ASSESSMENT OF SAFETY-CRITICAL SOFTWARE IN NUCLEAR POWER PLANTS

David Lorge Parnas
McMaster University

G.J.K. Asmis
Atomic Energy Control Board

Jan Madey
Warsaw University

ABSTRACT

This paper outlines an approach to the design, documentation, and evaluation of computer systems. We believe that this approach allows the use of software in many safety-critical applications because it enables the systematic comparison of the program behaviour with the engineering specifications of the computer system. Many of the ideas in this paper have been used by the Atomic Energy Control Board of Canada (AECB) in their safety assessment of the software for the shutdown systems of the Darlington Nuclear Power Generating Station. The main elements of this approach are: a) Formal Documentation of Software Requirements: Most of the details of a complex environment can be ignored by system implementors and reviewers if they are given a complete and precise statement of the behavioural requirements for the computer system. We describe five mathematical relations that specify the requirements for the software in a computerised control system. b) Design and Documentation of the Modular Structure: Complexity caused by interactions between separately written components can be reduced by applying "Information Hiding" (also known as Data Abstraction, Abstract Data Types, and Object-Oriented Programming), provided that the interfaces are precisely, and completely, documented. c) Program Function Documentation: Software executions are lengthy sequences of state changes described by complex algorithms. The effects of these execution sequences can be precisely specified and documented using tabular representations of the program functions discussed by Mills and others. Further, large programs can be decomposed and presented as a collection of well documented smaller programs. d) "Tripod Approach" to Assessment: There are three basic approaches to the assessment of complex software products: (i) testing, (ii) systematic inspection, (iii) certification of people and process. Assessment of a complex system cannot depend on any one of these alone. We outline the approach used on the Darlington Shutdown Software which included systematic inspection, as well as both planned and statistically designed random testing. Certification of Software Engineers remains a difficult issue.

1 Introduction

The use of computers in nuclear power plants is increasing. The nuclear industry has lagged behind other industries in the application of this new technology, but will inexorably be caught up in the trend. New or replacement controllers, monitor and display systems, data acquisition and

interpretation systems, and “smart” instruments will unavoidably be software or firmware controlled. Computerised systems have been found to have substantial advantages over the technology that they replace.

Unfortunately, software products are undoubtedly among the most complex artifacts made by humankind. They are also among the least trustworthy. These two facts are clearly related. Errors in software are not caused by a fundamental lack of knowledge on our part. In principle, we know all that there is to know about the effect of each instruction that is executed. Software errors are blunders caused by our inability to fully understand the intricacies of these complex products.

Software professionals come to accept design errors as commonplace and unavoidable. Many believe that we cannot fully understand the things that we build because they are too complex. It is considered normal for a computer system to fail frequently when first installed, only becoming reliable after a long sequence of revisions.

In safety-critical applications this attitude is inappropriate. Such systems must work when needed; we cannot wait for evolution-during-use to bring the reliability up to an acceptable level. The nuclear industry needs demanding standards, standards that specify precisely what can be expected of software that must be reviewed by a Regulatory Authority. This paper presents a model for software development and analysis that has evolved over a period of 18 years and was recently applied in the licensing of a Canadian nuclear power station.

Ontario Hydro's Darlington Nuclear Power Generating Station provides an illustration of such an application. Current Canadian design practice requires three systems that are capable of shutting down the reactor if an incident occurs. The main reactivity control system is responsible for adjusting the control rods to control the power produced. This system monitors the plant's performance and can cause power generation to stop if anything goes wrong. In addition, there are two safety-systems; their only job is to shut the system down if anything abnormal occurs. During normal operation the two safety systems have no function other than collecting and displaying data to be used in determining if the plant is functioning normally.

This “triple redundancy” approach is based on two observations. First, the reactivity control system has a difficult task and it will not engender confidence in the early days of its use. The shutdown systems, in contrast, have a very simple task and it should be possible to get them right. Second, two shutdown systems, of diverse design, are used so that even if both the reactivity control system and one of the shutdown systems fail, the plant will be shutdown when needed. The three systems use disjoint sets of sensors and control mechanisms. A triple failure is considered to be very unlikely.

In earlier Canadian reactors the shutdown systems were not computerised. They were constructed of analogue devices and relays. They were simple, easily studied, and engendered a great deal of confidence. However, this technology requires large amounts of equipment. More important, this old fashioned technology could not perform sophisticated checking procedures. Both Ontario Hydro, and the Atomic Energy Control Board of Canada (AECB), felt that safety could be improved if a computerised system was installed at the Darlington Station.

At the Darlington Station, the two shutdown systems were kept diverse in several ways. First, they used different sensors, sometimes monitoring different physical quantities. Second, they used different shutdown mechanisms. Third, they used different computers. Fourth, the software was written by independent teams. It was hoped that this would bring the probability of a common-

mode failure to an acceptably low level. It is impossible to know how successful this was.

However, when the AECB examined the software for these systems they became uncertain of the adequacy of the software. The computer systems were replacing simple, relatively easily inspected, systems; they turned out to be far more complex than the systems that they replaced. Functions that can be performed by well-understood hardware devices, were now being performed by software routines that were much harder to understand. Further, the component programs could interact in subtle ways that had no parallel in the analogue systems that they replaced. Sequential algorithms, with shared data, had replaced components that were physically separate and worked "in parallel". The AECB felt that the software of the safety-systems was of sufficient complexity to require outside help; they turned to Queen's University for advice on the advisability of licencing a plant whose safety depended on these programs. Further discussion of the Darlington situation can be found in [1,20].

In this paper we outline an approach to design and documentation of software intended to help both designers and reviewers deal with the complexity that is present in software products. If the software is to play a role critical to safety, it is essential to restrict the complexity to allow complete understanding and thorough analysis. We believe that the techniques described in this paper can allow the use of software in well-understood safety-critical applications. Many of the proposals in this paper have been used by the AECB in their assessment of the software for the shutdown systems of the Darlington Station. Much of the preliminary work was developed at the United States Naval Research Laboratory (NRL) and applied to an experimental redesign of the Onboard Flight Program for the A-7E aircraft. The techniques described in this paper are based on a theory of software design and documentation that has been described more extensively in [21]. Another discussion of the role of documentation can be found in [7].

2 Documentation of Software Requirements

2.1 The Case for Formal Requirements

Modern computer systems must interact with a "real world" that can never be completely understood and completely described. If those that had to design, implement, and review the Darlington Shutdown Software had to know everything that could be known about the plant, the task would be of such complexity that nobody could tackle it. Luckily, only a small portion of that information is actually relevant to understanding the shutdown systems. This information can be summarised in a system requirements document that describes exactly what properties the shutdown system must have. The availability of this document should make it unnecessary for the system implementors to understand the highly complex environment in which it will be used. The document can be seen as a barrier that separates the complex details about the outside world from the complex details about the design of the computer system.

It cannot be expected that all the programmers working on a safety-critical software system will be intimately familiar with the requirements of the application. Even those who are specialists in the application may have differing ideas of the functions that a safety-system should perform. For this reason, it is essential that the plant engineers agree on, and precisely document, the requirements that the software must satisfy. If this is not done, or if the document is not complete and precise, the designers and programmers will be forced to deal with the complexity of the outside world and they will probably make errors as a result.

For similar reasons it is essential to any software quality assurance (QA) group that they be given a precise statement of the requirements that the software must meet. For example, the Queen's group, while it had some expertise on software, had little information about Nuclear plants. The software specialists at Queen's could not be expected to determine whether or not a program would perform correctly unless they were told exactly what that would mean.

Years of experience with documentation written in broad variety of natural languages have shown natural language to be inadequate for the task of precise requirements specification. There are always unsuspected ambiguities in natural language specifications. For example, consider the simple statement,

"Shut off the pumps if the water level remains above 100 meters for more than 4 seconds."

Although this sentence appears clear, if we remember that the water level could be varying over the 4 second period, we can find at least three reasonable interpretations:

1) "Shut off the pumps if the mean water level over the past 4 seconds was above 100 meters".

$$\left[\int_{T-4}^{T} WL(t)dt\right] \div 4 > 100$$

2) "Shut off the pumps if the median water level over the past 4 seconds was above 100 meters".

$$\left\{MAX_{[T-4.T]}\left[WL(t)\right] + MIN_{[T-4.T]}\left[WL(t)\right]\right\} \div 2 > 100$$

3) ."Shut off the pumps if the 'rms' water level over the past 4 seconds was above 100 meters".

$$\left(\int_{T-4}^{T} WL^2(t)dt \div 4^{1/2}\right) > 100$$

The software can only implement one of these requirements, since they are distinct. The statement is not precise enough to say which is actually meant. The more serious fact is that an analogous ambiguity was not noticed by those working on a safety critical system and a misunderstanding resulted. There is a fourth possible interpretation of the sentence.

4) "Shut off the pumps if the minimum water level over the past 4 seconds was above 100 meters".

$$MIN_{[T-4.T]}\left[WL(t)\right] > 100$$

This fourth interpretation corresponds to the most literal interpretation of the statement that we started with. Unfortunately, with sizable rapid waves in a tank, the water level could be dangerously high but not trigger a control system built to implement the fourth interpretation. We have seen two, independently written, versions of safety-critical software that were based on the fourth interpretation of such a sentence. This type of ambiguity is very common in informal documentation.

Whenever we allow ourselves the luxury of unrestricted natural language we run the danger of unnoticed ambiguity.

We use this example to illustrate our position that the first step in keeping control of complexity is the production of an extremely precise specification of the requirements that a system must satisfy. In mature areas of engineering, ambiguities of this sort are resolved by the use of precise mathematical descriptions of the product. Unfortunately, the complexity of computer systems, as well as the lack of professional engineering traditions in the field, have resulted in *ad hoc*, anthropomorphic descriptions of computer system behaviour. Mathematical specifications offer the promise of concise, precise description. While computer science approaches to formal requirements specification have not met with great success (especially in dealing with real-time systems), approaches based on control-theory models have proven practical [5,6].

We propose[1] that the documentation of software requirements be composed of at least two distinct documents. The *system requirements document* treats the complete computer system as a "black-box". It begins with a description of the environment, including a set of environmental state variables of concern to the system's user which are to be denoted by mathematical variables. It describes any relationships between the values of the environmental variables that result from physical (or other) constraints and then documents the additional relationships between environmental variables to be established by the system of interest. It is valuable if the methods used to define the system requirements document are also applicable to hardware systems such as those built of analogue components and relays. The analysis of a network comprising computers and other components is much easier if the same notation and concepts are used throughout.

The *system design document* identifies the computers within the computer system and defines their communication with the environment. It describes each computer, with emphasis on the peripheral devices. Each of the input and output registers is denoted by a mathematical variable; the document defines the relationship between the values in the input and output registers and the values of the environmental state variables.

The *system requirements document* and the *system design document* determine the software requirements. Together, these two documents may serve as the *software requirements document*. However, it is often the case that the requirements expressed in this way do not fully determine the software behaviour. There may be several, quite easily distinguished, software products that would satisfy the system requirements equally well. In such a situation, many project managers may request an optional document, the *software function specification*, which records additional design decisions, and describes the exact behaviour of the software. A software function specification is particularly important in multiple computer systems.

The system specification is implementation independent; in theory, the functions could be provided by digital computers or by relays and analogue technology. The software function specification is implementation independent in another sense; the internal structure of the software is still open. Neither the division of the software into modules, nor the algorithms to be used, are specified.

The contents of these documents are representations of the five mathematical relations listed below.

[1] The methods proposed in this section have been discussed more completely in [21].

1) NAT(m^t,c^t)

2) REQ(m^t,c^t)

3) IN(m^t,i^t)

4) OUT(o^t,c^t)

5) SOF(i^t,o^t)

m^t is a vector of time-functions representing the behaviour of the monitored physical quantities. c^t is a similar vector describing the behaviour of the controlled physical quantities. REQ describes the behaviour that should be allowed by the system to be built. i^t represents the behaviour of the input registers of the computer(s) and o^t represents the behaviour of the output registers of the computer(s). IN and OUT describe the behaviour of the input and output devices respectively. NAT describes the behaviour permitted by natural forces in the system. SOF describes the behaviour of the software. It must be such that the actual behaviour of the system is a subset of that permitted by (1).

This approach offers rigour comparable to that used in traditional engineering. It is more fully described in [6,21,23]. When there is a great deal of information to be presented, we use a tabular form of expressions to define the mathematical functions. Experience has shown these tables to be more practical than more conventional mathematical notation because the functions usually implemented by computerised systems can have a great many points of discontinuity and the discontinuities can occur at arbitrary points in the domain. We have found the tabular format to be an excellent vehicle for communicating between the system designer and the software engineer. In earlier experiences we found that even those who are not professional engineers, for example pilots of military aircraft, and telephone system supervisors, could read the tables.

It is often practical to summarise the relevant aspects of the history of the observed variables by defining modes, classes of system states corresponding to classes of histories. It is then possible to describe the requirements by a simpler relation. The domain of that relation consists of ordered pairs, the first element of which is the mode, and the second element of which is current value of the observed variable; the range of that relation consists of possible values of the controlled variables at an instant in time. This relation specifies the acceptable values of the controlled variables, given the current value of the observed variables and the current mode.

Figure 1 is a software requirements document for a system that might be installed in a donut shop. Although very concise, the precision of this example allows a careful reader to see that it cannot be satisfied without placing a restriction on the relative timing of events.

Some may find documents of this sort unduly complex and hard to read. In part, this is because the notation is not familiar. However, in part it is because the reader is not considering the alternative. If this document were not available, system implementors would have to study far more complex documents describing the application environment.

2.2 Design and Documentation of the Modular Structure

Once we have filtered out, to the extent possible, the complexities inherent in the application environment, we must turn our attention to controlling the complexity within the system itself. It has long been recognised that the construction of modular software is the key to control of software complexity. However, there are varying approaches to the decomposition of software into modules.

If modularisation is done incorrectly, the complexity of the software is not reduced; if there are many interactions between the software modules, the fact that the modules are small will be of no avail.

The most common way to divide software into modules is to identify steps in the processing, usually subroutines, and consider them to be appropriate work assignments for members of the implementation team. This approach results in modules that are strongly coupled because they must exchange data through complex data structures; these modules often share devices and other resources. Changes in the meaning of the shared data structures, or the way that the shared resources are used, affect many modules. A thorough understanding of how the program works requires a simultaneous study of many modules, sometimes the whole system.

The computer science literature has long contained an interesting alternative to the usual approach. Modules are collections of programs which, together, hide (or abstract from), a data structure, device interface, mathematical model of some physical system, etc. These modules are defined first, their interfaces precisely specified and then used to write the programs that describe the processing. Those programs are much simpler as a result. Changes in the meaning of data structures, new devices, or improved modules result in changes in only a few of the modules. The understanding of the program can be done one module at a time. Variations of the basic idea, known as information hiding, among them object-oriented programming and abstract data types, are now widely accepted in the programming methods community [2,3,4,11,12,13,15].

Unfortunately, the use of information hiding will not have the desired effect unless it is accompanied by appropriate documentation of the module interfaces. The purpose of information hiding is to manage complexity by making it unnecessary for the implementors of one module to know the details of implementation of other modules. However, if the only way to find out what another module will do is to read the code, the advantage quickly disappears. Moreover, unless the interfaces are precisely documented, the changes made during maintenance will result in a deterioration of the module structure and the reintroduction of complexity. Approaches to the documentation of module interfaces are discussed in [18].

Often, safety-critical software must be presented for review by a regulatory agency, which is responsible for analysing it and showing that it conforms to the specifications. The requirement that the software be reviewable makes a modular structure based on information hiding much more desirable, almost a necessity. If, to really understand one module, one must constantly refer to another module, the process becomes so complex that nobody has much trust in the results.

We suggest that information hiding be used in all software. However, it is not easy to add information hiding to a program that has already been designed in the conventional way. With old programs, the costs of rewriting and revalidation must be weighed against the benefits of information hiding. In AECB's review of the Darlington Shutdown Software, we felt that it was too late to insist on the application of information hiding. Although information hiding could have greatly simplified the QA process, it was postponed until the next major revision of the software.

2.3 Program Function Documentation

As Mills and several others have pointed out [4,8,9,10,14,17], the effect of any deterministic program on its data structure can be described by a mathematical function whose domain consists of the set of starting states for which the program will terminate and whose range consists of the set of states in which the program terminates. The function maps a state, S, onto the state in which

the program will terminate whenever it is started in state S. Mills and his colleagues at IBM have used this function, the *program function*, as commentary and specifications for many years. This function describes the precise effect of the program without describing the intermediate states. By studying the program function of component programs, rather than the program itself, we reduce the complexity of the task of understanding the system as a whole.

The functions that must be described are piece-wise continuous with many points of discontinuity. Conditional expressions that describe such functions precisely tend to be unacceptably complex. To make the mathematics manageable for typical engineers, alternative notations for describing the functions had to be developed.

We describe these functions by program-function tables. The column headings contain predicates that partition the domain of the function. The predicates characterise state classes. Every starting state for which the program's termination is guaranteed, must be in one and only one of those classes. The row headings are program variable names; there is a row for every variable that may be changed by a program. The entries in a given row and column are expressions defining the final value of the row's variable whenever the program is started in the column's mode. Whenever an entry in a table would be a complex conditional expression, the entry can be another table.

Figure 3 shows a simple program. Figure 2 shows the tabular expression of the relational specification of that program.

In our experience, the use of these tabular expressions is essential to the success of a review. The expressions that arise otherwise are often so complex, that human beings simply cannot read them carefully and correctly. The tables control the complexity of the expressions in three ways:

1) the table parses the expression for the reader, eliminating many nested pairs of parentheses and revealing the intended structure of the expression,
2) the table eliminates many repetitions of the subexpressions that appear in column headings,
3) because each table entry only applies to a small part of the function's domain, the expression in that entry can be simplified.

3 Quality Control

Quality control for safety-critical software will always have three components: testing of the product, inspection of the code, and evaluation of the design personnel and the methods that they use. Because of the complexity of software, and its large state space, complete testing is rarely practical. Because software can implement functions with discontinuities at any point in the domain, interpolation between test points is not valid. It follows that testing must be supplemented by a systematic inspection, which we can call verification. Although, computer scientists often discuss testing and verification as if they were alternative approaches to QA, we view testing and verification as complementary. Verification can reveal problems that might never be found in testing, but testing can reveal errors in the assumptions made during the verification. Testing and verification will both be in vain unless the personnel working on the project have the proper training and experience. We feel that each of these, testing, verification, and qualification of personnel are like three legs of a tripod - all three legs are essential and mutually supportive.

3.1 Testing

Both planned testing and statistically valid random testing are needed. Planned testing means a programme of testing designed to insure coverage of all “interesting” cases, such as boundary values. With planned testing, one attempts to identify the discontinuities in the functions and to make sure that each interval between those discontinuities is tested. By statistically valid random testing we mean the use of realistic test cases randomly chosen using distributions typical of what will be encountered in practice. Experience shows that thorough planned testing is the most effective way to uncover errors that manifest themselves in many circumstances. However, planned testing often misses subtle cases. In fact, there is good reason to believe that those who design the planned tests will overlook the same cases that the program designers overlooked. This is particularly true when programmers test their own programs. Further, planned testing does not help in assessing the reliability of a software system. As Mills often says, “Planned testing is a source of anecdotes, not data”. Random testing is free of the prejudices that cause human testers to overlook certain cases. Moreover, carefully designed random tests can provide estimations of the reliability that will be experienced in actual use.

For real-time systems each “test” case must be a trajectory. The length of each trajectory must be longer than the longest period that the software can retain data. Each trajectory must begin at a randomly chosen state and should include a situation in which action by the system is essential to safety. The trajectories, and the initial states of the system, must be chosen from a distribution resembling the distribution of trajectories and system states that would be encountered in practice. Reliability estimates obtained in this way, will only be good if the distribution of trajectories that is used accurately reflects actual usage [22].

3.2 Review

The initial attempts by engineers from AECB and Queen's to review the software were quite frustrating. There was an iterative process in which one formed an hypothesis about the intended function of a section of the program, then checked to see if it actually did what one thought it should. Often, the hypothesis was wrong. The informal documentation was not of much help; its natural language statements were often vague, and sometimes quite misleading. It was not easy to find the information that one wanted. Although informal inspections revealed some discrepancies between the code and the documentation we had no confidence that we had found all of the problems. A more formal, and systematic, process was needed. Without it, the complexity of the system would have left us with too much uncertainty.

For the Darlington Station, the AECB required a formal review based heavily on the formal documentation described above. There were four teams involved in the review process.

1) Atomic Energy of Canada Ltd. (AECL), the designers of the CANDU reactor system, prepared the formal requirements documentation.
2) Consultants hired by Ontario Hydro, most without previous knowledge of the nuclear field, examined the code and produced the program function tables.
3) Employees of Ontario Hydro were asked to compare the tables produced by the first teams and produce documented “proofs” that the functions implemented by the program were those required by the formal documentation. If discrepancies were discovered, the second team was given a chance to revise its tables but was not told the nature of the discrepancy. All unresolved

discrepancies were reported to the AECB and to an Ontario Hydro safety analysis team.

4) AECB employees and consultants audited the process reviewing a subset of the proofs. These proofs were presented, classroom style, to the auditors, who could interrupt and question the validity of any step in the deduction.

This team structure reflects our “separation of concerns” or “divide and conquer” approach. The first team was concerned only with the requirements posed by the environment. They had no need to examine the structure of the computer system. The second team examined only the code, never the requirements. The third team compared two sets of function tables; they could ignore the environment and the code, focusing only on the tables. The audit team was concerned about the overall process, but could look at it one step at a time.

The use of the program function tables and requirements tables took the review out of the domain of algorithm analysis and into a domain of mathematical expression manipulation. This change of domain resulted in the discovery of some subtle discrepancies that had escaped both the testing and the thorough conventional reviews that preceded the AECB review.

The effect of this approach was to reduce the extremely complex task of reviewing the system to a large number of relatively simple tasks. The tasks were often dull and tiresome, but the systematic procedure, made possible by the tabular organisation of our expressions, made it possible to take breaks and to rotate personnel to prevent burnout.

Safety experts, not software experts, reviewed each discrepancy and determined the action that should be taken. Most of the discrepancies were benign, they would not have been hazardous. For example, the software did some self-checking that was not required. This might have resulted in plant shutdowns that were not essential, but not in a failure to shutdown in an emergency. Other useful thoughts on effective software reviews can be found in [16].

3.3 Personnel and Design Process

The evaluation of the qualifications of the software professionals as well as the procedures that are followed is the most difficult area for a licencing board. Unlike other engineering fields there are no bodies that licence “software engineers”. More important, there are no widely accepted standards for their education. A course that one professor may consider essential, can be labeled “bogus” and “misleading” by other colleagues. Still further, their are no standards for the process and procedures that they should follow. There are dozens of competing “methodologies” being touted by commercial firms and academic researchers. While there are many efforts to identify required knowledge for software engineers there is little agreement on the requirements. In fact, even the issue of whether or not they should be engineers in the traditional sense, is controversial. Agreement is badly needed.

4 Conclusions

We view the formal analysis methods described in this paper as essential requirements for safety critical software. For Regulatory Authorities these techniques provide the assurance that the software meets its specifications and will not exhibit unintended, unsafe behaviour. The requirements proposed are similar to those of conventional professional engineering where formal analysis of a design is required before the design is committed for fabrication and before it is placed into service.

While the techniques applied to the Darlington Shutdown Software are feasible, and have resulted in an effective review, they were costly. With our present tools, we believe that this approach is practical only when reliability and trustworthiness are extremely important and the codes are relatively small (less than 10,000 lines of code). However, the cost could have been reduced if: a) a precise set of standards had been in place when the development of the software began and b) a set of tools appropriate to the methods had been available for use during development and review. The Darlington code was written before it was known that the techniques applied in this paper would be applied. Because this was a case of reverse engineering, the job was much more difficult than it would have been if the procedures had been set out from the start. In the reverse engineering situation, we were not able to apply the information hiding principle or to use those aspects of our approach that depend on it.

Previous to our work the longest leg of the tripod was testing. QA depended on extensive testing supplemented by relatively informal inspections. The use of the function tables has made the second leg, systematic inspection, much longer. Because of the tabular format, the study of these complex programs was systematically reduced to the study of a large number of relatively simple cases. Much of the reasoning could be done by formula manipulation rather than an intuitive understanding of algorithmic processes. However, the third leg of the tripod remains too short. There is a clear need for improved education for computer professionals [19] and for Professional Certification of Software Engineers.

5 Acknowledgements

We would like to acknowledge Glenn Archinoff, Rick Hohendorf, Dominik Chan, and the staff of AECL and Ontario Hydro for their feedback on problems encountered and lessons learned in deriving program functions for the Darlington Shutdown Software. Dr. N. Leveson served as a consultant to Ontario Hydro during the review and her influence was very positive. A. J. van Schouwen participated in the development of the ideas and the examples. Jim Kendall played an important role in recognising that the code required more systematic analysis than it had received. His assistance with earlier papers on this topic is greatly appreciated. Four anonymous referees made some insightful and helpful comments. Unfortunately, space did not permit us to expand our explanations of the figures. We hope that interested readers will turn to the references for additional information.

6 References

1. Archinoff, G.H., Hohendorf, R.J., Wassyng, A., Quigley, B., Borsch, M.R., Verification of the Shutdown System Software at the Darlington Nuclear Generating Station,International Conference on Control & Instrumentation in Nuclear Installations,The Institution of Nuclear Engineers, Glasgow, United Kingdom, May 1990, No. 4.3, 23 pp.
2. Britton, K.H., Parnas, D.L., A-7E Software Modules Guide,.NRL Memorandum Report 4702,United States Naval Research Laboratory, Washington D.C., December 1981, 35 pp.
3. Clements, P.C., Parnas, D.L., Weiss, D.M., Enhancing Resuability with Information Hiding, Proceedings of the Workshop on Reusability in Programming,September 1983, pp. 240-247. Also in: Freeman, P. (editor), .Software Reusability,IEEE Computer Society Press, 1987, pp. 83-90.

4. Gannon, J.D., Hamlet, R.G., Mills, H.D., Theory of Modules, IEEE Transactions on Software Engineering, Vol. SE-13, No. 7, July 1987, pp. 820-829.

5. Heninger, K.L., Kallander, J., Parnas, D.L., Shore, J.E., Software Requirements for the A-7E Aircraft,NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington D.C., November 1978, 523 pp.

6. Heninger, K.L., Specifying Software Requirements for Complex Systems: New Techniques and their Application, IEEE Transactions Software Engineering, Vol. SE-6, January 1980, pp. 2-13.

7. Hester, S.D., Parnas, D.L., Utter, D.F., Using Documentation as a Software Design Medium, Bell System Technical Journal, Vol. 60, No. 8, October 1981, pp. 1941-1977.

8. Mills, H.D., The New Math of Computer Programming, Communications of the ACM, Vol. 18, No. 1, January 1975, pp. 43-48.

9. Mills, H.D., Function Semantics for Sequential Programs, Proceedings of the IFIP Congress '80, North Holland 1980, pp. 241-250.

10. Mills, H.D., Basili, V.R., Gannon, J.D., Hamlet, R.G., Principles of Computer Programming: A Mathematical Approach, Allyn and Bacon, 1987.

11. Parnas, D.L., Information Distributions Aspects of Design Methodology, Proceedings of the IFIP Congress '71, Booklet TA-3, 1971, pp. 26-30.

12. Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.

13. Parnas, D.L., A Technique for Software Module Specification with Examples, Communications of the ACM, Vol. 15, No. 5, May 1972, pp. 330-336. Also in: Gehani, N., McGettrick, A.D. (editors), Software Specification Techniques, AT&T Bell Telephone Laboratories, 1985, pp. 75-88.

14. Parnas, D.L., A Generalized Control Structure and Its Formal Definition, Communications of the ACM, Vol. 26, No. 8, August 1983, pp. 572-581.

15. Parnas, D.L, Clements, P.C., Weiss, D.M., he Modular Structure of Complex Systems, . Proceedings of the Seventh International Conference on Software Engineering, March 1984, pp. 408-417. Also in: IEEE Transactions on Software Engineering, Vol. SE-11, March 1985, pp. 259-266.

16. Parnas, D.L., Weiss, D.M., Active Design Reviews: Principles and Practices, Proceedings of the 8th International Conference on Software Engineering, London, United Kingdom, August 1985. Also in: Journal of Systems and Software, Vol. 7, No. 12, December 1987, pp. 259-265.

17. Parnas, D.L., Wadge, W.W., Less Restrictive Constructs for Structured Programs, Technical Report 86-186, Queen's University, C&IS, October 1986, 16 pp.

18. Parnas, D.L., Wang, Y., The Trace Assertion Method of Module Interface Specification, Technical Report 89-261, Queen's University, C&IS, Telecommunications Research Institute of Ontario (TRIO), October 1989, 39 pp.

19. Parnas, D.L., Education for Computing Professionals, IEEE Computer, Vol. 23, No. 1, January 1990, pp. 17-22.

20. Parnas, D.L., Asmis, G.J.K., Kendall, J.D., Reviewable Development of Safety Critical Software, International Conference on Control & Instrumentation in Nuclear Installations, The Institution of Nuclear Engineers, Glasgow, United Kingdom, May 1990, No. 4.2, 17 pp.

21. Parnas, D.L., Madey, J., Functional Documentation for Computer Systems Engineering, Technical Report 90-287, Queen's University, C&IS, Telecommunications Research Institute of Ontario (TRIO), September 1990, 14 pp.

22. Parnas, D.L., van Schouwen, J., Kwan, P., Evaluation of Safety-Critical Software, Communications of the ACM, vol. 33, no. 6, June 1990, pp. 636-648.

23. van Schouwen, A.J., The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems, Technical Report 90-276, Queen's University, C&IS, Telecommunications Research Institute of Ontario (TRIO), May 1990, 93 pp.