company number 3

General Methodology and Life Cycle Adopted

This company develops safety critical avionics such as flight controls and landing gear. The software for these applications is very large. Generally software is developed to DO-178A. Their architectures usually include a hardware backup to the software.

Feasibility and Project Definition

Formal methods have been used for case studies but have not yet been used for development. Rapid prototyping is done although the development rigs are produced early on.

Requirements Capture

Core and Yourdon are used for the requirements capture phase. These methods are used on both safety critical and non-safety critical phases. The Core method is supported by the Core Workstation and the Yourdon method is supported by Teamwork. The methods are adapted in-house to suit their applications. The software and hardware are partitioned by the systems team. The requirements specification is verified by review with the customer.

Software Design

Structured design is done either using Yourdon supported by Teamwork or Jackson type diagrams with no tool support. PDL is also used for algorithmic design. Verification and validation are done by design reviews. This approach is the same for both safety critical and non safety critical approaches.

Implementation

The specification is transformed to code by translating the PDL into code. The programming languages generally used are Pascal, Assembler and Ada. Subsets are used and defined internally. Code is not produced automatically. Code walkthroughs and testing are used to validate the code against the design specification.

Testing

Testing is carried out by the development team but an independent reviewer is used. The independent reviewer works for the same software manager as the development team. The system is validated against its requirements and design specifications by testing. The customer is present for some of this testing. LDRA is used for dynamic testing to show 100% statement and branch coverage. They are currently evaluating other tools such as Statemate for systems engineering and requirements capture.

Formal Methods

Formal methods are not currently being used. No proposals have been received asking for formal methods, although they expect to have to use them within 5 years.

A.4 Company number 4

General Methodology and Life Cycle Adopted

Systems developed include Command and Control, radar systems and sensor systems. These are typically 100K programs. The lifecycle model adopted is the waterfall phased development. Systems are not currently classified as safety critical.

Requirements Capture

Requirements are usually textual and RTM is being used on all new projects. Yourdon is also used, supported by Teamwork, and SSADM is used for data processing applications. Verification of the requirements is done by presenting the customer with their interpretation of the requirements and then discussing it. Prototypes of the human computer interfaces are built.

Software Design

Yourdon is used for software design. The transition from requirements to design and between design steps is checked manually. The Teamwork tool is not sufficient for automatic compliance checks.

Implementation

Pseudocode is used to transform the design into code. Programming languages used include Coral, Ada, C and less commonly, Pascal, Assembler and C++. No constructs are banned although there may be recommendations in the programmers' manual. Verification is performed by walkthroughs where both the code and test specifications are reviewed along with the RTM coverage.

Testing

The test team is independent to Software Manager level. The test team is responsible for writing and running the tests from the requirements. The use of Logiscope and LDRA testbed is being considered for the future.

A.5 Company number 5

General Methodology and Life Cycle Adopted

Systems developed are mainly train control systems. These are real-time embedded systems and cover all integrity levels. Software is developed to RIA 23 based on IEC WG 9 which uses integrity levels 0 to 4. The average size of software for safety critical software is 2-4K for class 4 and 8K for class 3. The lifecycle model is based on the V model.
Safety engineers carry out a hazard analysis to determine if the system is safety critical. Architectures usually use three channels of similar software. The systems are timing critical and fail safe.

Feasibility and Project Definition

Formal methods are not used although the customer is starting to produce formal specifications.

Requirements Capture

Yourdon is used for requirements capture. The Ward/Mellor style is used although they have adapted the method to meet their own needs. This method is used informally on the SELECT software tool and no real consistency checks are carried out. They have not invested in expensive tool support, since they have been looking for four or five years and feel that they are still not mature.

Software Design

The process of moving from data flow diagrams (requirements) to structure charts (design) is an intuitive step. They also use module definition tables which define the input and output of each module. These are then used by the design and test teams independently. Compliance between the requirements and design specification is achieved by reviewing the data flows one to one. This approach is used for both safety critical and non safety critical software.

Implementation

The module definitions are refined using a Program Design Language (PDL) into code. Assembler is generally used although C is used for medium integrity software. There are forbidden instructions although there are no procedures to ensure this. No automatic code generators are used and only controlled interrupts are allowed.

Testing

The test team is independent at team level only. A separate safety section validates the requirements. They have used SPADE for static code analysis. They also use Testbed and AFT (Automatic Function Tester) for their dynamic testing.

Management Issues of Formal Methods

They are looking for a seamless method from requirements capture to code. They do not feel Yourdon is appropriate and so intend to look at object-oriented methods. They are also looking at the B Method, since this appears to offer a seamless approach.
They are not using formal methods at the moment but expect to be using the B Tool within 5 years. They had previously looked at Z but see B as their best option.

A.6 Company number 6

General Methodology and Life Cycle Adopted

Systems include Air Traffic Management and Control Systems. These are mainly for the CAA and are safety related. The average size of the software for their safety related systems is 250,000 to 500,000 lines of Ada. Their general approach to safety related systems includes the use of FMECA. Failures are identified by brainstorming as are probabilities and consequences. They generally do what DEF STAN 00-56 and the IEC standards ask for but use their own internal procedures. The life cycle model used is based on DO-2167A. System architectures are generally fault tolerant.

Feasibility and Project Definition

Formal methods are not used but case studies have been done in VDM and Z. Rapid prototyping is used for user interfaces.

Requirements Capture

RTM is used for requirements capture. OOA approach based on Schlaer/Mellor is also used and documented on the Teamwork tool. The data flow diagrams of Teamwork are used to depict the operations on states. Hence tool support for this OOA method is not very good as few consistency checks can be performed but they feel the method itself is very good. They have adapted the OOA method to suit their applications.
The requirements specification is verified using prototyping and experience of current systems. The user interface is defined by prototyping and operator input.

Software Design

To produce the design they look at the requirements analysis and identify objects and operations of the data and then go straight to the preliminary design mapping between software requirements and the Ada structure graphs (the Ada structure graphs are an OOD extension for Teamwork).
RTM/Teamwork is used to demonstrate compliance between the requirements and design although completeness is more difficult.
Detailed design is done by completing the package bodies using the algorithms in the software specification.

Implementation

Programming language generally used is Ada. When the system is more critical some Ada features are banned such as tasking. The Teamwork Ada code generator is used and then the DEC Ada language sensitive editor is used to complete it. Code walkthroughs are used to verify the design to code transition.

Testing

A separate department is used for VV&T. The test cases and requirements are written from the requirements descriptions. Logiscope is used to provide test coverage data.

Management Issues of formal methods

Formal methods have not been used yet and they do not envisage that they will be. They will not be using them through choice.

A.7 Company number 7

General Methodology and Life Cycle Adopted

Systems developed include Command and Control. These are very large software systems and are generally classified as mission critical.

Feasibility and Project Definition

No formal methods are used.

Requirements Capture

Requirements are specified textually or using Core or Yourdon. The requirements are verified by review.

Software Design

No specific design method is used although HOOD supported by the IPSYS toolset has been used.

Implementation

Programming is carried out using Ada. No constructs are forbidden and the language and compiler are adequate.

Testing

No independent test team is used. Engineers write and test their own software. No test tools are used.

Management Issues of Formal Methods

They expect to submit proposals calling up DEF STAN 00-55 in the next few years although they have done no work in preparation.

A.8 Company number 8

General Methodology and Life Cycle Adopted

Systems developed include stores management and air data computers. Most systems are safety critical. The size of software for these applications is between 15,000 and 50,000 lines of Ada.
The general approach adopted is based on DOD-STD-2167A lifecycle. The system is identified as safety critical following a hazard analysis done by the systems team. Architectures adopted are based on fault tolerant, dual redundant systems.

Feasibility and Project Definition

No formal methods are currently used, although over the last five years many case studies and other formal development activities have been undertaken. Prototyping is sometimes used especially for complex timing sequences.

Requirements Capture

Requirements are specified using Core or Yourdon. The Yourdon method has been adapted to suit the applications. Tool support is provided by Teamwork. The interface requirements are specified in a separate document using a standard table layout. The requirements are validated by review both internally and with the customer.

Software Design

The requirements are transformed to a design specification by identifying data stores and the functions that update it. Both HOOD supported by the IPSYS toolset and structure charts supported by Teamwork are used.
Compliance between the requirements and the design is done by review.

Implementation

Ada PDL is used as the program design language which is then transformed to full Ada. The SPARK subset of Ada is used on all projects. No automatic code generators are used although a language sensitive editor is used. Code walkthroughs are used to inspect the code.

Testing

Independent testing is performed at functional validation level. Static analysis is used on the Ada source supported by the SPARK tool. Logiscope is also used to measure the coverage of dynamic testing. The statement and branch coverage required is defined to be 100%.

Management Issues of Formal Methods

A proposal has been submitted to use DEF STAN 00-55. It is proposed to use HOL to specify the top level system architecture and safety requirements and then to carry out a formal development using the B Method and B Toolkit. Over the next five years it is expected that the use of formal requirements specifications will become standard.

A.9 Company number 9

General Methodology and Life Cycle Adopted

Systems developed include flight control computers. All these systems are safety critical. The size of software for these applications is between 50,000 and 100,000 lines of Ada. The general approach adopted is based on DO-178A and the V lifecycle model is followed. The system is identified as safety critical following a hazard analysis done by the systems team. Architectures adopted are based on fault tolerant, triple redundant systems. Most systems use dissimilar software and hardware within the three channel system.

Feasibility and Project Definition

Formal methods have been used to model the redundancy management algorithms and to prove some properties about the system. This has been done in HOL. Many case studies and other formal development activities have been also been undertaken.

Requirements Capture

Requirements are specified using Yourdon. The Yourdon method has been adapted to suit the applications. Tool support is provided by Teamwork. The interface requirements are specified in a separate document using a standard table layout. The requirements are validated by review both internally and with the customer.

Software Design

The requirements are transformed to a design specification by providing more detail to the requirements specification and by giving an ordering structure to the data flow bubbles. Compliance between the requirements and the design is done by review using the compliance matrix.

Implementation

Ada PDL is used as the program design language which is then transformed to full Ada. Some features of Ada are not used. No automatic code generators are used. Code walkthroughs are used to inspect the code.

Testing

A test team is used which is independent up to software manager level.

B GEC Software Development Survey

B.1 Requirements Specification

Three main structured methods are used for requirements specification. These are Core, Yourdon and SSADM with Yourdon being the predominant method. Despite this level of commonality, 18 different tools are used to support the Yourdon method. This means that in many cases there are variations within the methodology being used. All the companies are using a functional approach to requirements specification although as tool support for object-oriented analysis becomes available this is being considered for use.

B.2 Software Design

Yourdon structured design is the primary method used for software design although HOOD and other object-oriented approaches are beginning to be used more often.

B.3 Implementation

Four main languages are used to develop code for high integrity systems. However, there are many variations of these four languages due to the target microprocessors used, the particular make of the compiler and the host operating environment.
Seventeen different Ada compilers are hosted on PCs, workstations and VAX computers. They are targeted to many variants of microprocessor including the M680X0 and Intel 80X86 families as well as Z8000, transputers and other less common microprocessors.
The use of C compilers showed even more variation with 25 different compilers being used on a similarly wide range of target microprocessors. Many of these projects are not high integrity. But this example does show how easy it is to have many different practices across different industrial sectors even when the companies belong to one management group.
The use of Pascal compilers shows a similar trend, with 18 different compilers targeted to a similar range of microprocessors. In addition, over 20 different microprocessor assembly languages are used as the primary coding language.

B.4 Testing

Software testing is supported by a number of types of test tools. This includes static analysers for control, data and information flow; dynamic analysers to provide test coverage data; and simulators to enable target code execution. Over 15 variants of test tools are used to support this phase of development.

B.5 Formal Methods

Nine different companies indicated some level of experience in applying formal methods. Most of the examples quoted are case studies aimed at developing expertise and have been performed without much, if any, tool support. The types of projects have included:

- Z to specify a three lane redundancy management system.

- Z to specify aspects of a Secure Computer System.
- Z to specify aspects of a Target Zone Validation System.
- VDM to specify aspects of a Storage Allocator.
- VDM to specify aspects of a Weapon Release Sequence.

References

[1] Proposed standards for software for computers in the safety systems of nuclear power stations, 1991. Final Report for contract 2.117.1 for the Atomic Energy Control Board, Canada.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[3] A. Arnold. Transition systems and concurrent process. *Mathematical Problems in Computation*, 21:9–20, 1988.

[4] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. on Software Engineering*, 19(1):24–40, January 1993. Special Issue on Software for Critical Systems.

[5] S. Austin and G. I. Parkin. Formal methods: A survey. Technical report, National Physical Laboratory, Teddington, Middlesex, UK, March 1993.

[6] D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

[7] R. E. Bloomfield. EWICS TC7 guidelines on the use of formal methods in the development and assurance of dependable industrial computer systems. Technical report, EWICS TC7, October 1992. draft version.

[8] D. Boudol, V. Roy, R. De Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. In *Automatic Verification Methods for Finite State Systems (J. Sifakis ed)*, number 407 in LNCS, pages 1–10. Springer-Verlag, 1989.

[9] G. Boudol. *Notes on Algebraic Calculi of Processes*, pages 261–303. Number F 13 in NATO ASI. Springer-Verlag, 1985.

[10] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, July 1993.

[11] T. Boyce. SafeFM case study report. Technical Report SafeFM-018-GEC-1, SafeFM project, January 1994.

[12] R. S. Boyer and J. S. Moore. *A Computational Logic.* ACM monograph series. Academic Press Inc., 1979.

[13] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Number 23 in Perspectives in Computing. Academic Press Inc., 1988.

[14] P. Bradley, L. Shackleton, and V. Stavridou. The SafeFM project. In F. Redmill, editor, *Proc. of Safety Critical Systems Symposium 93*, pages 168–176. Springer-Verlag, February 1993.

[15] D. Brownbridge. Using Z to develop a CASE toolset. In *Proc. of the 1989 Z user meeting.* Springer Verlag, December 1989.

[16] R. E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.

[18] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. Technical Report 91, University of Cambridge, Computer Laboratory, September 1986.

[19] A. J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, September 1990.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[21] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Conference on Computer-Aided Design*, pages 126–130. IEEE, November 1990.

[22] G. Cousineau, G. Huet, and L. Paulson. *The ML handbook.* INRIA, 1986.

[23] D. Craigen, S. Gerhart, and T. Ralston. An international survey of industrial applications of formal methods. Technical Report NISTGCR 93/626, NIST, March 1993.

[24] N. Delisle and D. Garlan. A formal specification of an oscilloscope. *IEEE Software*, pages 29–36, September 1990.

[25] I. Hayes Ed. *Specification Case Studies.* Prentice-Hall International, 1987.

[26] E. A. Emerson. *Temporal and Modal Logics*, volume B, pages 995–1067. Elsevier, 1990.

[27] ESA software engineering standards, February 1991. European Space Agency, Paris, France.

[28] D. Bjørner et al. The RAISE project – fundamental issues and requirements. Technical Report RAISE/DDC/EM/1, Dansk Datamatik Center, 1985.

[29] J.-R. Abrial et al. The B-method. In *VDM'91 Formal Software Development Methods.* Springer-Verlag, LNCS, 1991.

[30] J. S. Fitzgerald and R. Moore. Experiences in developing a proof theory for VDM. Technical report, University of Newcastle upon Tyne, April 1993.

[31] D. Garlan and N. Delisle. Formal specifications as reusable framework. In *VDM 90: VDM and Z*, pages 150–163. Springer Verlag, 1990.

[32] J. A. Goguen. OBJ as a theorem prover with applications to hardware verification. Technical Report SRI-CSL-88-4R2, SRI International, August 1988.

[33] J. A. Goguen and J. J. Tardo. An introduction to OBJ: a language for writing and testing formal algebraic program specifications. In *IEEE conf. on Specification for Reliable Software*, pages 170–189. IEEE Computer Society, 1989.

[34] D. M. Goldschlag. Mechanically verifying concurrent programs with the Boyer-Moore prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1023, September 1990.

[35] M. Gordon. HOL a proof generating system for higher-order logic. Technical Report 103, Computer Laboratory, University of Cambridge, January 1987.

[36] G. Guiho and C. Hennebert. SACEM software verification. In *Proc. of the International Conference on Software Engineering*, pages 186–191. IEEE, 1990.

[37] K. M. Hansen, A. P. Ravn, and V. Stavridou. Linking fault trees to software specifications. In *Analysis of requirements for software intensive systems.* Defence Research Agency Malvern, May 1993.

[38] M. E. Haykin and R. B. J. Warmar. SmartCard technology: New methods for computer access control. Technical Report Special Publication 500-157, NIST, September 1988.

[39] K. Heninger. Specifying software requirements for complex systems, new techniques and their applications. *IEEE Trans. on Software Engineering*, 6(1):2–12, January 1990.

[40] J. V. Hill. The development of highly reliable software – RR & A's experience for safety critical systems. In *Sofware Engineering 88, Conference Publication*, pages 169–172. IEE/BCS, July 1988.

[41] J. V. Hill. Software development methods in practice. In *Proc. 6th Annual Conference on Computer Assurance (COMPASS)*. IEEE, 1991.

[42] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[43] HOOD working group. HOOD reference manual. Technical Report WME/89-173/JB, European Space Agency, 1989.

[44] Programmable electronic systems in safety related applications, 1987. Health and Safety Executive, HMSO, Publications Centre, PO Box 276, London SW8 5DT, UK.

[45] Software for computers in the safety systems of nuclear power stations, 1986. International Electrotechnical Commission, IEC 880.

[46] Software for computers in the application of industrial safety related systems, August 1991. International Electrotechnical Commission, Technical Committee no. 65, IEC 65A.

[47] Functional safety of programmable electronic systems: Generic aspects, February 1992. International Electrotechnical Commission, Technical Committee no. 65, Working Group 10 (WG10), IEC 65A.

[48] Functional safety: safety related systems, part 1: General requirements, 1993. International Electrotechnical Commission, Technical Committee no. 65 (Industrial Process Measurement and Control), Sub-Committee 65A (Systems Aspects).

[49] Functional safety: safety related systems, part 2: Requirements for electrical/electronic/programmable electronic systems, 1993. International Electrotechnical Commission, Technical Committee no. 65 (Industrial Process Measurement and Control), Sub-Committee 65A (Systems Aspects).

[50] Functional safety: safety related systems, part 3: Software requirements, 1993. International Electrotechnical Commission, Technical Committee no. 65 (Industrial Process Measurement and Control), Sub-Committee 65A (Systems Aspects).

[51] Standard for software safety plans, February 1993. Draft J, P1228, Software Safety Plans Working Group, Software Engineering Committee, IEEE Computer Society, New York, USA.

[52] M. Jackson. *Systems Development.* Prentice-Hall, 1983.

[53] C. B. Jones. *Systematic Software Development using VDM.* Prentice-Hall International, 1986. 2nd Edition.

[54] S. King. Z and the refinement calculus. In *VDM'90.* Springer-Verlag, LNCS, 1990.

[55] D. R. Kuhn and J. F. Dray. Formal specification and verification of control software for cryptographic equipment. In *Proc. 6th Computer Security Applications Conference*, Phoenix, AZ, December 1990.

[56] LaCoS experiences from applications of RAISE, June 1991. LaCoS project report LACOS/CRI/CONS/13/V1.

[57] LaCoS experiences from applications of RAISE, March 1992. LaCoS project report LACOS/CRI/CONS/20/V1.

[58] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. Technical Report 92-108, University of California Irvine, Irvine, CA 92717, November 1992.

[59] H.-T. Liaw and C.-S. Lin. On the OBDD-representation of general boolean functions. *IEEE Transactions on Computers*, 41(6):661–664, June 1992.

[60] G. Longworth and D. Nicholls. *The SSADM Manual.* National Computer Centre, Manchester, UK, 1987.

[61] D. A. Mackall. Development and flight test experiences with a flight-crucial digital control system. Technical report 2857, NASA, November 1988.

[62] Z. Manna and A. Pnueli. The modal logic of programs. In *Proc. of the 6th ICALP*, number 71 in LNCS, pages 385–409. Springer-Verlag, 1979.

[63] G. J. Milne. The formal description and verification of hardware timing. *IEEE Transactions on Computers*, 40(7), July 1991.

[64] R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[65] MoD91. *The Procurement of Safety Critical Software in Defence Equipment.* Ministry of Defence, Directorate of Standardization, Glasgow, UK, 1991. Interim Defence Standard 00-55, Issue 1.

[66] MoD93. *Safety Management Requirements for Defence Systems Containing Programmable Electronics.* Ministry of Defence, Directorate of Standardization, Glasgow, UK, 1993. Draft Defence Standard 00-56, Issue 2.

[67] P. Mukherjee and V. Stavridou. The formal specification of safety requirements for storing explosives. *Formal Aspects of Computing*, 5(4):299–336, 1993.

[68] G. Mullery. CORE - a method for controlled requirements specification. In *Proc. 4th Int. Conf. on Software Engineering*, 1979.

[69] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18(9), September 1992.

[70] M. Nielsen, K. Havelund, K. Wagner, and C. George. The RAISE language, method, and tools. *Formal Aspects of Computing*, 1:85–114, 1990.

[71] D. L. Parnas, A. J. van Schouwen, and S. P. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, June 1990.

[72] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE symposium on Foudations of Computer Science*, pages 46–57, 1977.

[73] Provably correct systems. ProCoS I final report, Technical University of Denmark, DK-2800, Lyngby, Denmark, 1993. Esprit BRA 3104.

[74] Safety related software for railway signalling, 1991. BRB/LRU Ltd/RIA technical specification no. 23, Consultative Document, Railway Industry Association, London, UK.

[75] J. W. Robbin. *Mathematical Logic.* W. A. Benjamin Inc., 1969.

[76] C. Rodriguez. *Spécification et validation de systèmes en XESAR.* PhD thesis, Institut National Polytechnique de Grenoble, May 1988.

[77] Software considerations in airborne systems and equipment certification, December 1992. DO-178B, RTCA Inc., Washington DC, USA.

[78] J. M. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Trans. on Software Engineering*, 19(1):13–23, January 1993. Special Issue on Software for Critical Systems.

[79] S. Schlaer and S. J. Mellor. *Object Oriented Systems Analysis.* Prentice Hall, 1988.

[80] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the A.C.M.*, 26(2):351–360, April 1979.

[81] J. M. Spivey. Understanding Z: a specification language and its formal semantics. Technical Report 3, Cambridge Tracts in Theorical Computer Science, 1988.

[82] J. M. Spivey. *The Z Notation*. Prentice-Hall International, 1989.

[83] P. T. Ward and S. J. Mellor. *Structured Development for Real Time Systems*. Yourdon Press and Prentice Hall, 1989.

[84] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall International, 1989.