THE DESIGN AND VALIDATION OF SOFTWARE USED IN CONTROL SYSTEMS - SAFETY IMPLICATIONS

J Brazendale* and I Lloyd**

This paper gives an overview of software engineering and its role in safety. Strategies for software design are discussed, and the need for a system-level approach emphasised. HSE's initiative to introduce safety integrity levels for software and a framework for its implementation is outlined and comments invited on its form.

KEY WORDS: SAFETY, PROCESS CONTROL, SOFTWARE

INTRODUCTION

Software now plays a key role in the effectiveness and efficiency of UK industry with the process industries being no exception. However there is a growing concern that a more structured and rigorous approach to software development is required if safety standards are to be maintained

The main problem with software is that unlike hardware it is not wear and tear that is the main cause of failure but:-

1. Inadequate design
2. Inadequate control of maintenance changes.

The problems associated with software can be appreciated by considering the following incident involving a chemical batch reactor. (from (1) based on (2)). Programmers, designing alarm handling routines, were told that if an alarm occurred on the plant they were to design the safety system program so that all control variables were left as they were and to sound an alarm. On one occasion the process computer received a signal indicating a low oil level in a gear box just after reactant had been added to the batch. The computer acted as it had been programmed, it sounded an alarm and left the controls as they were. Unfortunately this

* HSE, Technology Division
Control Systems Group
Magdalen House, Bootle
Merseyside

** - Admiral Management Services Ltd
Camberley, Surrey.

included the cooling water control which remained "frozen" at a low level. The plant operators were too busy investigating the low oil alarm to notice that the reactor was overheating. An exotherm occurred and the reactor contents discharged to atmosphere.

In analysing the causes of the incident Kletz (2) found that although a HAZOP had been carried out at the design stage of the reactor those concerned did not understand what went on inside a computer and it was also clear that the programmers were unsure as to what was meant by "keeping the control variables as they were".

Clearly a systems level approach is required to ensure a safe design of computer controlled plant.

In recognition of this requirement HSE has published guidance on the safety of programmable electronic systems (3).

The principles behind this guidance have been reported extensively elsewhere (4) and therefore are not discussed in detail here. The key to the HSE guidance is a systematic design and assessment procedure based upon the following steps (see also fig. 1):-

STEP 1 ANALYSIS OF THE HAZARDS
STEP 2 IDENTIFY THE SAFETY-RELATED SYSTEMS (SRS)
STEP 3 DECIDE ON THE REQUIRED LEVEL OF SAFETY INTEGRITY FOR THE SRS
STEP 4 DESIGN THE SRS USING SAFETY CRITERIA FROM 3
STEP 5 CARRY OUT SAFETY INTEGRITY ANALYSIS
STEP 6 CHECK SAFETY INTEGRITY ACHIEVED

In the HSE guidance, software is classified as a systematic (ie. design related) failure and to overcome this failure mode it is recommended that attention is paid to improving the quality of the software design process. Sixteen checklists with over 100 questions relevant to software are detailed in the HSE guidance documents to act as an aid to critical appraisal during the design process. (See fig. 2 for an example of a checklist).

However although these checklists form a useful input into the design process it was recognised that further guidance in this area was required. HSE therefore commissioned a research project to strengthen its existing advice. The project has reached the stage where it has produced a framework for the development of safety-related software. This framework is discussed below but first a brief account of the principles behind software engineering is given.

SOFTWARE ENGINEERING

Software has become increasingly complex over the last thirty years. In order to cope with this complexity various software engineering techniques have evolved. Basically these techniques split the problem of producing software into a number of stages (Divide and Conquer) so that each stage becomes manageable. These stages form what is called the Software Lifecycle (fig. 3 from the STARTS Purchases Handbook (5) shows a typical software lifecycle).

The main phases are as follows:-

Requirements Specification

This stage involves specifying what you want the computer system to do. The textbooks say it should be accurate, complete, unambiguous and non-contradictory. A tall order in practice and more so with safety requirements because as we all know it is more what the system (machine, chemical plant etc) shouldn't do than should do that is important.

Software engineers have found the requirements phase one of the most difficult parts to get right. Lack of knowledge of plant and processes; division of responsibilities (electrical, chemical, computer engineers?) misunderstandings and ambiguities in written specifications have all caused problems. Formal methods are, at present, being suggested as the solution to this problem. Formal methods refer to techniques for writing the specification in a mathematical form so that you can prove (as in a mathematical equation) that the design meets the specification. Undoubtedly the very act of analysing the requirements mathematically will bring benefits but they will obviously not show up the unforseen. (This is the same problem as when one asks: Is a Fault Tree correct?). The key issue is obtaining confidence that the software has the right level of integrity for the application in question. Formal methods represent the highest level of rigour possible in software design; but that does not mean that a less rigorous approach could not result in the same level of integrity. As one might imagine this is an area of intense debate at the current time.

However one matter that is clear is that engineers have a vital role to play in requirements specification, after all it is they who know the plant best.

Architectural Design

After the specification has been written the developer then starts an interactive process of splitting the system into various modules which describe how the requirements will be met.

Detailed Design

This is the process of transforming the architectural design into a form which can be given to a programmer. Program design languages (PDL) are often used at this stage. These are a notation which describe requirements in a form halfway between a computer language and English. The idea is to show the programmer the preferred procedures for meeting the requirements specification without going into the detail of a particular language.

Coding

The programmer would then convert the above PDL into Basic, "C" or whatever. Increasingly the two jobs of specifying the PDL and coding are done by one person.

The choice of language does have an impact on safety. To quote Abelson and Sussman (6).

" .... A language is more than just a means for instructing a computer to perform tasks - the language also serves as a framework within which we organise our ideas about processes".

It is this philosophy that has led to the development of strongly-typed, high-level languages which allow the programmer to more easily describe the problem to hand; and which reinforce good programming habits. The overall aim being to reduce the chance of human error. This should be contrasted with assembler and other low level languages were there is the danger of becoming engrossed in register manipulations and forgetting the key features of the problem. However a disadvantage of high level languages is that they require compiling, a process that can introduce errors. The selection of a high or low level language has no easy answers at present.

Module Testing/Integration

As each major module is produced it will be tested for errors, joined to the next module and then the composite system tested for interface errors. This process continues until the whole system is assembled.

Acceptance Testing

The final system is subjected to a series of tests usually involving real data to demonstrate to the customer that the system works.

Verification and Validation

Experience has shown that if software errors are to be avoided a very structured approach to testing throughout the lifecycle is required. (By testing I mean a set of techniques to achieve "good quality" software and not just the exercising of the program with data).

These testing strategies can be divided into 2 types:-

(1) Verification
(2) Validation

Verification is testing to see that the results of a particular phase meets the requirements of the previous phase

OR Have we built the product right?

Validation is testing to see that the results of the whole project meet the requirements

OR Have we built the right product?

An example of verification is the mathematical proof involved in checking a formal requirements specification. The complete opposite in terms of sophistication (but commonly used) is "eyeballing" the code in a module probably as part of a formal review by a panel. Group and peer-review "inspection" techniques based upon Quality Assurance schemes are a development of this method.

Examples of validation techniques include the acceptance testing mentioned above and prototyping. Prototyping is the production of an early version of the software product. The customer is than able to see if the design is the same as he had in mind.

SOFTWARE AND SAFETY

As I hope is clear by now, software of itself is not a safety problem. It is the system (machine/chemical process) that causes the problem and therefore any safety analysis of software needs to consider the system as a whole. In particular the requirements specification cannot be developed unless the system as a whole is considered. Hence the need for the top level approach illustrated in fig. 1.

However having identified the hazards and produced the safety requirements specification, a number of strategies for the software design are possible including:-

Fault Avoidance
Fault Detection
Fault Tolerance

Fault Avoidance

Fault avoidance is concerned with making software as fault free as possible by the use of rigorous development methods throughout the lifecycle. Examples include:-

- Formal methods for the specification
- Avoid complexity (make it as simple as possible)
- Use structured design methods

The latter include graphical notations for describing the system

- Quality assurance (particularly the verification and validation steps)

Fault Detection

This takes as its premise than many faults if detected quickly can be prevented from causing harm (similar to fail-safe design philosophy).

Example include :

- Watch dog timers
- Self-checking eg. ROM checksum techniques

Fault Tolerance

This takes as its premise that no software is fault free and therefore certain techniques are needed to make the system tolerant to the (inevitable) faults that will occur. (NB. It could be argued that even if "perfect" software existed fault-tolerance is needed because software can be corrupted by for example electro-magnetic interference).

Techniques include:-

- Software Diversity (programming in two or more languages to avoid the same design faults)
- Error correcting codes (eg. in communication systems)
- Voting (eg. 2 out of 3 voting systems)

SOFTWARE DEVELOPMENT FRAMEWORK

The research project mentioned above confirmed to HSE that the spread of knowledge and use of appropriate techniques for producing safety-related software in UK industry was sporadic thus justifying the need for further guidance. It was decided that this guidance should be based upon a Framework that would define a range of safety integrity levels. The Framework will point to various approaches that could be used to meet those levels. This gives industry the flexibility to develop its own particular approach yet still meeting the standard of safety required.

The structure of the proposed framework is as follows:-

- Integrity Levels
- An integrated set of attributes and requirements which must be met for each stage of the lifecycle for each integrity level.
- An integrated set of generic methods and techniques for each set of attributes and requirements at each integrity level.
- Alternative sets of methods and techniques where it is clear that the same level of integrity can be achieved in a different way.
- A set of generic roles with pointers to the their responsibilities for the generic methods and techniques within each set.

The structure is illustrated in fig. 4. The key to the structure is the integrity level. Qualitatively, the higher the level (ie level 1) the lower the chance of the software causing a failure **of the system.** Each level will contain an integrated set of generic attributes and requirements that must be met for that level. Mapping onto these will be one or more sets of methods and techniques which if applied correctly will meet the requirement. More than one set will be proposed if there are alternative ways in meeting requirements. For example, rigorous specification using formal methods may meet the highest level, but the same level of confidence would be obtained by complete functionality testing where it is possible to do 100% path testing.

Clearly we do not yet have all the solutions to fill in the framework, but we can take advantage of this to identify areas requiring:-

Research
Techniques/Tools
Standards/Guidance.

Overall the intention is to reach a stage where any remaining doubts about the suitability of software for safety related systems is removed.

CONCLUSIONS

Achieving safety with software requires a structured approach that allows innovation and flexibility whilst still maintaining the appropriate level of safety integrity.

The framework described above is suggested as the way to move towards those objectives. Clearly there will be much debate on its precise form and aims (eg. number of integrity levels (3 or 4?); how do we match probability of failure with a given software development method?). HSE intends therefore to carry out extensive consultation. In that respect I would welcome your comments on the matters raised in this paper.

References

1. Levenson N G (1986). Computing Surveys Vol 18 No. 2 June 1986.

2. Kletz T (1983). Hazard Prevention (March/April 1983) pp 24-6.

3. "Programmable electronic systems in safety related applications: 2, General technical guidelines. HSE publication. HMSO London (ISBN 011 8839063).

4. Pearson J and Brazendale J (1988). I. Chem. E Symposium Series No. 110.

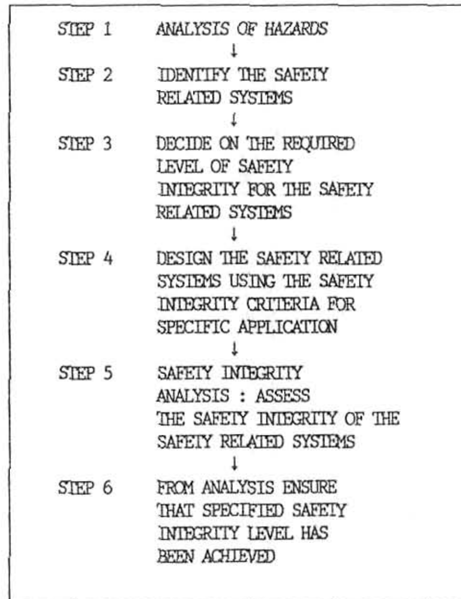
5. The STARTS Purchasers' Handbook, National Computing Centre, Oxford Road, Manchester.

6. Abelson H and Sussman G J (1985). Structure and Interpretation of computer programs. MIT Press, 1985.

FIG 1: OVERALL FRAMEWORK : DESIGN AND ASSESSMENT

KEY STEPS IN THE DESIGN AND *ASSESSMENT* PROCEDURE

NOTE:
1. Step 2 identies all safety related systems - PES and non-PES.
2. The HSE guidelines were primarily developed for those situations where one or more of the safety related systems was a PES.
3. The HSE guidelines can also be applied when none of the safety related systems are PESs.



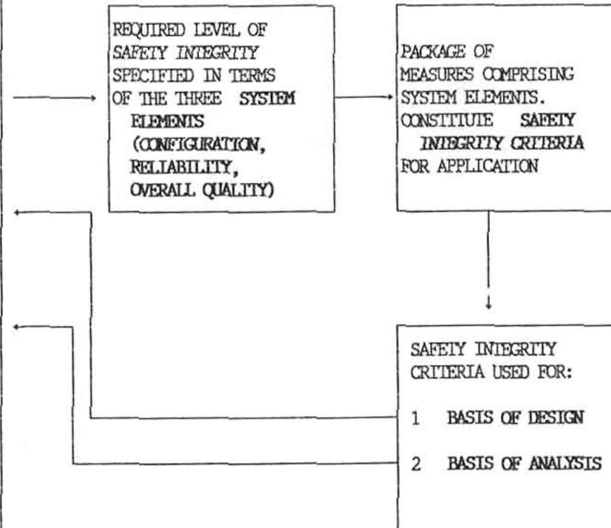


FIG 2 : CHECKLIST FROM PES GUIDELINES

Software specification

Item No	Item	Y N NA	Comments
10A.7	Are design reviews carried out in the development of the software specification involving users, system designers and programmers?	Y N NA	
10A.8	Is the final specification checked against the user requirements by persons other than those producing the specification before beginning the design phase?	Y N NA	
10A.9	Are automated tools used as an aid to the development of the software specification in	Y N NA	
	(i) documentation?		
	(ii) consistency checking?		
10A.10	Within the software specification, is there a clear and concise statement of:	Y N NA	
	(i) each safety related function to be implemented?		
	(ii) the information to be given to the operator at any time?		
	(iii) the required action on each operator command including illegal or unexpected commands?		
	(iv) the communications requirements between the PES and other equipment?		
	(v) the initial states for all internal variables and external interfaces?		
	(vi) the required action on power down and recovery? (eg saving of important data in non-volatile memory)		
	(vii) the different requirements for each phase of plant/machine operation? (eg start-up, normal operation, shutdown)		
	(viii) the anticipated ranges of input variables and the required action on out-of-range variables?		
	(ix) the required performance in terms of speed, accuracy and precision?		
	(x) the constraints put on the software by the hardware? (eg speed, memory size, word length)		
	(xi) internal self-checks to be carried out and the action on detection of a failure?		
10A.11	(i) Is there a software test specification?	Y N NA	

FIG 3 : SOFTWARE LIFECYCLE FROM STARTS HANDBOOK (5)

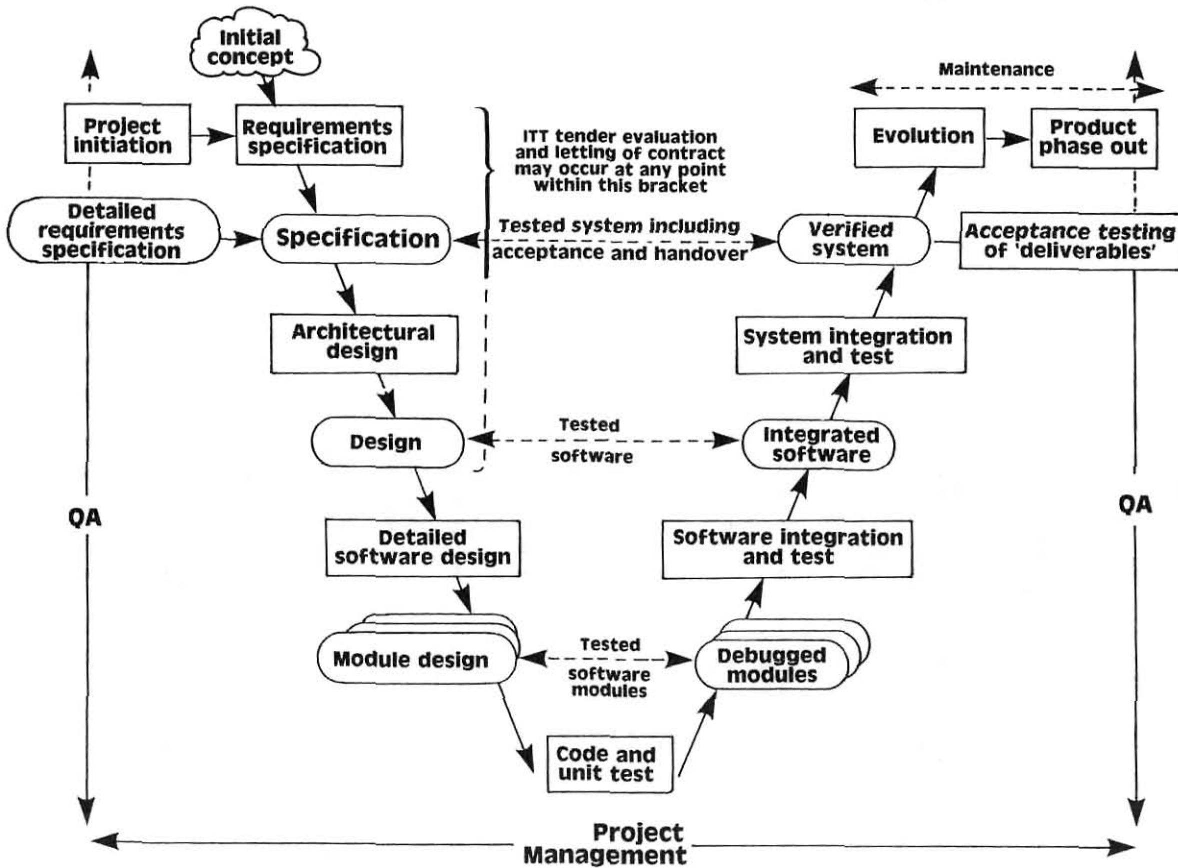
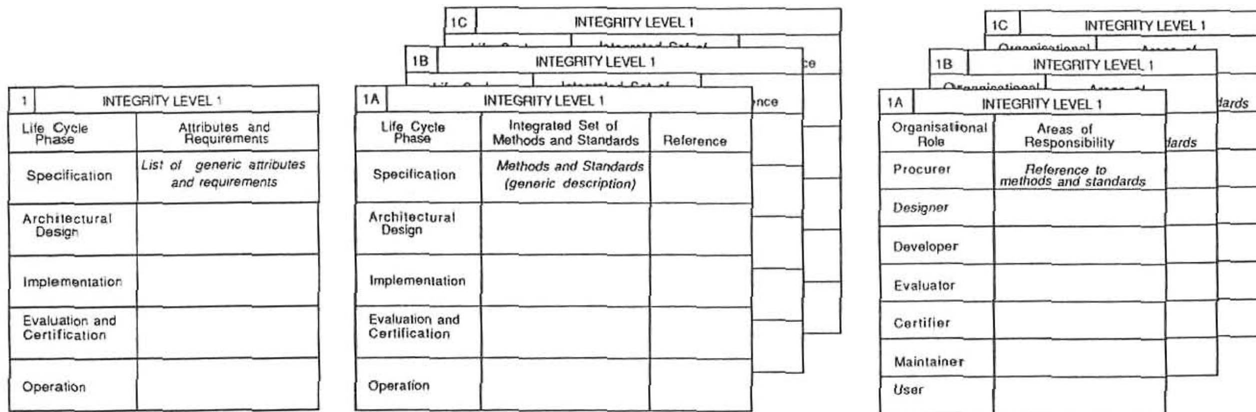


FIG 4 : SOFTWARE DEVELOPMENT FRAMEWORK



1	INTEGRITY LEVEL 1
Life Cycle Phase	Attributes and Requirements
Specification	*List of generic attributes and requirements*
Architectural Design	
Implementation	
Evaluation and Certification	
Operation	

1A	INTEGRITY LEVEL 1	
Life Cycle Phase	Integrated Set of Methods and Standards	Reference
Specification	*Methods and Standards (generic description)*	
Architectural Design		
Implementation		
Evaluation and Certification		
Operation		

1A	INTEGRITY LEVEL 1
Organisational Role	Areas of Responsibility
Procurer	*Reference to methods and standards*
Designer	
Developer	
Evaluator	
Certifier	
Maintainer	
User	

Framework is replicated for Integrity Levels 2 - 4