Dependability of Software in Airborne Mission Systems

Kiril Uzunov and Thong Nguyen

Air Operations Division
Defence Science and Technology Organisation

DSTO-TR-2111

ABSTRACT

Dependability is an important quality attribute of software. As systems move towards complex system of systems and Ultra Large Scale systems, demand for methodologies for evaluating dependability will grow. The problem we aim to solve is how to evaluate the dependability of software early in the development cycle and before its deployment in order to assist Technical Risk Assessment, and to provide an additional assurance when monitoring the project health. This report surveys the existing approaches to software dependability.

RELEASE LIMITATION

Approved for public release
Dependability of Software in Airborne Mission Systems

Executive Summary

**Dependability** is a generic term which combines several software quality attributes: reliability, availability, maintainability, safety, and integrity. It has been a major topic of research for years, but its importance will grow when we move towards more complex, decentralised systems. The growth of complexity of software has exceeded the increase of complexity in physical components and is a major obstacle for achieving dependable systems. Most of the technologies in software engineering are human based and the variations in human skills and motivation add to the issues of complexity and difficulty. Software development faces other challenges, such as integration of legacy systems which were not designed for integration, interoperability with systems which are under development themselves, evolution in protocols, standards and underlying technologies, and an environment which is fluid both during and after development. These challenges put even more stress on the dependability attributes.

The objective is to evaluate software dependability early in the development cycle and before deployment (see Figure E-1). This evaluation will assist the Technical Risk Assessment (TRA) and provide an additional approach when reviewing the project health. Evaluation of software dependability would also provide input for making managerial decisions such as when to stop testing and release the software, planning for verification and maintenance etc.

![Figure E-1 Dependability evaluation framework](image)

The dependability of a system needs to be considered in all stages of the software development life-cycle in order to avoid expensive rework in later stages. A failure to
meet the quality requirements (which include dependability) usually manifests itself when the system is submitted for integration and system testing. At this stage it is much more expensive to fix problems whose root cause is in the requirements definition and architecture design. The report surveys the approaches to dependability evaluation applied at various phases of the software development life-cycle: eliciting dependability requirements during Requirements Definition; architecture-based reliability models, dependence models, scenario based assessment of reliability and Model Based Software Engineering during Architecture Design; static analysis during Coding; reliability growth models during Test/Verification. The evidence from these approaches can give us only limited confidence in the reliability of a system which needs to be augmented by indirect ways of evaluation – the maturity of the engineering process and the standards used for development. Combining such disparate evidence (which includes uncertainty) to make an assessment of dependability is difficult because: (1) attaching a weight to the various pieces of evidence is subjective and (2) the diversity of the evidence makes its consolidation hard.

Among the several formalisms which can be applied to reason about the disparate evidence for dependability, Bayesian probability has appeared as the best approach based on its maturity and availability of tools. The development of a Bayesian Networks model is proposed to assist in doing TRA, monitoring project health and providing information for management decisions. Ultimately, the human expert will be feeding data in such a model, investigating possible options and interpreting the results from the tool.

In conclusion, this report baselines the research in the available methods for assessment of dependability, establishes the limitations of the existing evidence, formulates the problem to be solved and proposes the development of a Bayesian Network (BN) model as a solution to help with the assessment. This baseline will be used to define the graph (i.e. the qualitative specification) of a BN model as well as its conditional tables (i.e. the quantitative specification).
Authors

Kiril Uzunov
Air Operations Division

Kiril Uzunov's research interests are in software dependability modelling, software metrics and reliability assessment, and software engineering. His background is in software development (Electronic Design Automation software and real-time - hard drive subsystems), software testing and verification, and project management. Before joining DSTO in 2006, he spent 11 years with Freescale Semiconductors Inc. and Motorola Inc. He has a Masters degree in Computer Systems Engineering.

____________________

Dr Thong Nguyen
Air Operations Division

Dr Nguyen holds BE (Honour I), BSc and PhD from Adelaide University and MBA (TechMgt) from Deakin University. He is currently the Research Program Manager for Airborne Mission Systems Branch, AOD, where he has been leading a team of 8 researchers and NICTA colleagues to develop DSTO's expertise on Capability Integration. The current AMD research program employs different techniques such as AADL, Colour Petri Nets, Bayesian Networks, software architecture evaluation methods to assess software qualities and integration at the architecture level and also includes three aspects of Information Integration Management, Fusion, and Exchange.

____________________
Contents

LIST OF ACRONYMS..............................................................................................................ix

1. INTRODUCTION............................................................................................................... 1

2. OVERVIEW OF DEPENDABILITY ATTRIBUTES AND DEFINITIONS .......... 3
   2.1 Definitions .................................................................................................................. 3
   2.2 Taxonomy .................................................................................................................. 4
   2.3 Software Reliability Modelling............................................................................. 4

3. PEOPLE RELATED ASPECTS OF DEPENDABILITY ................................................ 6

4. PROCESS RELATED ASPECTS OF DEPENDABILITY ............................................. 7
   4.1 The SEI Capability Maturity Models...................................................................... 7
   4.2 ISO 9000 Standards................................................................................................. 7
   4.3 Industry practices...................................................................................................... 8
   4.4 Relevant Standards................................................................................................... 9
      4.4.1 Standards Related to Reliability ............................................................... 10
      4.4.1.1 IEEE Std. 982.1 “IEEE Standard Dictionary of Measures of the Software Aspects of Dependability” .................................................................................. 10
      4.4.1.2 IEEE 982.2 “IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software” ................................................................. 11
      4.4.1.3 IEC 60300 series of Dependability Management ........................................ 11
      4.4.1.4 Failure Mode and Effects Analysis (FMEA) standards.............................. 11
      4.4.1.5 IEC 61025: Fault Tree Analysis ............................................................. 11
      4.4.1.6 IEC 61014-2003: Programmes for Reliability Growth ......................... 12
      4.4.2 Safety Standards ............................................................................................. 12
         4.4.2.1 MIL-STD-882 ................................................................................... 12
         4.4.2.2 RTCA DO-178 ................................................................................ 13
         4.4.2.3 RTCA DO-278 ................................................................................ 13
         4.4.2.4 Eurocontrol Draft Air Traffic Management (ATM) Standard ............. 14
         4.4.2.5 IEC/AS/NZS 61508 .......................................................................... 14
         4.4.2.6 Def Stan 00-56.................................................................................... 14
         4.4.2.7 Def(Aust) 5679................................................................................... 15
         4.4.2.8 ARP 4754.......................................................................................... 15
         4.4.2.9 EN 50128 ......................................................................................... 15
         4.4.2.10 The Joint Software System Safety Handbook (JSSSH) ....................... 16
      4.4.3 Security Standards .............................................................................................. 16
      4.4.3.1 ISO/IEC 15408 .................................................................................... 16
      4.4.4 Software Development Standards ....................................................................... 16
      4.4.4.1 MIL-STD-498 ................................................................................... 16
      4.4.4.2 Software Development Standard for Space Systems ......................... 16
      4.4.4.3 ISO/IEC 12207 .................................................................................. 16
   4.5 Discussion .................................................................................................................. 17
5. TECHNOLOGY AND PRODUCT RELATED ASPECTS OF DEPENDABILITY 19

5.1 Concept and Requirements Definition .............................................................. 19
  5.1.1 Elicitation of Dependability Requirements.................................................. 19
  5.1.2 Defects and Risks Prediction ........................................................................ 20

5.2 Architecture/High Level Design .......................................................................... 21
  5.2.1 Model Based Software Engineering............................................................ 21
  5.2.2 Architecture Dependence Analysis ............................................................. 22
  5.2.3 Using Scenarios for Assessing the Dependability Attributes ..................... 23
  5.2.4 Architectural Patterns for Non-Functional Requirements (NFR) ............... 23
  5.2.5 Architecture-based Software Reliability Models ......................................... 23
  5.2.6 Review Based Techniques .......................................................................... 26

5.3 Low Level Design and Coding ............................................................................. 27
  5.3.1 Static Analysis ............................................................................................. 28
  5.3.1.1 Programmer Errors Found ..................................................................... 28
  5.3.1.2 Studies on the Effectiveness of the Static Analysis Tools ....................... 28
  5.3.1.3 Limitations of the Static Analysis Tools ................................................ 31
  5.3.1.4 Adoption of Static Analysis Tools ......................................................... 31
  5.3.2 Mining Versions History ............................................................................. 32
  5.3.3 Notes .......................................................................................................... 32

5.4 Testing ....................................................................................................................... 33
  5.4.1 Analogy Models ........................................................................................... 33
  5.4.2 Using Test Metrics ....................................................................................... 33
  5.4.3 Software Reliability Growth Models .......................................................... 34
     5.4.3.1 Time Domain Approach ...................................................................... 34
        5.4.3.1.1 Classical Models ........................................................................... 34
        5.4.3.1.2 Bayesian Models ......................................................................... 35
        5.4.3.1.3 Markov Models ........................................................................... 36
        5.4.3.2 Error Seeding and Tagging Approach ............................................ 36
        5.4.3.3 Data Domain Approach .................................................................. 36
        5.4.3.4 Software Reliability Models – Comparison and Selection ............... 37
        5.4.3.5 Tools Implementing SRM ................................................................ 39
        5.4.3.6 Notes on the Reliability Growth Models ......................................... 39
  5.4.4 Evaluation of Dependability Based on Software Metrics ......................... 40
     5.4.4.1 Definition of the Measure .................................................................... 40
       5.4.4.1.1 Size Measures ............................................................................... 40
       5.4.4.1.2 Complexity Measures .................................................................. 40
       5.4.4.1.3 Object Oriented Design (OOD) Measures .................................. 42
       5.4.4.1.4 Function Point Analysis ................................................................. 42
       5.4.4.1.5 Multivariate Approaches ............................................................... 42
       5.4.4.1.6 Other Non-code Measures ............................................................. 43
     5.4.4.2 How the Measures are Collected and Used .......................................... 44
     5.4.4.3 Discussion on Metrics ......................................................................... 45

6. CONCLUSIONS ........................................................................................................... 47

7. FUTURE WORK .......................................................................................................... 50
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>AMS</td>
<td>Airborne Mission System</td>
</tr>
<tr>
<td>ATM</td>
<td>Air Traffic Management</td>
</tr>
<tr>
<td>ATSL</td>
<td>Approved Technology Standards List</td>
</tr>
<tr>
<td>BN</td>
<td>Bayesian Network</td>
</tr>
<tr>
<td>CIS</td>
<td>Communications and Information Systems</td>
</tr>
<tr>
<td>CMMI</td>
<td>Capability Maturity Model Integrated</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The Shelf</td>
</tr>
<tr>
<td>CTMC</td>
<td>Continuous Time Markov Chain models</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DECOS</td>
<td>Dependable Embedded Components and Systems</td>
</tr>
<tr>
<td>DII</td>
<td>Defence Information Infrastructure</td>
</tr>
<tr>
<td>DTMC</td>
<td>Discrete Time Markov Chain models</td>
</tr>
<tr>
<td>FAA</td>
<td>Federal Aviation Authority</td>
</tr>
<tr>
<td>FMEA</td>
<td>Failure Mode and Effect Analysis</td>
</tr>
<tr>
<td>FMECA</td>
<td>Failure Mode, Effect Analysis</td>
</tr>
<tr>
<td>GOTS</td>
<td>Government Off-The Shelf</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>INCOSE</td>
<td>International Council of Systems Engineering</td>
</tr>
<tr>
<td>ISSRE</td>
<td>International Symposium on Software Reliability Engineering</td>
</tr>
<tr>
<td>MIS</td>
<td>Management Information Systems</td>
</tr>
<tr>
<td>MTBF</td>
<td>Mean Time Between Failures</td>
</tr>
<tr>
<td>MoD</td>
<td>Ministry of Defence (UK)</td>
</tr>
<tr>
<td>MTTR</td>
<td>Mean Time To Repair</td>
</tr>
<tr>
<td>NFR</td>
<td>Non-Functional Requirements</td>
</tr>
<tr>
<td>NRO</td>
<td>National Reconnaissance Center</td>
</tr>
<tr>
<td>OOD</td>
<td>Object Oriented Design</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal Component Analysis</td>
</tr>
<tr>
<td>SAN</td>
<td>Stochastic Activity Networks</td>
</tr>
<tr>
<td>SCAMPI</td>
<td>Standard CMMI Appraisal Method for Process Improvement</td>
</tr>
<tr>
<td>SDLC</td>
<td>Software Development Life Cycle</td>
</tr>
<tr>
<td>SDSSS</td>
<td>Software Development Standard for Space Systems</td>
</tr>
<tr>
<td>SESS</td>
<td>Software Engineering for Secure Systems (conference)</td>
</tr>
<tr>
<td>SEW</td>
<td>Software Engineering Workshop</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety integrity level(s)</td>
</tr>
<tr>
<td>SMC</td>
<td>Space and Missile Systems Center</td>
</tr>
<tr>
<td>SPN</td>
<td>Stochastic Petri Nets</td>
</tr>
<tr>
<td>SRGM</td>
<td>Software Reliability Growth Model</td>
</tr>
<tr>
<td>SRM</td>
<td>Software Reliability Model</td>
</tr>
<tr>
<td>SVRC</td>
<td>Software Verification Research Centre (University of Queensland)</td>
</tr>
<tr>
<td>TER</td>
<td>Test Effectiveness Ratio</td>
</tr>
<tr>
<td>TOR</td>
<td>Technical Operating Report</td>
</tr>
<tr>
<td>TRA</td>
<td>Technical Risk Assessment</td>
</tr>
<tr>
<td>ULS</td>
<td>Ultra-Large Scale</td>
</tr>
<tr>
<td>UMD</td>
<td>Unified Model of Dependability</td>
</tr>
<tr>
<td>WBS</td>
<td>Work Breakdown Structure</td>
</tr>
</tbody>
</table>
1. Introduction

The need for reliable systems, which can meet the high requirements of security and safety critical applications, is growing. Such systems are used in avionics, space, railway and transport, process control and medical systems. For example, in cars, embedded systems are tasked with critical functions like the anti-lock braking system (ABS) and airbags. Reliability is also coming under the spotlight in the field of consumer electronics due to the big volume of production and the high cost of rectifying defects. The dependability of a system is one of its major quality attributes and it needs to be considered in all stages of the development life-cycle: concept and requirements definition, design, development, verification and validation, and certification.

Although dependability and the means to attain it have been a focus of research for a long time, dependability evaluation still remains a challenge for software development. The demand for tools, methodology and frameworks for evaluating dependability will grow as systems move toward complex systems of systems and Ultra Large Scale (ULS) systems [Northrop et al. 2006], where software and hardware failures will be the norm rather than the exception. The acquisition of ULS systems will be simultaneous with their operation and this will require new methods of control, where the dependability analysis capability can play a pivotal role in achieving the desired software quality:

- The software has to be sufficiently reliable. This holds especially true for high-assurance systems.
- The software has to be maintainable, both in the sense of providing enough diagnostic information at start-up and runtime as well as providing a path for adding new functionality, upgrading Commercial/Government Off-The Shelf (COTS/GOTS) software or porting to different hardware platforms.
- No conclusion can be drawn for a specific system based solely on the fact that a similar system has achieved a certain level of reliability, or that the developer has successfully produced reliable systems before, or even that a certain software development process was successful in the past. It is necessary to measure how reliable the concrete system is and this has to be done before the system is deployed in operational use [Littlewood 2005].

Software is unreliable because of its:

- Novelty – software is used to implement functionality that has never been implemented before using a different technology. Typically, the software product is a single production item in comparison with most other manufactured products.
- Non-repeatatability – unlike manufacturing, the software products differ from each other.
- Difficulty – the problems to be solved by software require considerable intellectual effort.
- Complexity – Complexity is a major obstacle for achieving dependable software. The growth of the complexity of software has outstripped the increase in complexity in the physical components of the system [MOD_TECHN 2006]. Modern systems have most of their functionality enabled by software. The F-4 fighter in the 1960s had less than 10 percent of its functionality based on software; in the F/A-22 it is more than 80 percent [Wait 2006]. The effect of the complexity is
compounded by the size of the effort, which translates to long term projects involving many people. The planning and execution of such projects as a result is associated with risks which can be diminished only by applying a disciplined process and the ability to enhance transparency into the software development. A capability to define useful metrics and to make conclusions based on them is of the utmost importance. [DoD 2003] underlines the need for integrating the measurement and risk management and recommends the implementation of independent analysis capability: “All projects should include an assessment of measurement data by a group that is independent of the organization producing the data. This ensures that the measurement process is objective and provides an accurate, unbiased assessment of the project. In an outsourcing or acquisition scenario, both the supplier and the acquirer should perform their own analysis of the project’s measurement data. Objective communication can only occur when both parties understand the data. Because each party has its own perspective on project issues, independent analysis allows each organization to assess the impact of problems identified by measured data. Ideally, both supplier and acquirer should perform analysis continually.”

- Most of the technologies in software engineering are human based, and, as a result, the variations in human abilities add to the issues of complexity and difficulty.

As pointed out in [Boehm et al. 2001], all other things being equal, it costs 50 percent more per line of code (LOC) to develop high-dependability software products than to develop low-dependability products. Based on a typical life-cycle cost distribution (30 percent development and 70 percent maintenance), the cost per LOC for both types of software becomes approximately the same. Therefore, for Airborne Mission Systems, where the operation and maintenance costs are significant, the investment in dependable software is not only a requirement stemming from safety and reliability concerns, but it is also economically practical.

This research in software dependability is part of the AOD-AMS architecture evaluation program, which has the objective to develop a framework for qualitative and quantitative evaluation of avionics architectures, which may or may not include generic COTS components. The problem we aim to solve is the evaluation of dependability of the software before its deployment. This evaluation will assist the Technical Risk Assessment (TRA) and provide an additional approach when checking the project health. Such evaluation would also provide input for making managerial decisions such as when to stop testing and release the software, and in planning for verification and maintenance.
2. Overview of Dependability Attributes and Definitions

2.1 Definitions

The original definition of dependability underlined the justification of trust and stated that dependability is that property of a computer system such that reliance can justifiably be placed on the services it delivers [Laprie et al. 1994]. [Rushby 1994] examined the properties of the critical systems as they are viewed from the tradition of dependability (ultra-reliable and fault-tolerant systems), safety, security and real-time systems. He identified differences in the overall philosophy, the mechanisms and means of specification and assurance. For example, the dependability approach defines system failures in terms of system services; safety, however, is concerned with mishaps, i.e. external consequences which may be a result of correctly functioning, but incorrectly specified system. According to [Avizienis et al. 2004] dependability is the ability of a system to avoid service failures which are more frequent and more severe than expected.

Dependability is a generic term which includes several areas: reliability, availability, maintainability, safety, and integrity [Avizienis et al. 2004]. The attributes of dependability are:

- **Availability** – readiness for correct service;
- **Reliability** – continuity of correct service;
- **Safety** – absence of catastrophic consequences on the users or environment;
- **Integrity** – absence of unauthorised system alterations; and
- **Maintainability** – ability to undergo modifications and repairs.

**Software Reliability** is the probability that software will not cause the failure of the system for a specified period of time in a specified environment. The probability is a function of the inputs to, and use of, the system as well as a function of the existence of faults in the software. Reliability represents a user-oriented view of software quality [Musa et al. 1990] and is probably the most important characteristic of “software quality”.

**Reliability growth** [IEEE-729] is the improvement in software reliability that results from correcting faults in software.

As the **security** is a composite of the attributes of availability, integrity and confidentiality, the criteria for the evaluation of dependability of safety-critical systems have sometimes been referred to as **RAMSS** (Reliability, Availability, Maintainability, Safety and Security). The attributes of dependability define the capability of a system to provide the expected services. Often they are referred to as quality attributes i.e. properties of the system by which its quality will be judged. Quality-attribute requirements such as those for reliability, security, modifiability, and usability have a significant influence on the architecture of a system.
2.2 Taxonomy

The deviation of a system from its correct service is called an error. The cause of an error is called a fault.

The means to attain dependability are grouped as follows [Avizienis et al. 2004]:

- **Fault Prevention** aims to avoid fault occurrences by construction. It is the object of various development methodologies (e.g. information hiding, theorem proving, strongly-typed programming languages) and process improvement efforts.
- **Fault Tolerance** is aimed at fault avoidance. It is based on Error Detection and System Recovery (rollback, roll forward, compensation, diagnosis, isolation, reconfiguration and re-initialisation). The choice of error detection, error and fault handling depends on the fault assumptions considered during development.
- **Fault Removal** during development consists of verification, diagnosis and correction. Typical verification methods used during the different life-cycles phases of the SDLC are:
  - FTA, FMECA and Hazop (during system analysis and evaluation);
  - Theorem proving and model checking (formal methods);
  - Simulation and modelling;
  - Audit, inspection (review);
  - Functional, white box and black box testing, coverage, static analysis (testing); and
  - Fault injection.

During maintenance Fault Removal is executed through corrective or preventive maintenance. Fault removal depends on time, operational profile, and the quality of the repair activity.

- **Fault Forecasting** can be qualitative (ordinal) which aims to identify the causes of system failures or quantitative (probabilistic). It relates to **Fault introduction** which depends mostly on the characteristics of the developed code and the development process. Quantitative forecasting is based mainly on modelling and testing.

This classification is graphically represented in Figure 1.

2.3 Software Reliability Modelling

The factors affecting the reliability of software are fault introduction, fault removal, fault forecasting, fault tolerance and the environment. Fault introduction and removal depend on the characteristics of the code/technologies/tools, the process and the people. The environment depends on the operational profile. All these factors have to be considered when modelling the software reliability.

A **Software Reliability Model** (SRM) is a mathematical expression that specifies a general form of the software failure process as a function of factors such as fault introduction, fault removal and the operational environment. Because software reliability is a function of fault introduction, fault removal and the environment, the discovery of a new fault is a random event. One approach is to use past failure data (either from current testing activities or from previous projects) in order to predict future behaviour. A quantitative assessment of software reliability will provide guidance as to when to consider testing as
complete; makes possible a decision for release to be taken; helps to estimate the resources needed for the verification, the expected level of support after release and to plan the maintenance phase. The SRM can be used for:

- **Estimation (analysis)** – in this case statistical inference procedures are applied to observed failure data in order to make conclusion about the number of remaining faults (or verification time needed to reduce the number of remaining faults below specified threshold). The data can be used to compute the MTTF and MTTR, from which an estimate of the system availability can be obtained. The approach is also known as measurement-based approach.
- **Prediction** – conclusions are made based on the properties of the software product and the development process; conclusions can be also made based on historical data. This step can be undertaken before any execution of the program and is usually model based.

**Figure 1  Dependability taxonomy**

Architecture level reliability models can be applied early in the SDLC to do predictive model based analysis. Such analytical models have to make simplifying assumptions and as a result their prediction may not adequately reflect the characteristics of the system. The alternative approach - measurement-based analysis - is much more precise. It can be applied, however, only when the code is written or at least there is a working prototype. A combination of these two methods is used to overcome their problems [Gokhale et al. 2005].
3. People Related Aspects of Dependability

The dependability is a quality characteristic of the software and it can be evaluated by various means. We can group the various aspects of the evaluation based on the association with the ubiquitous “triangle” – People, Process and Technology.

The people factor aspect of the dependability evaluation will be kept out of this review and we will concentrate on the process, the technology and ultimately on the product issues. We will only briefly mention here that the people related aspect is very important. Most of the technologies and the theories of software engineering are human based and as such depend on the variations of skill level, competence and motivation. The provenance of open source software is another example of people related aspects to be considered during evaluation. In addition to the developer/manufacturer issues, there are human factors and concerns on the user/operator side, where the operator can be viewed as an integral component of the system or as an entity outside of the system. A lot of studies are dedicated to integrated Man-Machine models, e.g. [Ludtke et al. 2006]. Research initiatives like DIRC (The Interdisciplinary Research Collaboration in Dependability) undertaken by five British Universities is an example of an approach encompassing human factors in addition to the attributes like reliability, availability and safety. There is a need for methods, data and tools to resolve the human performance issues and the design of the systems needs to include capability to recover from human errors.
4. Process Related Aspects of Dependability

The quality of the software engineering process applied during the Software Development Life Cycle reflects on the product. This connection was pointed out early in [Conway 1968]: “The basic thesis of this article is that organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations. We have seen that this fact has important implications for the management of system design.”

Process assessment is looking into the methods used to produce the software [Beus-Dukic et al. 2003]. The process approach to quality evaluation is based on various standards and industry practices. This section contains a brief description of the major industry models and standards which promote the process approach. The purpose is to provide input into a dependability evaluation framework which characterizes the project environment.

4.1 The SEI Capability Maturity Models

The SEI Capability Maturity Models (CMM) promote the quality of the process [SEI CMMI]. The SEI CMM development was in response to a request to provide the US federal government with a method for assessing the capability of its software contractors. A capability maturity model defines the characteristics of a mature, capable process. It identifies the practices that are basic to implementing effective processes. The practices are grouped in Process Areas (PA) and can be used for benchmarking and for establishing process improvement plans. It should be noted that a CMMI model is not a process; it defines what needs to be done, but not how or who should do it.

SEI provides a methodology - Standard CMMI Appraisal Method for Process Improvement (SCAMPI) - and training for assessing the maturity level of an organisation. According to the CMMI terminology, appraisal is an examination of one or more processes by a trained team of professionals using an appraisal reference model, while assessment is an appraisal that an organization does to and for itself for the purposes of process improvement. Based on evidence of how the organisation implements the aspects of the PA, it is assigned a maturity level (in case of levelled presentation). The lifespan of an appraisal is three years - similar to that established for ISO 9000. The current version of CMMI is v1.2.

The Defence Materiel Organisation (DMO) is using the CMMI in its acquisition and maintenance of software intensive systems. However, since safety is only covered in the informative sections of the CMMI, the DMO and the Software Verification Research Centre (SVRC) developed a safety extension to CMMI "SAFE - A Safety Extension to CMMI" in the style of CMMI to address adequately the needs relating to safety-critical systems [Robinson et al. 2000].

4.2 ISO 9000 Standards

ISO 9000 is a family of standards for quality management. The standards originated in manufacturing, but they are applied now to different types of products, including
software. ISO 9001:2000 provides a set of requirements which are a necessary condition for an organization to fulfil in order to meet the customer expectation. The standard advocates the quality of the process by including the documenting, monitoring and reviewing the manufacturing processes to ensure a quality product is released. The AS/NZS ISO/ISO/IEC 90003:2007 Std. “Software Engineering - Guidelines for the application of AS/NZS ISO 9001:2000 to computer software” (which replaced the old ISO 9000-3:1997), is the quality management standard for software and related services. It provides guidance in applying ISO 9001:2000 to software. An independent audit may grant a certification of compliance with the ISO 9001 to an organization. The life span of this certification is three years. Similar to the SEI CMMI model, the certification concerns the process being applied and does not guarantee the quality of the product. It is focused on the methods and techniques that a process has to use in order to produce a quality product, but does not address development technology capabilities. There is an overlap to a degree with the CMMI model, but the major difference is that the CMMI is focused on continuous process improvement and the higher quality is a by-product of this.

Various industry sectors have developed their own version of ISO 9000 in order to match closer the specifics of the industry. The ones that may be relevant for us are:

- AS 9100 - The Aerospace Basic Quality System Standard developed by major aerospace manufacturers.
- ISO/TS 16949 is developed for the automotive industry. It includes the full text of ISO 9001:2000 plus specific requirements for the automotive industry.
- TickIT is a guide and framework developed by the British Standards Institute to help software development organizations to be audited and certified against ISO 9001 [TickIT 2001]. The certification is recognised in UK. [TickIT 2002] is a guide aiming to introduce measurements of the process, the product and the quality objectives as part of meeting the ISO 9001:2000 requirements.

4.3 Industry practices

Trillium was a model developed by Bell Canada to assess the product development and the support capability of suppliers of telecommunications or information technology-based products [Trillium 1994]. The objectives of an assessment were to assess risks associated with the procurement of a given product, and/or monitor a quality/capability improvement program. The model is not a prescribed process, but it provides best practices which contribute to the organisation’s capability to develop software with a required quality and minimum defects in the shortest time. Trillium was based on the CMM v1.1, ISO 9001 and included also its own specific practices. In comparison with the other models, Trillium has a strong product perspective.

Of interest for this review are the Trillium specific practices which have impact on the various dependability attributes (mainly reliability) in two main areas:

- Measurement related:
  - The number of additions, changes and deletions to requirements, design, code and test specifications are measured.
  - Product failures and statistics are systematically measured throughout the life-cycle of the product.
The number of detected and corrected defects is systematically measured for each life-cycle phase.

Source code metrics (e.g., code complexity) are taken and compared to a predefined set of permissible values according to a documented procedure.

- Development related:
  - Programming language selection - state of the practice high-level languages are used appropriately for all software products.
  - Requirements capture - Prototyping techniques are used to verify the correctness of critical specification items (e.g., user interface specification) with the customer.
  - Prototyping techniques are used to explore design options and validate design decisions prior to implementation.
  - Code skeletons are automatically generated from designs by CASE tools. The majority of source code (>80%) is automatically generated by CASE tools.
  - Formal methods (e.g., VDM, Z, LOTOS) are used for the development of critical software components.
  - Design complexity is measured and compared to the organizational baseline according to a documented procedure.
  - The generation of test cases is automated from requirements and designs.
  - FMECA and Fault Tree Analysis are performed on all critical hardware and software modules as part of the design activities and are part of the design reviews.
  - Reliability modelling influences the selection of design alternatives for non-software sub-systems, and for system level for software.
  - Reliability growth models are used for resources and/or schedule estimation using past project parameters (size, initial failure intensity and intensity decay parameters). These models are continually updated.
  - Field reliability estimation is based on the determination of the testing acceleration factor and the reliability growth model.
  - Reliability modelling influences software design alternative selection from both static code properties and dynamic code behaviour data (in-line with re-use philosophy) i.e., Reliability Block Diagram, Markov modelling, Fault Tree Analysis, Strength-Stress Analysis, Stochastic Petri-Net Modelling.

4.4 Relevant Standards

We will describe briefly in this section the major standards, guidelines and handbooks which relate to different attributes of dependability. The purpose of this brief is to augment the process frameworks described before. The fact that certain standards have been applied may be linked to a corresponding level of the relevant attribute of dependability.
4.4.1 Standards Related to Reliability

4.4.1.1 IEEE Std. 982.1 “IEEE Standard Dictionary of Measures of the Software Aspects of Dependability”

This standard provides measures for continual self-assessment and improvement of the software aspects of dependability [IEEE 982.1]. It defines criteria for inclusion of new measures and, based on these criteria, introduces new measures, modifies or deletes existing measures from the previous version of the standard. It categorizes the measure into three classes – reliability, maintainability and availability. The standard takes quite a pragmatic approach to the measures selection – a measure must satisfy at least one of the following criteria:

- Have a minimum number of recognized uses;
- Demonstrated or potential utility in producing reliable, maintainable and available software;
- Should not be overly complex to understand and implement; and
- Should be independent of the development paradigm.

Measures, such as software documentation, system performance, reliability etc., from the previous revision of the standard, have been removed. As a result, the number of measures was reduced from 39 to 16. The recommended measures are:

- Reliability measures: newly introduced measures in this revision (time to next failure, risk factor regression model, remaining failures, total time to achieve specified remaining failure, network reliability); modified from the previous revision (defect density, test coverage index, requirements compliance, failure rate); retained measures from the previous revision (fault density, requirements traceability, mean time to failure - MTTF);
- Maintainability measures: mean time to repair (MTTR), network maintainability; and
- Availability measures: availability, network availability.

For each of these measures, the standard provides definition, elaborates the parameters and the data requirements, and describes briefly the experience with applying the measure and the relevant tools if available.

The IEEE 982.1 is a tool that can be used for evaluation and prediction of the software reliability [Schneidewind 2007]. Various reliability quantities, such as remaining failures, time to failure etc., can be calculated. A project can successfully predict reliability by using a reliability model and the 982.1 measures. Schneidewind gives an example of applying the IEEE 982.1 to the Space Shuttle’s avionics software (430 KLOC). Using reliability measurement and predictions like the remaining failures, cumulative failures and time to next failure, a decision can be made when to stop testing. For example, the remaining failures and the time to next failure allow an assessment of the probability of a failure occurring during a mission. This quantifies the risk of deploying the software.
4.4.1.2 IEEE 982.2 “IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software”

[IEEE 982.2] is the guide for the use of IEEE 982.1. The end goal is to improve software reliability by supporting development processes which promote fault avoidance, early fault detection and prompt removal, and system-designed fault tolerance. A process strategy to achieve reliable software includes activities, methods, tools, and measurements. The standard relates the applicability of various measures taken at different stages of the development to the objectives of the stage and the different stakeholders - developers, project managers and verification engineers.

4.4.1.3 IEC 60300 series of Dependability Management

The IEC 60300 series of dependability management Standards provide general guidelines for establishing a dependability management system to meet most organizational or project needs.

4.4.1.4 Failure Mode and Effects Analysis (FMEA) standards

The IEC 60812 “Analysis techniques for system reliability —Procedure for failure mode and effects analysis” describes Failure Mode and Effects Analysis (FMEA) and Failure Mode, Effects and Criticality Analysis (FMECA), and gives guidance as to how they may be applied to achieve various objectives by providing the procedural steps necessary to perform an analysis. It identifies the terms, assumptions, criticality measures, failure modes and provides examples of the necessary worksheets and forms.

Other FMEA standards are MIL-STD-1629A FMECA standard (cancelled Aug 1998), the SAE ARP5580 FMEA standard, the SAE J1739 FMEAs, the Automotive Industry Action Group (AIAG) FMEA, Daimler Chrysler, Ford, and GM FMEA methodologies, and BS 5760.

4.4.1.5 IEC 61025: Fault Tree Analysis

This standard (last edition 2006) describes FTA and provides:

- Definition of basic principles (including describing and explaining the associated mathematical modelling, and the relationships of FTA to other reliability modelling techniques); and
- Description of the steps involved in performing the FTA.

This standard addresses two approaches to FTA:

- The qualitative (traditional) approach does not address the probability of events and their contributing factors. It is based on a detailed analysis of events/faults and is used largely in nuclear industry applications and many other instances where the potential causes or faults are sought out, without regard to their likelihood of occurrence. At times, some events in the traditional FTA are investigated quantitatively, but these calculations are disassociated with any overall reliability concepts and in these cases no attempt is made to calculate overall reliability using FTA.
- The quantitative FTA models an entire product, process or system and the final result is the probability of occurrence of a top event representing reliability or probability of fault or a failure.
4.4.1.6 IEC 61014-2003: Programmes for Reliability Growth
This international standard specifies requirements and gives guidelines for the exposure and removal of weaknesses in hardware and software items for the purpose of reliability growth. Reliability growth is described based on analyses such as Failure Mode and Effects Analysis (FMEA), test of modules and sub-assemblies as well as test of prototypes and delivered items.
The standard applies when the product specification calls for a reliability growth programme of equipment (electronic, electromechanical and mechanical hardware as well as software) or when it is known that the design is unlikely to meet the requirements without improvement. The standard outlines the principles of mathematical modelling for estimating the growth achieved and the projected reliability. In the latest edition, the emphasis on mathematical modelling is reduced. The statistical methods can be found in IEC 61164: Reliability growth – Statistical test and estimation methods.

4.4.2 Safety Standards
Standards for software for safety-critical systems relate recommended practices to the level of required reliability, that is, they prescribe a level of software engineering rigour relevant to the risk. Standards are defining requirements both to the production process as well as to the product. Meeting the requirements of some standards can be used to claim that the desired reliability has been achieved.

Different standards have different objectives, e.g. DO-178 concentrates on software development, in particular the software development process, whereas EN 50128 puts emphasis on formal analysis. This review cannot be used as a substitution for studying the relevant standard - it aims to provide a brief list of the major applicable standards which may contribute to our development of a dependability framework.

For more detailed comparison of the various international safety standards the reader is referred to [Wabenhorst et al. 1999]. Eleven standards for computer-based safety-critical systems are surveyed:
- The Australian Def(Aust) 5679;
- MIL-STD-882C;
- NATO STANAG 4404 and STANAG 4452;
- UK Def Stan 00-56;
- Def Stan 00-55 and Def Stan 00-54;
- Avionics standards ARP4754, ARP4761 and RTCA/DO-178B; and
- Civilian standard IEC 61508.
The review is based on a wide range of attributes - levels of prescription and tailoring; safety management issues such as agents, their responsibilities, and deliverables required; and technical issues such as development constraints, hazard analysis, risk assessment, implementation assurance, human factors and non-development items. The intention of this review was to draw comparisons to Def(Aust) 5679.

4.4.2.1 MIL-STD-882
MIL-STD-882 [MIL-STD-882] outlines a wide range of safety engineering and safety management issues. 882C (released 1993) described multiple tasks which can be used to
meet the general requirements of the standard, while the current version 882D omits this. As a result some organisations still use 882C. The standard does not address adequately the range of engineering issues; in particular it provides only a limited version of the concept of a Safety Integrity Level (SIL) to be followed in software development. MIL-STD-882C contains no specific software implementation assurance requirements. As a result it is commonly paired with another standard such as DO-178B [Savive 2007].

4.4.2.2 RTCA DO-178

DO-178 “Software Considerations in Airborne Systems and Equipment Certification” [DO-178] is one of the oldest software safety standards which has influenced other software safety standards. It is intended to provide “guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence that complies with airworthiness requirements”. It should be noted that what is certified is not the software, but the system. The Federal Aviation Authority's (FAA) Advisory Circular AC20-115B established DO-178B as the accepted means to secure FAA approval of the digital computer software. The standard establishes five software levels which are mapped to system failure conditions as shown in Table 1. A developer must map its processes to the ones prescribed by DO-178B. COTS products are allowed only if they adhere to the same standard. In a manner very similar to the SEI CMMI process, the standard establishes requirements for the software planning and development process. As a result, certification to DO-178B requires delivery of multiple supporting documents and records. The quantity of items needed for DO-178B certification, and the amount of information that they must contain, is determined by the level of certification being sought. As DO-178B has issues like ambiguity in some areas and conflicting requirements, [DO-248] provides further clarification and resolution to inconsistencies contained in DO-178B. Due to the lack of general safety management and safety engineering material, this standard is most commonly used in combination with another safety standard such as MIL-STD-882.

<table>
<thead>
<tr>
<th>Level</th>
<th>Failure classification</th>
<th>Failure description</th>
<th>Probability description</th>
<th>FAR/JAR definition per flight hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Catastrophic</td>
<td>Aircraft loss and/or fatalities</td>
<td>Extremely improbable</td>
<td>.. &lt; 10-9</td>
</tr>
<tr>
<td>B</td>
<td>Hazardous</td>
<td>Flight crew cannot perform their tasks</td>
<td>Extremely remote</td>
<td>10-9 &lt; ... &lt; 10-7</td>
</tr>
<tr>
<td>C</td>
<td>Major</td>
<td>Flight crew efficiency impaired. Occupants discomfort including injuries</td>
<td>Remote</td>
<td>10-7 &lt; ... &lt; 10-5</td>
</tr>
<tr>
<td>D</td>
<td>Minor</td>
<td>Some inconvenience to occupants. Workload within crew capabilities</td>
<td>Probable</td>
<td>10-5 &lt; ...</td>
</tr>
<tr>
<td>E</td>
<td>No safety effect</td>
<td>No effect</td>
<td>Frequent</td>
<td>...</td>
</tr>
</tbody>
</table>

4.4.2.3 RTCA DO-278
[DO-278] covers the software for flight management which is situated on the ground. It provides six assurance levels as per Table 2.
In contrast to DO-178B, DO-278 defines a process for planning, acquisition, verification and quality assurance of COTS. It is required to demonstrate that unused COTS functionality does not affect the rest of the software. COTS service experience may be used instead of applying a DO-278 compliant development process, but as can be seen from the table, the requirements are high. For example, “one year” service experience means that no failure may have occurred for a continuous period of 8760 hours of representative use. DO-278 will be brought together with DO-178B into DO-178C.

<table>
<thead>
<tr>
<th>DO-178 Level</th>
<th>DO-278 assurance level</th>
<th>COTS service experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>AL 1</td>
<td>Not allowed</td>
</tr>
<tr>
<td>B</td>
<td>AL 2</td>
<td>Negotiate with approval authority</td>
</tr>
<tr>
<td>C</td>
<td>AL 3</td>
<td>One year</td>
</tr>
<tr>
<td></td>
<td>AL 4</td>
<td>Six months</td>
</tr>
<tr>
<td>D</td>
<td>AL 5</td>
<td>Not needed (typically)</td>
</tr>
<tr>
<td>E</td>
<td>AL 6</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

4.4.2.4 *Eurocontrol Draft Air Traffic Management (ATM) Standard*
This standard is expected to combine DO-178B, IEC 61508 and the SEI CMMI into one combined safety and quality assurance document [Kesseler 2003]. It covers operational use and maintenance phases in extension to DO-178B.

4.4.2.5 *IEC/AS/NZS 61508*
The civil standard IEC/AS/NZS 61508 "Functional safety of electrical/electronic/programmable electronic safety-related systems" provides detail on a wide range of safety management, safety engineering and engineering topics, aimed at the process industry but also widely applied in other domains. It is a generic standard for the functional safety of electrical/electronic/programmable electronic safety-related systems. Part 3 defines requirements for software assurance. Verification activities are planned during development, and consider whether or not:

- The software architecture fulfils the software safety requirements;
- The software system design satisfies the software architecture;
- The module design fulfils the software system design; and
- The code conforms to module design.

Designs and requirements are checked for feasibility, testability, readability and safe modification, and are verified with respect to appropriate test specifications. Data are also verified. A number of generic verification techniques are nominated with recommendations for use dependent on the safety integrity level. Formal proof is highly recommended for the highest level of integrity. Testing techniques are recommended depending on the safety integrity level.

4.4.2.6 *Def Stan 00-56*
The UK Ministry of Defence (MoD) developed Def Stan 00-55 "Requirements for Safety Related Software in Defence Equipment" which covered the development of software in safety-critical systems, filling a similar role to Part 3 of 61508. Def Stan 00-55 was generally used within a wider safety standard such as [DEF STAN 00-56]. Other closely related standards from the UK MoD were Def Stan 00-58 “HAZOP Studies on Systems Containing..."
Programmable Electronics” and Def Stan 00-54 “Requirements for Safety Related Electronic Hardware in Defence Equipment”; both of them were cancelled in 2004. All mentioned MoD standards have been superseded by the interim release of Issue 3 of 00-56. 00-56 Issue 4 was published on 01 June 2007.

4.4.2.7 Def(Aust) 5679
Def(Aust) 5679:1998 "The Procurement Of Computer-based Safety Critical Systems" encapsulates some of the policies of the Defence Materiel Organisation (DMO) with regards to the procurement, analysis, development, and operation of systems that may be safety-critical. The standard requires that an implementation assurance is conducted when a software component has been designed and implemented. It should be noted, that Def(Aust) 5679 is not widely used in the Airborne Mission Systems (AMS) mainly because of the prohibitive cost to prove adherence to the standard. Instead, standards from the system’s country of origin apply.

4.4.2.8 ARP 4754
The Society of Automotive Engineers (SAE) provides two standards representing Aerospace Recommended Practice (ARP) to guide the development of complex aircraft systems. SAE ARP 4754 "Certification Considerations for Highly-Integrated or Complex Aircraft Systems" deals with the system development processes of aviation systems and how to show compliance to a regulator. The complete development process is covered. The standard is designed for use with ARP4761, which contains guidance and examples of safety assessment procedures. Software assurance, however, lies outside the scope of the ARP standards. Coverage of complex hardware aspects of design are dealt with in RTCA document DO-254 "Design Assurance Guidance for Airborne Electronic Hardware".

4.4.2.9 EN 50128
The principles of software design assurance/software integrity can be observed also in the CENELEC European Norm (EN) 50128 Standard, “Railway Applications: Software for Railway Control and Protection Systems” which is a part of a group of related Standards like the EN 50126, “Railway Applications - Dependability for Guided Transport Systems”. It was prepared by Technical Committee “CENELEC TC 9X, Electrical and Electronic applications in railways”. The EN 50128 Standard concentrates on the methods which need to be used in order to provide software meeting the demands for safety. The key concept of this European Norm is that of SIL: the more dangerous the consequences of a software failure, the higher the software safety integrity level will be. Following the guidelines of the related standards, a System Safety Requirements Specification, which identifies all safety functions allocated to software and determines the system safety integrity level, needs to be created. This document is an input to the Software Requirement Specification and the Software Architecture. The EN 50128 Standard recommends the use of formal methods in the Software Requirements Specification as stated in Table A2, Software Requirements Specification (clause 8): “The Software Requirements Specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application”. The techniques listed are: (1) Formal Methods such as CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z and B; (2) Semi-Formal Methods (3) Structured Methodologies such as JSD, MASCOT, SADT, SDL, SSADM, and Yourdon.
4.4.2.10 The Joint Software System Safety Handbook (JSSSH)
JSSSH covers the entire software systems safety process from concept to system retirement. It is built around the DoD acquisition process [SSSH 1999].

4.4.3 Security Standards

4.4.3.1 ISO/IEC 15408
The Common Criteria for Information Technology Security Evaluation (known as Common Criteria) ISO/IEC 15408 [ISO 15408] is an international standard for computer security. It includes security requirements and can provide certification for complying products. The standard aims to prevent loss of confidentiality, loss of integrity and loss of availability and adds further requirements on the software development process.

4.4.4 Software Development Standards

4.4.4.1 MIL-STD-498
MIL-STD-498 [MIL-STD 498] was a United States standard aiming to "establish uniform requirements for software development and documentation." Released in 1994, it was cancelled in 1998 and replaced by [ISO 12207]. Several programs outside of the U.S. military continued to use the standard, due to familiarity and perceived advantages over alternative standards, such as free availability of the standards documents. The commercial equivalent of MIL-STD-498 (also released as IEEE-498) - J-STD-016 is no longer maintained.

4.4.4.2 Software Development Standard for Space Systems
The Aerospace Corporation developed a Technical Operating Report (TOR) which includes the Software Development Standard for Space Systems (SDSSS) [SDSSS 2005]. The standard was in response to a request from the National Reconnaissance Office (NRO) and the Air Force Space and Missile Systems Center (SMC) for a recommendation for a set of software development standards to be used as compliance documents on NRO and SMC contracts. The document is based on MIL-STD-498 augmented with requirements from J-STD-016. The document pays a lot of attention to the testing activities – this area is expanded by adding exit criteria for the various levels of software testing. Despite its name, the document is not space specific – it applies to any software development where the mission assurance needs to be addressed.

4.4.4.3 ISO/IEC 12207

The Australian Department of Defence maintains an Approved Technology Standards List (ATSL) which contains the mandatory technology standards that are to be used for all Defence Information Infrastructure (DII), Communications and Information Systems (CIS) and Management Information Systems (MIS). The ATSL is to be used by all ADO staff, Defence consultants and contractors, responsible for DIE-related capability development,
architecture development, procurement and projects. The DIE ATSL is also used to inform industry, other Government departments and allied organisations of the ADO’s position on Information and Communications Technology (ICT) standards and related DIE architecture technical standards views. ATSL contains a subset of the standards mentioned above plus other standards which are not of interest for this review.

4.5 Discussion

The CMMI [SEI CMMI] is the de-facto international reference for Software Engineering process maturity evaluation. The importance of a good process was underlined by Deming: “The quality of a product is directly related to the quality of the process used to create it.”

A good process, however, by itself does not guarantee a quality product. Many contractors advertise high maturity levels (3 and above), but still there are cases of schedule and cost overruns. As the capability models are not a standard, in some cases there is a need for enforcing stricter requirements on some aspects on the development process. Example of this is the Trillium model which pays a lot of attention to concrete development and product characteristics. Another example is using a development standard (like [SDSS 2005]) as a contractual obligation to enforce rigorous approach to testing [Eslinger 2006]. The opposite is also true – an organisation may produce high quality software without achieving a high maturity level - for example, Linux is a product of an organisation that is at SEI CMMI level 1.

Although it is unlikely that an organisation at a low maturity level will consistently produce high quality products, we cannot categorically conclude the opposite, i.e. that a good process or a successful completion of a project will guarantee a high quality new product. Therefore the result of a process assessment (together with expert judgement) needs to be used as an additional criterion which augments other evidence. [Fenton et al. 1999] considers the Cleanroom method the best available evidence relating a specific process method to defect density. Also, a relationship between the CMM levels and delivered defects was suggested by Capers Jones as shown in the table below:

<table>
<thead>
<tr>
<th>SEI CMM Levels</th>
<th>Defect Potentials</th>
<th>Removal Efficiency</th>
<th>Delivered Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>85</td>
<td>0.75</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>89</td>
<td>0.44</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>91</td>
<td>0.27</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>93</td>
<td>0.14</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>95</td>
<td>0.05</td>
</tr>
</tbody>
</table>

No one process method solves all the problems. Various methods need to be integrated and applied, such as formal inspections, FMA, testing, verification and validation, risk analysis, statistical process control, reliability modelling and prediction. The results of these methods need to be used not only for improving the product, but also for improving the process which produces this product.
Together with the software development process, the standards contribute to the quality of the product. The IEEE standards provide detailed examples, guidelines or templates for intermediate work items, such as Project Management Plans and Detailed Designs. As such, they are prescriptive in nature, and are only used as references when developing related process elements (e.g. IEEE Std. 730-2002 for Software Quality Assurance Plans, IEEE Std 828-2005 for Software Configuration Management Plans etc). Other IEEE standards provide best-practice guidance as defined by representatives of the software development industry, for example IEEE 982.1 prescribes Software Reliability Engineering measures which can be used to quantify the software quality. Schneidewind shows in [Schneidewind 2007] how standards such as IEEE 982.1 can be applied in practice as a tool complementing the process. This understanding of the role of the standards leads to their use in the certification process. In the safety critical systems, certification against defined standards is required. For example, DO-178B is the accepted means of certifying all new aviation software.

Although the use of development standards declined in the 90s [Hecht et al. 2005], with the rise of software complexity they are increasingly regarded as a tool to help improve software quality, and hence dependability.
5. Technology and Product Related Aspects of Dependability

In this chapter we will review the main approaches to evaluate and attain dependability related to specific stages of SDLC (concept and requirements definition, high-level and low-level design, coding, verification and validation, and certification).

5.1 Concept and Requirements Definition

It is recognised that the requirements definition is a major source of both testing and post-release defects. There are several issues associated with this. One is simply the quality and management of the requirements. Another is the requirements creep, i.e. the activity of evolving the system to match changes in the initial vision about the system [Abelson et al. 2006]. The result of this is cost increase, quality reduction and schedule slip. Finally, requirements can be discovered throughout the whole life-cycle, even in the phases of testing and post-release [Lutz et al, 2004]. Often these new requirements are previously unrecognised constraints or misunderstanding by the testers or users of existing requirements.

Successful development of complex systems needs a precise definition of the requirements. Requirement management tools may help with the problems of ambiguity and contradiction. As a minimum, they support the construction of structured technical documentation. A comprehensive list of requirements management tools can be found on the website of the International Council of Systems Engineering [INCOSE 2007]. Formal specification of requirements provides a basis for reasoning about the requirements and for closing the gap between the application specification and its architecture/design. A tool currently under development in DSTO - HiVe [Cant et al. 2006] - aims to provide a unified framework for capturing and supporting entire design projects. HiVe is using the proof tool Isabelle.

5.1.1 Elicitation of Dependability Requirements

Dependability requirements cover a wide scope of system aspects like failure modes and acceptable failure rates, potential hazards, recovery time and system reaction to specific failures etc. [Donzelli et al. 2005a]. Dealing with these requirements is a complex task for both the stakeholders and analysts. At this stage of development tools are needed which may help to elicit, capture and elaborate the dependability requirements.

Essentially any rigorous approach based on available models and classification will help to extract the dependability requirements. One such example of a practical framework build around the concept of a dependability issue is the Unified Model of Dependability (UMD) presented in [Basili et al. 2004], [Donzelli et al. 2005a], [Donzelli et al. 2005b].

The UMD supports the elicitation of dependability requirements by guiding the stakeholders through the process of defining hazardous events, finding their cause, investigating the manifestation and the scope of the associated failure, defining measures and specifying reactions to be triggered if a failure occurs. To illustrate the concept of
“dependability issue” we can look at the example given by the authors: if the requirement “response time less than or equal to 10 seconds” is not met, then depending on the application, this may be a performance issue, an availability issue or even a hazard (a safety issue). Simplistically, the analysis moves along the lines: specific event (e.g. denial of service) causes an issue (response time more then 10 s) which may manifest as a failure (characterised by impact, severity etc), it concerns a scope (a service or the whole system etc); the failure manifestation can be captured by a measure (MTBF, maximum cases etc) and will trigger a reaction (impact mitigation, recovery etc).

A web based tool implementing the UMD is introduced in [Basili et al. 2004], which is organised around two tables. This tool provides the user with predefined events that may be of concern to dependability, failure types and definitions. The list can be extended by the user. The tool essentially guides the user and captures the information submitted by him/her. The process goes through:

- Scope definition (data gathering) - the stakeholders identify the critical system services (based on the functional requirements) which may be relevant for dependability. This information is captured in the first table.
- Model building: for every service defined during the scope definition
  - All possible failures (issues) are identified;
  - The events causing the failure are captured;
  - The failure characterization: severity, availability impact and issue type are captured;
  - The resulting hazard is described;
  - Define or select the appropriate measure; and
  - The system reactions are defined: warning services, mitigation services, alternative services, guards and the recovery behaviour.

The UMD may be also applied for the evaluation of COTS component dependability [Donzelli et al. 2005b].

5.1.2 Defects and Risks Prediction

At the early stages of project development, estimations of the effort as well as of the product quality can be made based on historical in-process and product metrics. [Nagappan et al. 2006] reported that historical in-process and product metrics from the development of Windows XP-SP1 can be used to estimate the post release failures of Windows Server at a statistically significant level. Such metrics can also be applied to identify failure-prone modules in the new development.

It would be beneficial, however, if we could evaluate dependability as early as the requirements definition phase by measuring the artefacts produced during this phase. The main technical document produced at this stage is the requirements definitions document. Changes of requirements may introduce risks. For example, the implementation of a requirement may increase the complexity of the code, which impacts the reliability and maintainability. [Takahashi et al. 1985] identified the specification change activity (measured in pages of specification changes per KLOC) as one of three factors which significantly influence the error rate. This metric can be used as a fault prediction variable.
Another approach for predicting the defect rates is based on the risk factors associated with requirement changes [Schneidewind 2005]. Initially risk factors associated with the requirements changes were identified and studied in [Schneidewind 2001]. The factors are domain specific, but the methodology may be applicable to other domains. The study is associated with the software maintenance of the NASA Space Shuttle Flight Software, i.e. we are discussing an existing system which is evolving. Nineteen risk factors were suggested initially and after analysis four of them were identified as statistically significant: the number of modifications of the proposed change, the number of LOC affected by the change, the number of possible conflicts with other requirements and impact on the memory usage. The risk factors were used in [Schneidewind 2005] for predicting the risk to reliability resulting from making requirements changes. The paper proposes a model based on the hypothesis that the failure count, occurring as a result of requirements change, will increase non-linearly with increasing values of the risk factors. Risk predictor equations were created and the analysis of the plots showed that two of the risk factors (number of possible conflicts and impact on memory usage) could be used as good predictors of failures. This method identifies metrics for requirement changes and relates these measures to the number and type of faults inserted into the code as a result of implementing the changes. It provides a base for making decisions on the change implementation. The measures, however, are domain specific and their collection is not an easy task.

5.2 Architecture/High Level Design

The software architecture is essentially a framework for formal description and analysis of the system. Various techniques can be used to ensure rigorous development and evaluate the number of defects in the system. They include formal methods, model driven software development and architecture models analysis. Although the data we have at this stage is mainly based on historical data and expert opinion, the availability of an ADL based model gives opportunity for more accurate assessment and analysis of the software. The weight which can be attached to the output of architecture based models would depend on their accuracy i.e. how the fault behaviour is defined.

5.2.1 Model Based Software Engineering

One avenue of work which enables the rigorous development, early validation and automated analysis is Model Based Software Engineering. This approach aims to provide means for the domain expert to describe the application in their language in a way that can be used to further refine the requirements to some architecture view or any representation of the design or even generate code automatically (or under some guidance). In this way the traditional V model is transformed into a Y model [McDermid 2005].
There are many “threads” within this approach – notations to capture the system specifications and to express the Non-Functional Requirements (NFR), formal methods and tools to support the reasoning and verification against the requirements. The availability of an architecture model allows the execution of an early dependability analysis and assessment of what impact different architectures have on dependability. The developed models need to support (apart from the functionality requirements) the dependability by considering the degree of redundancy, the failure modes and appropriate methods to achieve fault tolerance.

5.2.2 Architecture Dependence Analysis

The representation of the architecture in a software architecture description language (ADL) provides a basis to analyse the architecture. [Stafford et al. 2001], [Stafford et al. 2003] investigated how a dependence analysis at architectural level using an ADL can be done, similar to the dependence analysis applied to program code. A prototype tool called Aladdin was developed to automatically perform the analysis. The dependence analysis provides answers related mainly to the maintainability of the system, like:

- Are there components that are never needed by any other component;
- If a component has to be reused, what other components will be needed;
- If a component is changed, what other components may be affected etc.

The tool takes as an input an architectural description and produces information about the direct dependencies of the various ports. This information can be used for verification to check misconnections or missing connections. Potentially, such information could be used to assess the risk of change requests in the context of the approach described in 5.1.2.

Figure 2  From V to Y development model
5.2.3 Using Scenarios for Assessing the Dependability Attributes

[Sutcliffe et al. 2002] and [Gregoriades et al. 2005] described a scenario based approach to evaluate the reliability of human based systems based on human attributes, the characteristics of the environment and the complexity of the task. The system is described using the i* language and then scenarios are created based on interviews with the users and stakeholders. The scenarios (lists of linked tasks) are converted into executable form and a Bayesian Network (BN) model is plugged into the tool. The model takes as inputs the characteristics of each task and the environment and calculates the reliability for the given scenario. In this way different scenarios can be compared to each other.

5.2.4 Architectural Patterns for Non-Functional Requirements (NFR)

At the early stages of SDLC we are also looking at methods to represent formally and map/incorporate NFR, like dependability, in the design. Design patterns are increasingly used in architecture-based software development. A field of research which concerns the dependability is the description of software architectures that include NFR. A representative sample of this work is [Xu et al. 2006], where an architectural pattern for dependability NFR is proposed. The proposed architectural pattern is independent of specific architectural styles or ADL and includes three types of architectural components and one type of architectural connector. The connectors are modelled as XML-based binding specifications which include instructions on weaving the components into the functional layer of the architecture. The pattern allows mapping of requirements to architecture components and weaving these components with one or more functional components. This approach can be used with any traditional design technique.

5.2.5 Architecture-based Software Reliability Models

The correlation between the software structure and its reliability has been pointed out in the 70s. The research in reliability models based on the software architecture is further stimulated by the advance of component based systems where complex software systems are assembled from simpler components. These components can be taken off the shelf or developed in-house.

The models can be used throughout all phases of the SDLC starting from the architecture design and ending with the operational phase. Applied during the architecture definition and design states, the models can help with exploration of early design decisions. Later on they can be used to assess the porting of the software between platforms, to assess the reliability of the system or help with reliability improvement programs. An architecture-based model needs to know the structure of the system:

- The interconnection between the components;
- The reliability of the components;
- The frequency of visits to the components during execution or the transition probabilities between the components; and
- The time spent in every component.

A classification and comparison of architecture based models is proposed in [Goseva-Popstojanova et al. 2001]. The models are grouped within three different approaches.
depending on the method of combining the software architecture with the failure behavior (see Figure 3).

- **State based models** are analytical models which assume that the transfer of control between the components of the architecture has Markov property. The assumption is that components fail independently and a component failure will lead ultimately to a system failure. State-based models use control flow graphs for representation of the architecture and investigation of reliability.

- **Path-based models** calculate the software reliability of all possible execution paths of the program. The execution paths can be identified by simulation (if the source, a prototype, or a model of the system exists). The reliability of each path is a product of the reliabilities of all components along the path and the system reliability is estimated by averaging the path reliabilities over all paths. Path-based models also assume that components fail independently. [Yacoub et al. 2004] suggest a scenario-based analysis approach for large-scale systems where the state space explosion is a hindrance for the application of Markov models. Path-based models are related to path testing where the control-flow graph of the program is used as a structural model of the architecture. The difference between path-based and state-based models is demonstrated when the control-flow graph contains loops. State-based models account analytically for the infinite number of paths that may exist as a result of loops. For path-based models, the number of paths is restricted to the number observed during program runs.

- **Additive models** assume that the reliability of each component is described by a non-homogenous Poisson process (NHPP). The system failure process is also NHPP, where the number of cumulative failures and failure intensity function are the sums of the failures and failure intensity functions of each component. They concentrate on estimating the failure intensity of the system using component failure data. The additive models do not consider the architecture of the system explicitly.

![Figure 3](image-url)  
*Classification of the architecture-based reliability models*
The models which have been studied most are the state-based models. A comprehensive unifying framework for the state-based models has been proposed in [Gokhale et al. 2006]. The models known from the literature are grouped according to three criteria:

- The model used to represent the architecture of the application, i.e. the transfer of control, which is given by the inter-components transition probabilities. Information about the time spent in each module may be also included. The architecture of the application can be modelled either as a discrete time Markov chain model (DTCM), continuous time Markov chain model (CTMC), Semi-Markov process model (SMP), Directed Acyclic Graph (DAG) or Stochastic Petri Net (SPN). The state of the model at any point of time is represented by the component execution at that time. DTMC, SMP and CTMC can be applied to model sequential application. DAG can be used to modelling concurrent applications with no loops and SPN can be applied to concurrent applications with loops.

- The model used to represent the failure behaviour of the components and the interfaces between the components.
  - Modelling the failure behaviour of a component with its reliability has the lowest accuracy comparing with the next two approaches because the component is treated as a black box.
  - Using constant failure rate provides a better accuracy of the reliability because it is assumed that there is a correlation between the time spent within a component and its failure rate.
  - Modelling the failure behaviour with time-dependent failure intensity is the most accurate representation because the usage profile is accounted for by measuring the code-coverage (which provides information about the usage of a component) or failure data.

The transition between the components can be either instantaneous or not, but in either case the failure behaviour of the interfaces can be described using one of the three models mentioned above.

- The solution method for the state-based models:
  - The first method is to combine the architecture of the application with the failure behaviour of the components – often called “Composite Method”. Most of the state-based models are composite models where the architecture of the software and the failure behaviour of the components are combined into a single model [Gokhale et al. 2002]. Such models are costly to maintain because during the design exploration process when different combinations of components need to be assessed, they have to be reconstructed and re-solved. This is complicated further by the large number of states which the software typically has. Another drawback of this approach is that it cannot solve all combinations of architectural and failure models, for example DTMC model for the architecture and constant failure rate for the failure behaviour.

  - The “hierarchical method” overcomes these problems. The reliability evaluation is decomposed into two steps – the architecture is represented by one model (e.g. a semi-Markov process), while the failure behaviour of each individual component is described by another process (e.g. Poisson or time-independent failure rate). On the negative side, the hierarchical
models are less accurate than the composite. [Gokhale et al. 2002] suggest a method for improving the accuracy of the hierarchical models by including second-order architectural statistics to improve the accuracy of the prediction.

The composite method is recommended when the accuracy of prediction is a priority, while the hierarchical method is recommended for exploration of architecture solutions.

The reader is referred to [Gokhale et al. 2006] for a detailed description of the various models published in the literature and their classification against the criteria elaborated above. Another detailed literature survey on the different Markov chain-based models can be found in [Mura et al. 1999]. They are compared to an approach based on Stochastic Activity Networks (SAN). The paper itself suggests using Deterministic and Stochastic Petri Nets for modelling and evaluation of phased-mission systems dependability. (Phased mission systems are systems which perform a series of tasks which must be executed in sequence. During operation such a system transitions through non-overlapping periods called phases). An application of SAN approach is reported in [Lunheim et al. 2006] - several SAN models of various parts of an industrial control network are combined in order to evaluate the end-to-end dependability.

The selection of a model depends on the artefacts available at a given phase of the SDLC. [Gokhale et al. 2006] provides a detailed table where for each DTMC and CTMC model the required inputs are listed. Overall, all the models need the transition probabilities between the components. Depending on how the failure behaviour is modelled, some models require the mean execution time per visit to each component; the CTMS-based models additionally require that these times follow an exponential distribution.

During the design phase, the inter-component transition probabilities can be estimated on the basis of expert opinion or historical data. As the time per visit is difficult to estimate without having the code, the failure behaviour of a component can be characterised by its estimated reliability. When the code is available, the time per visit can be measured during execution using profilers or by instrumenting the code and observing the distribution of this time. This observation makes possible to determine if a CTMS model can be applied. The data from the test failures can be used to estimate the parameters of the time-dependent failure intensity of the components. The records of failures and associated times can be used as an input to reliability growth models (RGM) to calculate the reliability of each component [Gokhale et al. 2002]. The architecture model can be further employed in the operational phase, where the failure behaviour of the components may be characterised by a constant failure rate as estimated at the end of the system test. An empirical study reported in [Goseva-Popstojanova et al. 2001] reported that the results from the architecture models (for comparatively simple application) match well the actual reliability.

5.2.6 Review Based Techniques

Various review based techniques are used for analysis and evaluation during architecture design and further phases of the SDLC.
[Firesmith 2006] presents the QUASAR system architecture assessment method, which is essentially an audit following the CMU SEI QUASAR methodology and is based upon quality cases. A quality case is a generalisation of a safety case consisting of claim, argument and evidence. The audit looks at whether the presented evidence proves that the architectural decisions and rationales lead to software architecture able to support the quality requirements.

A failure mode and effect analysis (FMEA) is an engineering technique used to define, identify, and eliminate known and/or potential failures, problems and errors before they reach the customer [Stamatis 1995]. It starts from a potential failure and traces it back its possible causes.

Fault Tree Analysis (FTA) – see 4.4.1.5 - is a deductive analytical technique of reliability and safety analysis and is applied generally to complex dynamic systems. It is a model which represents the various combinations of events (both normal and faulty), occurring in a system that may lead to a mishap. The FTA always supplements FMEA. A tree represents the relation between a single undesired event (at the top of the tree) and the various contributing causes (at the root of the tree).

Failure Mode Analysis (FMA) is a systematic approach to quantify the failure modes, failure rate, and root causes of known failures. It is usually based on historical information and in this sense it is rather a diagnostic tool.

Failure mode and critical analysis (FMCA) is a systematic approach to quantify the failure modes, rates and root causes from a criticality perspective. In all other aspects it is similar to FMEA. The main advantage is that by focusing on criticality, the so called single point failure modes can be identified. A single point failure mode is a single human error or a system fault which can result in an accident.

Safety Hazard Analysis, Fault hazard Analysis, Operating Hazard Analysis and Preliminary Hazard Analysis are all methods defining hazards and supplementing the FMEA.

Other techniques can be used as part of FMEA in order to identify the root causes, like Pareto analysis, Fishbone diagrams, brain storming etc.

5.3 Low Level Design and Coding

Once the code is written, it can be subject to static and/or dynamic analysis. Dynamic analysis (or testing) performs tests on the object code and executables derived from the source code [Giesen 1998]; it is based on the execution of the program. There are different types of testing – white box vs. black box, functional and performance etc.; we will only mention here that the results of the operational and system testing are the only means of direct evaluation of the software.
5.3.1 Static Analysis

Static analysis (sometimes referred to as static test) performs tests on the source code of software. It includes code reviews, code inspections and compilation [Giesen 1998]. Static analysis tools automate the code inspection process. They are based on derived models, i.e. an abstract interpretation, of the program behaviour. The tools need to be able to scan and parse the code to be analysed, built a representation, explore possible program executions and provide a user friendly presentation of the results.

Static analysis tools use different methods such as data flow analysis, control flow analysis, interface analysis, check of source against formal mathematical specifications (theorem proving) etc. [Ball et al. 2006] applies iterative refinement to reduce false errors. A common approach is to define typical bug patterns based on the domain, experience or published studies. Some tools include additional models of the environment - [Ball 2006] describes a tool (Static Driver Verifier) which finds kernel API usage errors in drivers written in C. The tool includes a model of the OS and the environment of the device driver and a number of API usage rules.

Finally, some tools require the introduction of annotations in the source code to facilitate extended checks. A code analysis technology designed by Microsoft Research is based around the internal Standard Annotation Language (SAL) and is included with Visual Studio 2005. SAL is a meta-language and supports static analysis tools for C/C++ code.

5.3.1.1 Programmer Errors Found

The empirical studies show that the static analysis tools are useful in finding certain classes of errors such as usage of uninitialised memory, memory leaks, buffer overflow, null pointer dereferencing etc. [Zheng et al. 2005b] point out that in their study “Possible Use of Null Pointer” was the most frequent fault (top 1) and accounted for 46% of the total faults reported by the tools. The top five faults included Possible Access Out-of-Bounds, Memory Leak, Uninitialised Variable, Pointer not Freed or Returned in addition to the (top 1) most frequent fault. Static analysis tools are particularly valuable for languages which are not strongly typed, such as C. A list of some popular C oriented tools can be found on the Spin web page [Spin 2007].

5.3.1.2 Studies on the Effectiveness of the Static Analysis Tools

The effectiveness of three Java static analysis tools on five industrial and one university projects has been investigated in [Wagner et al. 2005]. After eliminating the false positives, the tools were able to find 22% of the most severe errors, 20% of the second most severe error class and 70-88% for lower severity errors. The conclusions made by the analysis were that the errors found are a subset of the errors which can be found during manual inspection, while the dynamic tests found completely different errors. The recommendation is to use the static analysis tools as a pre-screening stage towards manual inspection. The ratio of false positives reported by the tools was too high – two thirds of the reports. The different tools “excelled” in finding the patterns they were programmed for.
The effectiveness of manual and automated static analysis was analysed in [Zheng et al. 2005a], [Zheng et al. 2005b] and [Zheng et al. 2006] based on 3 large scale C/C++ projects in the Nortel Networks group (over 3 million LOC in total). Three third party commercial static analysis tools were used in the development process. The selected projects provide a good base for comparison, as some projects passed through automated analysis only, some through manual inspection and some through both automated and manual inspection. Based on the Orthogonal Defect Classification (ODC) scheme, the results indicated that the static analysis tools were effective in identifying two of the eight ODC defect types: Assignment and Checking. These types of errors are injected in the coding phase or in the low level design phase of the SDLC, i.e. static analysis tools are effective at identifying code-level defects. The manual inspection most often identified Algorithm, Documentation and Checking faults (approximately 90% of the faults found). 88% of the tests failures belonged to the Function and Algorithm type and all customer failures (100%) were classified as Function and Algorithm. Table 4 and Table 5 illustrate the observations in [Zheng et al. 2006]. The authors also classified the errors according to their severity. Overall, the conclusion that can be made is that static analysis tools find errors of lower severity, which are injected mostly during the coding phase (and less in low design). The defect removal rate of static analysis tools was essentially the same as that of inspections, while the defect removal yield of execution-based testing was two to three times higher. Based on the type of possible errors flagged by the static analysis tools, the authors conclude that a large percentage of the errors could cause security vulnerability. With regards to the cost, static analysis tools had a lower average fault detection cost than the manual inspections, but still of the same order.

<table>
<thead>
<tr>
<th>Defect Type</th>
<th>ASA [%]</th>
<th>Inspection [%]</th>
<th>Test [%]</th>
<th>Customer [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>0</td>
<td>1.09</td>
<td>55.73</td>
<td>69.70</td>
</tr>
<tr>
<td>Assignment</td>
<td>72.27</td>
<td>4.37</td>
<td>3.82</td>
<td>0</td>
</tr>
<tr>
<td>Interface</td>
<td>0</td>
<td>0.87</td>
<td>0.20</td>
<td>0</td>
</tr>
<tr>
<td>Checking</td>
<td>27.73</td>
<td>20.52</td>
<td>0.80</td>
<td>0</td>
</tr>
<tr>
<td>Timing/Serialisation</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Build/Package/Merge</td>
<td>0</td>
<td>1.77</td>
<td>1.81</td>
<td>0</td>
</tr>
<tr>
<td>Documentation</td>
<td>0</td>
<td>35.37</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Algorithm</td>
<td>0</td>
<td>36.03</td>
<td>37.63</td>
<td>30.30</td>
</tr>
</tbody>
</table>
Microsoft is using static analysis as part of their development process [Nagappan et al. 2005]. The PREfix tool processes the program call graph bottom-up and performs symbolic evaluation on a fixed number of paths across function boundaries in a complete C/C++ program. It checks for NULL pointer de-reference, the use of uninitialised memory, double freeing of resources etc. The analysis requires a server-side solution. The PREfast tool is aimed for the developer desktop that looks for errors locally. It is inexpensive requiring a negligible percent of compile time. Analyses are based on pattern matching or local dataflow analysis. MaX is a dependency analysis tool which constructs a graph of dependencies between binaries (DLLs) in the system. The tool rejects code that breaks the component architecture of the product. Static analysis tools found over 12.5% of the defects fixed pre-release in Windows Server 2003. [Das 2006] reports that the Center for Software Excellence, which is involved in program analysis, in total found more than 100,000 fixed bugs in Windows Vista. For Vista:

- The annotation cost of applying SAL (used in the PREfast tool) was 100,000 parameters requiring annotations;
- Defect detection value 1 buffer overrun exposed per 20 annotations; and
- 9.4 out of 10 buffer accesses were validated.

Another application of static analysis tools is investigated in [Nagappan et al. 2005], where a strong positive correlation is proved between the static analysis error density and the pre-release fault density. [Zheng et al. 2005b], however, reports that the number of static analysis errors itself is not a very good measure of fault-prone module.

[Littlewood et al. 2000a] quoted an analysis for the safety system software of the Sizewell nuclear reactor. Some problems (without significant safety impact) were discovered, while parts of the system were too complex for the tool to handle. At the end, the contribution of this evidence to the safety case was based on experts’ judgement.
5.3.1.3 Limitations of the Static Analysis Tools

The major issue with static analysis tools is the false error reports. Nortel, as reported in [Zheng et al. 2006], used pre-screen teams which are not domain experts but programming language experts. They analysed the reports from the static analysis tools and created a short list of suspected errors, which was reviewed and then passed to the developers for investigation and fix. Microsoft, which invests heavily in program analysis, has a research ongoing on improving the quality of the tools in order to increase the false reports. [Ball et al. 2006] describe using iterative refinement to reduce the number of false errors. As mentioned before, the reports from the PREfix tool are entered automatically in a bugs database, which is an indication of the level of trust in the tools output.

Another impediment for the adoption of the static analysis tools as a routine technique by the developers is that the defect reports are not accompanied by user scenarios exposing the error. An approach which can possibly in the future rectify this is the combination of static methods for program analysis and dynamic testing. The method suggested in [Yorsh et al. 2006] is in this context, but has a particular objective. It is oriented towards finding a proof of correctness rather than detecting errors. The method monitors concrete executions of the program and gathers information of the states of the program. Then a theorem prover is applied to check that all potential executions of the program were covered. [Smaragdakis et al. 2007] and [Casanova et al. 2005] are using the Extended Static Checker for Java ESC/Java (a compile time program checker) to obtain abstract error conditions. Such conditions are constraints on program values and an example of such an integer constraint can be \( x > 2 && y > 3 \times x \) or object reference constraint \( p.next==q \&\& q.prev==null \). These error conditions are constrained using a combination of constraint solving tools (e.g. \( x==3 \&\& y==10 \&\& p.next.prev==null \)). The last step is applying these specific error conditions as an input to a tool which generates test cases for JUnit (a Java unit testing tool). Another tool at Microsoft - MUTT (MSIL Unit Testing Tools) is aimed at automatic generation of a set of unit tests based on symbolic analysis of the code. The tool then further interacts with the programmer in order to increase the coverage of the set of tests.

5.3.1.4 Adoption of Static Analysis Tools

Applying static analysis tools to the code base is becoming more commonplace. The Department of Homeland Security has provided a grant to Coverity to improve their source code analysis tool and to run scans of code contributed to popular open-source projects. The data is meant to help secure open-source software, which is increasingly used in critical systems [Evers 2006]. Static analysis tools are part of the development process in Microsoft [Larus et al. 2004], [Das 2006]. A suite of static analysis tools is being developed which includes: a global inter-procedural symbolic evaluator (PREfix) for detecting memory usage errors; a global inter-procedural path-sensitive dataflow analysis (ESP) for detecting security vulnerabilities and concurrency defects; a local intra-procedural abstract interpretation with widening and linear constraint solving (espX) for detecting buffer overruns; a global inter-procedural dataflow analysis (SALinfer) for inferring function pre-conditions and post-conditions; and a formal language of function specifications (SAL)
that is understood and enforced by all tools. SAL is now available via the Visual Studio compiler [Das 2006].

A comprehensive list of C tools is listed on the Spin web page [Spin 2007].

5.3.2 Mining Versions History

A similar approach to the automated static analysis is leveraging the versions history. The approach has been used for understanding a program and/or support navigation in the code. [Zimmerman et al. 2005] reported on the development of a tool (ROSE – Reengineering of Software Evolution) aimed at detecting coupling between modules of low granularity (functions, variables, classes). They use the term evolutionary coupling to denote dependencies resulting from the code evolution (e.g. “whenever the database schema was changed, the sqlquery() method was also updated”) as opposed to logical coupling determined by the program analysis. The tool has access to more information comparing with the program analysis because it processes the older versions in addition to the latest one. It aims to predict likely changes based on similar changes implemented earlier in the code base. As a result it can suggest to the user certain points in the code that may need an update after he/she did a change somewhere else. For stable systems like GCC, ROSE gives more precise recommendations. In 63 percent of all transactions, the tool made a recommendation which contained 45 percent of the related items with a precision more than 30 percent. For rapidly evolving software the tool was most useful at file level (as it cannot predict new functions), i.e. the tool showed to be most useful for maintenance. Overall the results are modest from the point of view of practical application on a development project, but it is an interesting approach.

5.3.3 Notes

There is not a single fault detection technique capable of addressing all fault-detection concerns. As pointed out in [Zheng et al. 2005a], static analysis tools cannot replace manual inspection, but they can add value to the development chain by detecting certain types of faults (primarily simple faults) and enforcing adherence to coding standards and guidelines. Static analysis tools are a useful complement to inspections which cannot check all possible execution paths in the code. Static analysis tools can also find errors on paths which were not covered by testing or did not manifest during testing. Theoretically at least, if the automated static analysis does not find an error of a particular kind, then the analysed portion of the program can be considered as free of this type of error.

As pointed out in [Das 2006], automated static analysis has its own niche because:

- Inspection/review has too many code paths to think about;
- Testing is an inefficient tool for finding simple errors; and
- Local program analysis lacks calling context, which limits accuracy.

Testing cannot provide a proof of correctness of the code. It is able, however, to find deeper functional and design errors and for the foreseeable future, testing is the primary technique to check the correctness of the software [Larus et al. 2004]. As mentioned earlier, work is ongoing in the field of applying static analysis engines in tools for generation of (predominantly unit level) tests.
In terms of cost, [Zheng et al. 2006] reported that the ROI of static analysis tools compared with inspection is the same. The quantity of false reports generated by static analysis tools, however, and the lack of support for transition to test cases, are deterrents for the wide adoption of static analysis tools. Also, static analysis cannot easily handle some coding techniques like recursion and pointers.

5.4 Testing

Testing remains the best practical method for faults detection. Operational testing in particular is the only means of direct evaluation of reliability. Statistical methods using the results of operational testing allow estimation of measures such as Mean Time Between Failures (MTBF), reliability function (i.e. probability of failure-free operation for a given time t), failure rate, etc. Statistical methods require: (1) a means of generating test data that is statistically representative of operational use; (2) an oracle to allow unacceptable (failed) output to be detected. The observed failed outputs and the time spent on verification are used to estimate and predict reliability.

It is worth stressing that the accuracy of the methods and approaches described below depend on the correct emulation of the operational profile.

5.4.1 Analogy Models

Analogy Models is an approach relying on test results from previous projects. The number of residual defects can be predicted on the basis of comparison between the project under investigation and previous projects which have similar characteristics. The idea is that the defect rate is tracked through all phases of the SDLC. To predict the defect rate $d_n$ at phase $n$ the actual sequence $d_0, d_1, \ldots, d_{n-1}$ is compared with the analogous sequence from a similar project. Statistical exploration techniques can be used to estimate the expected number of defects. This approach provides good results in a stable and repeatable testing and development environment. If a local data base is not available, then published benchmarks can be used [Fenton et al. 1999].

5.4.2 Using Test Metrics

Test coverage measures can be used to predict the number of defects. The proportion of coverage (statements, branches etc) for a given set of test cases is a metric called Test Effectiveness Ratio (TER). As the TER increases, it is assumed that the number of faults found approaches the total number of faults in the module. [Veevers et al. 1994] have derived a relationship between reliability and test coverage. They also noted that the structure of the code affects the strength of this relationship. [Karcich et al. 1996] used the data from a system of several hundred KLOC C code to investigate the correlation between the MTTF and the code coverage and report that the SRM may be able to improve the accuracy by accounting for code coverage.
5.4.3 Software Reliability Growth Models

The results from operational testing can be fed into RGM. These models apply mathematical and statistical methods to model past failures in order to predict future behaviour. The name of the models comes from the assumption that the reliability grows as faults are identified and fixed in operational testing.

Some of the RGM can be obtained analytically by making assumptions about the software testing and debugging process. Others are more fundamental and are based on modelling of the testing process itself.

Reviews of the various approaches to creating software reliability models, the model assumptions, the data required for implementation and the precision of the results can be found in [Farr 1983], [Lyu et al. 1995] (chapter “Software Reliability Modelling Survey”), [Musa et al. 1987] and [Musa et al. 1990]. Here we will only provide a brief description of the classifications and rationale behind the more popular models based on the references above. The purpose is to give an idea of the numerous approaches, and, based on this, to note the main tendencies in models and associated tools which may have practical applications for AMS branch projects.

[Farr 1983] classified the different approaches to the RGM as Time Domain approach, Data Domain approach and Seeding and Tagging approaches.

5.4.3.1 Time Domain Approach

The Time Domain approach is to model the error generation over time. Either the failures per time period or the time between failures is used. Time Domain models have received most attention. They are able to predict the time until the next failure or the number of failures to be found in the next interval of testing. They were motivated mainly from hardware reliability concepts. The difference is that hardware failures are mainly a result of wear, whereas the software failures result from errors of design and implementation. Another difference is that duplicate software copies produce identical results. The models from the Time Domain category can be grouped in three types: Classical, Bayesian and Markov [Farr 1983].

5.4.3.1.1 Classical Models

The Classical models stem from the hardware reliability concepts being adapted to software models. Some failures are not simply a “combinatorial” result of a specific input to the program, but the program may act as a state machine and several steps need to be taken before a failure manifests itself. Also, because the input space of a program may be quite large, the operational usage of a program appears like a randomisation over the input space. This causes the fault occurrence to appear like following a probabilistic model over time. The RGM is based on the characterization of this probabilistic nature. Many of the models assume that the events of failure over a given period of time have a Poisson distribution or that the time intervals between successive failures have an exponential distribution. (The exponential distribution assumed in many models is the reason to name this type of models classical.) Another group of models assume that the per-fault fault distribution follows the Weibull and gamma distributions.
Historically, the first study of software reliability (by Hudson, 1967) viewed the software development as a birth and death process, where fault generation was a birth, and a fault correction was a death. The state of the process at any moment of time was defined by the existing number of faults. The assumption was that the rate of fault detection was proportional to the number of remaining bugs and it was shown that the number of detected faults followed a binomial distribution. The next step (Jelinski and Moranda 1972; Shooman 1972) was the assumption that every bug fix leads to a change in the per-fault hazard rate by a constant amount (the rate was fixed between bug fixes). Several more models in the 70s looked to increase the dependency on time, for example (Schick and Wolverton, 1973) assumed that the hazard rate was proportional to the product of the number of remaining faults and the time. The idea is that as testing progresses, more attention will be paid to areas of the code where bugs were already found and the chance of finding new bugs increases. Schneidewind initially suggested an empirical approach, where every project should find a fault distribution which best fits the concrete project. He also suggested that recent faults are of more use than earlier faults and that forecasts need to be corrected from the actual data. Musa (1975) suggested that the processor time was the best measure of stress on the software. As a result, when rates were based on execution time, it was observed that the bug correction rate was proportional to the bug detection and the variability of the models from project to project did not occur. The Geometric model (proposed by Moranda as an extension of the Jelinski-Miranda model above) does not assume a fixed finite number of faults in the software, nor does it assume that the bugs are equally likely to be found. As testing progresses and more bugs are found and corrected, it is increasingly more difficult to find new bugs. The Musa-Okumoto logarithmic Poisson model (1984) has been extensively applied in practice; it was one of the four selected models in the American National Standard Recommended Practice for Software Reliability (ANSI/AIAA R-013-1992).

The Classical models are very sensitive to violation of their assumptions and especially sensitive to increasing failure rate (which may be caused by non-uniform testing or introduction of new errors during fixes). They cannot handle a lack of independence among failures, i.e. when discovery of one bug leads to quick detection of other associated bugs or “hidden” behind it bugs.

5.4.3.1.2 Bayesian Models

The Bayesian models view the reliability growth in contrast with the classical “frequentist” approach. While the classical models allow change in the reliability of a program only when a fault was found and fixed, the Bayesian models take the subjective view that if no error was discovered during testing, then the reliability for such a program should increase. The reliability of the software depends on the number of failures and length of failure-free testing time. The Bayesian models suggest that the faults may not be equal. A program with a fault in a frequently executed part of the code is less reliable than a program with two or more faults in rarely executed area of the code. Thus the Mean Time To Failure (MTTF) is an important measure in this framework.

The Littlewood-Verrall reliability growth model is probably the best known of this class of models and it is the one recommended in ANSI/AIAA R-013-1992. Contrary to the classical approach, where the hazard rate is a function of the number of remaining faults, here the hazard rate is modelled as a random variable. This allows for the probability that
the software may become less reliable after a bug fix as a result of introducing new faults. This model formed a base for other variants and extensions.

The Bayesian models offer a more realistic approach than the classical ones. They suggest that past information (e.g. from similar projects) should be incorporated into the model in the form of prior distribution in order to estimate current and future reliability. This is both one of the strengths and the weaknesses of the Bayesian methodology.

5.4.3.1.3 Markov Models
The Markov models can be used to model the random behaviour of software in time. The future of a Markov process depends only on its present state and is independent of its history. This assumption may be applied for a software failure process as it is mainly dependent on the number of remaining faults and the operational profile. In general, Markov processes are characterised by the amount of time spent in a state and the transitions between states. Littlewood’s Semi-Markov model considers the structure of the program in defining its reliability. The program is represented as a set of modules which exchange control between them following a semi-Markov law. It is semi-Markov, because the time spent in any module can be taken as a random variable.

In [Musa et al. 1987], several of the Poisson-type models are discussed using Markov processes. The existing Markov models are providing insight, but as far as it concerns practical applications, they are very complex and difficult to apply.

5.4.3.2 Error Seeding and Tagging Approach
The error Seeding and Tagging approach is based on the concept of error introduction into software. Based on the proportion of injected to inherent errors found during testing, a conclusion is drawn about the total number of undetected faults. The problem with this method is that in practice, the injected errors are different in character and distribution from the inherent faults.

5.4.3.3 Data Domain Approach
The Data Domain approach estimates the reliability of a software program strictly on the bases of successful runs compared to the total number of runs. If time is used in any of these models, it represents the total elapsed time of a test run and not the times when the failures occurred. This approach is reduced to a method of selecting the test inputs according to the probability distribution of the intended operational profile. The input space is portioned into regions which are then assigned probabilities according to the operational profile. The major weakness of the Data Domain Approach is the partitioning of the input space and the probability assigned to each partition.

- In the simplest model, if $S$ runs are successful out of totals of $N$ runs, then the current program reliability $R = S/N$. This basic formula was further refined/modified to include additional information. Brown and Lipow modified the formula to account for the characteristics of the tests’ input space. The input space was divided into “homogenous” regions (homogenous in the sense of fault generation).
• Nelson’s model also takes into account the input space. It defines the program as a computable function defined on the set of all input values. The input values are chosen according to some operational requirement. This model was further expanded based on the assumption that the runs are not independent of each other and some variables may, for example, be chosen in ascending order from run to run. The input space is divided in regions similar to the Brown and Lipow’s model, but the regions are based on logic paths versus regions which are homogenous in the sense of fault generation.

• A model by Sugiura, Yamamoto and Shiino divides the input space in user space (where the inputs for testing are taken from) and the rest of the possible entries (which are not checked during verification time).

The same models described above are grouped in a slightly different, but principally similar schema in [Gokhale et al. 1996].

Another grouping is given in [Rakitin 2001]:

• Empirical models (time domain, data domain, axiomatic complexity, error seeding) and
• Mathematical (behavioural models) - they model the reliability growth as a stochastic process: Markov models, Petri nets.

5.4.3.4 Software Reliability Models – Comparison and Selection

After the period of offering and investigating new RGM, the attention shifted in the 80s to the comparison of SRM. This is not to say that new SRM are not investigated anymore, on the contrary - the existing stochastic models are extended and new approaches are suggested. A lot of studies are being conducted into various regression approaches in order to develop metrics-based RGM.

The definition of taxonomy helped to highlight the relationship between the models and compare them. An early classification scheme was proposed in [Musa et al. 1983] and [Musa et al. 1987]. It allows the models to be compared and also visually represented in a table.

All models make some assumptions and this is why the selection of an appropriate model depends on these assumptions. There are three types of assumptions [Rakitin 2001]:

• Universal (times between failures are independent, testing is representative of actual use, faults are of similar severity, time is used as basis of failure rate);
• Criteria assumptions made by some models (the number of potential faults is fixed and finite, detected faults are fixed immediately, individual fault occurrence times are recorded (vs. fault occurrences grouped by time intervals); and
• Particular assumptions made by individual models (e.g. fixed number of errors in the code, no new errors are introduced through the bug fixing process, the program size is constant – no new code is added, error detection is an independent process, testing is performed in a manner similar to the intended usage, error detection rate is similar to the number of errors remaining in the code).

Each model also has its specific data requirements.
Our observation is that no one best model can be recommended, i.e. each model has advantages and disadvantages and needs to be selected based on the specifics of the project. [Farr 1983] summarizes the results of applying the models to actual software failures data. Different studies included different models, so generalizing across all models would not be productive, but in general, the Geometric models are better when the MTBF are concerned and they provide better predictive results if the times between failures are known. When the error count per time interval is known, the Schneidewind Model is preferred. The Jelinski-Moranda Model and Schneidewind model give similar results, but the former one is very sensitive to changes in data while being reasonable for smaller projects. A serious problem noted was the lack of convergence of parameter estimates - the reason for this is assumed to be violations of the assumptions of the model. The convergence problem appears to diminish with the increase of the testing period. In principle, the models grouped as Classical are sensitive to violations of the assumptions on which they are based and especially sensitive to increasing failure rates.

Usually, data domain models require a long period of testing based on a representative input set. The time domain models are more economical to apply [Gokhale et al. 1996]. Some models have been tried more extensively in practice, like the Musa-Okumoto logarithmic Poisson model, the Schneidewind model or the Littlewood-Verrall model. They have been recommended in ANSI/AIAA R-013-1992 and should be the first candidates to be considered when selecting an initial set of models to apply to a project.

[Rakitin 2001] suggests a process to select a model appropriate for a project:
1. Select a group of candidate models which are robust to the violations of the assumptions.
2. Compare models’ assumptions with the software development process and narrow the list of candidates to two or three.
3. Identify the data requirements for each selected model and determine how to collect the data.
4. Once the data have been collected, apply the model to the data.
5. Perform a goodness-of-fit test and determine if each model meets the defined goodness-of-fit criteria.
6. Rank the models based on 5.
7. Collect data.
8. Create periodically graph showing progress toward the reliability goal.
9. Validate the results against actual experience.

The following factors need to be considered when selecting a model:
- The model needs to be simple.
- It is based on sound assumptions.
- There is no one best model, i.e. each model has advantages and disadvantages.
- Accuracy of models varies widely.
- Match the model assumptions to the software development process.
- Try as many models as possible and select the one that matches the observations best.
- Data collection is usually the most difficult problem.
- Software reliability growth modelling, like anything else, needs to be subject to continuous process improvement.
Finally, it should be noted that in order to use the number of defects/faults found during test as accurate predictors of the number of remaining faults, the system needs to be tested under conditions as close as possible to the operational profile. It is important to run operational tests which are representative.

5.4.3.5 Tools Implementing SRM
A number of free tools have been developed to help the practitioners by implementing several reliability growth models behind a Graphical User Interface (GUI) and hiding away the theoretical details. The PETERS tool developed by the Centre for Software Reliability, City University London, lacks in the user interface and thus requires the user to have good understanding of the modelling methodology. The CASRE tool is practically available for private use only outside of USA (according to its current licence). The SMERFS3 tool developed by the US Naval Surface Warfare Center, Dahlgren, VA is the best available option. It has been successfully used in a feasibility evaluation of the methodology [Wallace et al. 2001].

5.4.3.6 Notes on the Reliability Growth Models

Many sophisticated RGM have been developed. Their aim is to predict future failure behaviour from observation of the past. The following issues are identified with this method:

- Some fundamental assumptions made in most RGM such as “failures are traced back to unique faults in a module” and “components fail independently” do not always match reality. As pointed out in [Goseva-Popstojanova et al. 2006] for several empirical studies, these assumptions are true for approximately two thirds of the failures, but the rest of the failures can be mapped to more than one software component.

- Another limitation of the method is demonstrated by the observation that the further the project goes into testing, the more time it takes to find new bugs – the law of diminishing returns [Littlewood 2000a]. Some defects may need thousands of years of operational use in order to demonstrate their presence.

- The RGM need failure data which is not always readily available.

- Some models (especially Data Domain models) require large number of runs in order to establish statistical confidence in the results.

- The output from a tool (or a model) needs to be interpreted by an analyst who relies on his/her knowledge about the project. A good understanding of statistics and the methodology is required in order to decide which results are meaningful. As a result the time to interpret the results may be “overwhelming” [Wallace et al. 2001]. Using RGM in an organisation requires planning for training, for resources and data collection and is a major effort.

The conclusion is that the kind of evidence to be obtained from the RGM is relatively weak and none of the models can be used as an absolute point of reference. Therefore, the results need to be validated against other metrics and can contribute to a dependability evaluation framework.
5.4.4 Evaluation of Dependability Based on Software Metrics

Fault introduction depends mostly on the characteristics of the developed code and the development process. Therefore, the code size and complexity measures can be used as an indirect measure of reliability. Empirical evidence confirms the relationship between the faults found during testing/operational use and the measures of size or complexity. Different product metrics, methods and modelling techniques have been studied. An extensive discussion on the usage of complexity metrics is provided in [Fenton et al. 1999] and [Fenton et al. 2000a].

There are two issues that need to be considered in relation to metrics:

- Definition of the measure – different measures describe the product (specifications, design, code, test), the process or the resources (human resources, the organisation or the technology); and
- How the measures are collected and used.

5.4.4.1 Definition of the Measure

5.4.4.1.1 Size Measures

The first key metric was the lines of code metric (LOC, or KLOC for Kilo LOC). It was used to measure productivity at first (LOC per man-month) and later quality (defects per KLOC). The earliest studies showed that there is linear dependency of the total number of defects (D) found during testing and post-release from some simple metrics. A work by Akiyama (done at Fujitsu) predicted defects from KLOC to be 23 defects per 1 KLOC [Musa et al. 1987]. The proposed equation was:

\[ D = 4.86 + 0.018L \]

5.4.4.1.2 Complexity Measures

With the increasing diversity of programming languages the interest in measures reflecting various aspects of complexity grew. One of the early classifications of complexity metrics groups them in lexical metrics, connectivity metrics and semantic metrics [Henry et al. 1981]. Lexical metrics, such as Halstead metrics and McCabe cyclomatic complexity, are based on the count of various lexical tokens. Connectivity metrics are based on the degree of interconnectivity among system components (e.g. Henry and Kafura measure). Semantic metrics are based on the principle that connections between components are based on shared assumptions and entropy may be used to assess this structure.

- Halstead, [Halstead 1977], proposed a number of size metrics defining a quantitative measure of complexity directly from the operators and operands in the software module. They have been used as maintenance metrics, for evaluating the development effort after the code has been developed and as predictors of program defects.

The Halstead measures are derived directly from the source code:

- \( n_1 \) = the number of distinct operators
- \( n_2 \) = the number of distinct operands
- \( N_1 \) = the total number of operators
- \( N_2 \) = the total number of operands
From these numbers, five measures are derived as shown in Table 6.

Table 6 Halstead complexity measures

<table>
<thead>
<tr>
<th>Measure</th>
<th>Symbol</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program length</td>
<td>N</td>
<td>N = N1 + N2</td>
</tr>
<tr>
<td>Program vocabulary</td>
<td>n</td>
<td>n = n1 + n2</td>
</tr>
<tr>
<td>Volume</td>
<td>V</td>
<td>V = N * (LOG2 n)</td>
</tr>
<tr>
<td>Difficulty</td>
<td>D</td>
<td>D = (n1/2) * (N2/n2)</td>
</tr>
<tr>
<td>Effort</td>
<td>E</td>
<td>E = D * V</td>
</tr>
</tbody>
</table>

The formula suggested by Halstead was

\[ D = \frac{V}{3,000} \]

where 3,000 is the mean number of mental discriminations between the decisions made by the programmer.

Despite the opposing opinions on the worth of Halstead measures, the measures are still used in many current studies.

- McCabe cyclomatic complexity (CC) – also referred to as program complexity or McCabe’s complexity - is another metric which aimed to overcome the restrictions of the size based metrics and to be language independent. CC measures the number of linearly-independent paths through a program module. The measure is essentially still a size metric, but it is considered that the complexity number is a stronger measure of the structural complexity than the size in KLOC. The metric was originally applied for formulating test strategies.

The CC of a software module is calculated from its control flow diagram, which is a design artefact and it means the CC can be derived before the coding phase:

\[ CC = E - N + p, \]

where
- \( E \) = the number of edges of the graph
- \( N \) = the number of nodes of the graph
- \( p \) = the number of connected components

A set of threshold values for the CC is often used to evaluate risk as shown below in Table 7.

It is interesting to note that this metric was taken off the list of recommended measures of dependability in the latest release of [IEEE 982.1] with the note that these values do not have applicability to reliability and dependability.
Table 7 Threshold values for the cyclomatic complexity

<table>
<thead>
<tr>
<th>Cyclomatic Complexity</th>
<th>Risk Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>a simple program, without much risk</td>
</tr>
<tr>
<td>11-20</td>
<td>more complex, moderate risk</td>
</tr>
<tr>
<td>21-50</td>
<td>complex, high risk program</td>
</tr>
<tr>
<td>greater than 50</td>
<td>un-testable program (very high risk)</td>
</tr>
</tbody>
</table>

- Bowles metrics measure the module and system complexity in terms of coupling via parameters and global variables.
- Ligier metrics measure the modularity of the structure chart.
- The following three metrics are specialized measures that are used in specific situations [VanDoren 2000]:
  - Essential complexity - measures how much unstructured logic exists in a module (e.g., a loop with an exiting GOTO statement).
  - Design complexity - measures interaction between decision logic and subroutine or function calls.
  - Data complexity - measures interaction between data references and decision logic.

5.4.4.1.3 Object Oriented Design (OOD) Measures

Various metrics were suggested for OOD. Object oriented metrics are different from the metrics described above, because of the characteristics of the object oriented approach - localization, encapsulation, information hiding, inheritance, and object abstraction techniques [Berard 1995]. Comprehensive resources on the subject of object-oriented metrics are provided on the Open Seminar web site ([Williams et al. 2005]), the Karakap’s and Sultanoglu’s website [Karakap et al. 1998] and [Cetus 2005], to name but a few.

5.4.4.1.4 Function Point Analysis

Another trend was the development of measures independent of the programming language, coding style or architecture design, e.g. measuring a size of software in terms of its ‘functionality’ as opposed to its physical components. It was formulated by Allan Albrecht of IBM in 1979 who proposed a method called Function Point Analysis (FPA). The FPA method uses functional, logical entities such as inputs, outputs, and inquiries which are related closer to the functions of the software as compared to the size measures. FPA has been renamed functional size measurement, but FPA remains the more commonly used term. Basic function points are categorized into five groups: outputs, inquiries, inputs, files, and interfaces.

5.4.4.1.5 Multivariate Approaches

It was noted that using a single metric as a predictor is seldom useful. As a result, sets of metrics were suggested and investigated as predictors for fault proneness. Typically they are a mix of measures of product attributes like coupling, size, data structure and control-flow structure. Examples of such sets of metrics can be found in [Nikora et al. 2003a], [Lanubile et al. 1997]. Both studies (e.g. [Nikora et al. 2003b]) and practitioner’s experience
[Berard 1998] point out that the metrics are usually highly correlated. Therefore, statistical methods are used to derive the distinct orthogonal sources of variation and map the raw metrics onto the uncorrelated metrics. These derived metrics present the same information which is contained in the raw metrics. For example, in [Nikora et al. 2003b] the original set of 12 metrics is reduced to three derived metrics reflecting the structural change of the software during its development. The derived metrics are further used to predict the error-proneness of the modules. This approach was used in a software evolution process to estimate the rate with which faults are introduced between successive revisions of software. In another study ([Nagappan et al. 2005]) discriminant analysis is applied to the results produced by static analysis tools to distinguish between high and low quality components.

Various statistical methods are used to process the collected metrics and formulate a prediction model. Among them are multiple linear regression analysis, discriminant analysis, logistic regression, logical classification models, layered neural networks etc. **Principal component analysis** (PCA) is usually applied as the first step in order to simplify the data collection. It provides a concise overview of a dataset and is very powerful at recognising patterns in data: outliers, trends, groups etc. The PCA is used to extract principal components, i.e. expressing two or more variables by a single new derived factor. In this way, the PCA is used here as a data reduction method, that is, as a method for reducing the number of variables, which in our case is the number of metrics we selected to characterize the software. Overall, there is a consensus that the accuracy of prediction using the derived set of metrics is similar to the one when the raw metrics are used.

Aiming to further simplify the structure of software complexity, a cumulative measure of complexity was suggested. [Nikora et al. 2003b] uses the Fault Index, where the Fault Index is a weighted sum of a set of uncorrelated attribute domain metrics. Each raw metric contributes proportionally to the Fault Index, depending on the unique variation contributed by the raw metric.

The application of these methods to real life projects is hindered by the necessity to collect many metrics and the need for tools to automate the process of data gathering.

5.4.4.1.6 Other Non-code Measures

The low quality of requirements specifications, the change of the specifications and lack of skills in the development team have a negative effect on the quality of the software product. [Takahashi et al. 1985] suggested quantitative measures which can be used as fault predictors early in the development. Based on the analysis of 30 projects, three factors were identified to significantly influence the defect rate:

- Specification change activity, measured in pages of specification changes per KLOC;
- Average programmer’s skills, measured in years; and
- Thoroughness of design documentation, measured in pages of developed (new plus documented) design documents per KLOC.
5.4.4.2 How the Measures are Collected and Used

The value of the metrics is in their application – helping to assess risks and resource/time requirements. [Fenton et al. 2000a] considers the most important advances in the metrics management to be:

1. The body of knowledge and practice concerning the implementation of metrics programs; and
2. The use of metrics in empirical software engineering, more specifically the development of benchmarks and validation of specific software engineering methods and tools.

One of the major issues for producing adequate predictions is the quality of metrics:

- To start with, the quality of the software product reflects the state of the technology, the organisation structure and culture, the environment in which requirements were specified, the competency of the team etc. All these factors are volatile and may change from project to project (especially for projects with long life span) and even more from organisation to organisation. Therefore, the historical metrics will be more relevant to the organization which produced them, than for another organisation. This is why the need for a local metrics database is often mentioned (e.g. [Fenton et al. 2000a], [Lanubile et al. 1997]). It may be expected that organisation with higher maturity level will be more rigorous in metrics collection as a result of including the metrics collection in their process and budgeting for these activities; the metrics would be more consistent as a result of applying organisation wide standards, and the metrics will be more useful because of the repeatability of the engineering process.

Measurement is also a process, as underlined in [Nikora 2003b]. Whenever a change is done in a software system, the updated modules have to be re-measured.

- Another issue is the subjectivity of the team members gathering the metrics – the classification of the type of faults and their severity may vary for different persons, the traceability of failure reports to faults may vary, the rigor with which failure time is captured also may vary. Studies often have to use data from projects that have been completed and assumptions may have to be made about aspects of the collected metrics.

Measurements must be done in a consistent manner. One way to achieve this is to automate the metrics collection and make this part of the development process. For example, [Nikora 2000] reports that measurement and analyses are performed automatically at check-in into the CM library.

In order to compare measurements in several development efforts, organisational standard/guidelines can be used. [Nikora et al. 2003a] describes the standard for faults identification and counting applied for their model. They have also developed a standard for structural measurements and a practical framework for automating these measurements [Nikora et al. 2003b].
Another issue is selecting a right approach for the concrete software product/project.

- [Fenton et al. 1999] concludes that the existing models are not able to accurately predict the number of defects based on size and complexity metrics alone. Also, they pointed out that the count of defects found before release is not a good indicator of the number of defects remaining. Post-release, a module may have no remaining defects as a result of better test coverage (hence possibly more pre-release defects), or it may have defects as a result of poorer quality of implementation (again more pre-release defects may be assumed).

- Multivariate approaches have provided useful insights, but at the same time regression models often lead to misunderstandings about cause and effect. A correlation between two measures is not necessarily a causal relationship. In this case the variations determined by regression approaches are explained by other measurable variables.

Lanubile and Visaggio in [Lanubile et al.1997] attempted independently to validate in an empirical study several of the more popular statistical predictive methods reported in the literature (discriminant analysis which has been applied to detect fault-prone modules; logistic regression and logical classification models used to identify high-risk components; and layered neural networks). Based on the number of software components incorrectly classified as high risk or low risk, they concluded that none of the methods was able to discriminate well enough between software components with faults and components without faults.

- Most of the techniques cannot provide quantitative prediction results; instead they are useful to point to modules which are more error prone comparing with other modules. They also cannot provide guidance to the developers how to develop better software.

5.4.4.3 Discussion on Metrics
The product metrics are a necessary element of the dependency evaluation and can be used as input for causal models and preparation of risk analysis. The following notes can be made on the subject of using product metrics as predictors of software reliability:

1. There is not a “best” or “standard” set of metrics. The definition of meaningful attributes for the software product and their standard measures is an issue still to be solved.

2. [Musa et al. 1990] noted that most of the complexity metrics have a high correlation with program size - Lines of Code (LOC). They offer little improvement over the Size in predicting the number of inherent faults at the start of testing. Nowadays it still cannot be ascertained that more complex metrics (or sets of metrics) are able to produce a better prediction compared to simple metrics, which are easier to collect, compare and apply for risk analysis. The hypotheses that complexity metrics are better predictors of fault prone modules than simple size metrics was weakly rejected in [Fenton et al. 2000b]. LOC and complexity metrics are reasonable-to-good predictors of absolute number of faults and failure prone modules, but poor predictors of fault density. Complexity metrics may be expected to be good predictors of comprehensibility or maintainability [Fenton et al. 2000b].
3. The main weakness of using the product metrics is that the prediction is based on the assumption that there is a correlation between the product metrics and the software faults within the product, which is not always true.

4. The relationship between faults and failures is not always straightforward.

5. Another major problem is the quality of the metrics and the need for company specific metrics. Given the many variances in the software development process mentioned before, like human and organisational differences, different tools and technologies, underlines the importance of locally collected metrics. Collecting fault data at different testing phases enables the transition to statistical process control for software development. It explains the observation that success in understanding statistically the software process occurs predominantly in large companies with highly repetitive process [Littlewood et al. 2000b].

6. Finally, metrics need to be presented in a form useful for making decisions (reports or recommendations) or for input to a tool.
6. Conclusions

The software dependability can be assessed by various means; however, in general, two complementing assessments can be delineated: process assessment and product assessment. Evidence for dependability can come from different artefacts. Metrics definition and collection, reliability modelling, fault removal and fault tolerance techniques, architecture definition, quality assurance, formal verification etc. are all relevant to software dependability. Some trends and challenges are described in [Littlewood 2005] and [Littlewood et al. 2000b]. This report looks into the various aspects affecting the dependability of a software system which are summed up below.

1. The only means of direct evaluation is through operational testing of the software. The operational tests used to provide input to the reliability models need to be representative. Reliability growth models using failure data and measures from the testing activity like test coverage can be an input for consideration.

2. Other information is available for assessment in addition to the testing. The quality of the software engineering process applied during the Software Development Life Cycle will reflect on the product. Standards for software for safety-critical systems relate recommended practices to the level of required reliability and this can be used to claim that the desired reliability has been achieved. With the rise of software complexity the software standards are increasingly looked at as a tool to help evaluate, predict and improve software quality, and hence dependability.

3. The application of requirement management tools and disciplined elicitation of the dependability requirements can contribute to prevent difficulties with the requirements which are the main contributor to defects found at later stages of the development life-cycle such as testing and post-release.

4. The application of rigorous development through all stages of the SDLC is a trend that will only expand. One avenue of work which enables rigorous development, early validation and automated analysis is Model Driven software development. The representation of the architecture in a software architecture description language provides a basis for a dependence analysis. The component-based software development stimulates the application of approaches (like design patterns) to incorporate the dependability requirements into the design.

5. Architecture-based software reliability models can be used throughout all phases of the SDLC starting from the architecture design and ending with the operational phase. Applied during the architecture definition and design states, the models can help with exploration of early design decisions. Initially such models can be based on simplifications. The existing models of software reliability are not as mature as the hardware models where it is possible to derive a number for the reliability of the system based on the reliability of the components.
6. Various review based techniques need to be used for analysis and evaluation during architecture design and further phases of the SDLC. They include QUASAR, HAZOP for safety analysis, Fault Tree Analysis or Failure and Effect Analysis.

7. Evidence from static analysis of the software. Although these methods do not provide an exact figure for the reliability of software, the application of manual techniques (such as reviews and inspections) as well as automated static analysis increase the confidence in the quality of the produced artefacts. These results need to be interpreted by an expert in order to be used for making decisions.

8. An assessment of a specific design for dependability would require looking into the development techniques which are used to facilitate validation, failure prevention, fault tolerance (error detection and recovery schemes), fault removal and fault forecasting. Development for dependability requires notations, methods and tools that reduce the probability of making errors and detect any errors as soon as possible after they are made [Thomas 2005]. Def(Aust) 5679 "The Procurement Of Computer-based Safety Critical Systems" views testing as complementary and unable to replace formal methods.

9. The consistent collection of metrics creates a base for process improvement, for prediction of dependability attributes and provides greater control over the projects.

The information above can be graphically represented in Table 8.

None of the evidence mentioned above can alone provide enough information about the dependability characteristics of software. As pointed out in many publications, probably the most important thing for understanding the software reliability measurement is experience and judgement. Our conclusion is that the only way to build a holistic picture of the state of dependability is to develop a comprehensive framework, where all the quantitative and qualitative evidence described in this report (people, process and product) is captured. Such a framework needs to provide input, guidance and/or suggestions to the expert to make conclusions. The selected approach should be able to capture uncertainty, expert judgement and incomplete information. Such a framework can be used for assessment of dependability and consequently to help Technical Risk Assessment (TRA).
<table>
<thead>
<tr>
<th>SDLC Phase</th>
<th>Input</th>
<th>Method</th>
<th>Tool</th>
<th>Output/Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Requirement Specification</td>
<td>Review/Analysis</td>
<td>UMD</td>
<td>Identify missing dependability requirements</td>
</tr>
<tr>
<td>Definition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirement</td>
<td>Requirement Specification</td>
<td>Review</td>
<td>Metrics</td>
<td>Number of requirement changes, number of contradicting requirements, risk</td>
</tr>
<tr>
<td>Specification</td>
<td></td>
<td></td>
<td>capture, DOORS</td>
<td>factors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or any</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>requirements</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>management</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>tool</td>
<td></td>
</tr>
<tr>
<td>Architecture</td>
<td>Architecture views</td>
<td>Architecture-based</td>
<td>ADL (AADL, CSR</td>
<td>Information for maintenance</td>
</tr>
<tr>
<td>definition</td>
<td></td>
<td>dependency analysis</td>
<td>etc), UML</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>State-based,</td>
<td>Reliability prediction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>path-based and</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>additive</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>models</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Simulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>models</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Requirement specification</td>
<td>Formal methods</td>
<td>Spin, HiVe etc</td>
<td>Verification of requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Requirements specification</td>
<td>Design patterns</td>
<td></td>
<td>Design document capturing NFR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Architecture views</td>
<td>Design for</td>
<td>Review</td>
<td>Expert opinion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fault-tolerance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td>Source code</td>
<td>Inspection</td>
<td>Fagan</td>
<td>Nr of errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>inspection</td>
<td>Error classification</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Automated static</td>
<td>Coverity, etc.</td>
<td>Nr of errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>analysis</td>
<td></td>
<td>Error classification</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Version history</td>
<td>ROSE</td>
<td>Nr of errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mining</td>
<td></td>
<td>Error classification</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Metrics collection</td>
<td>Scripts</td>
<td>KLOC, complexity measures</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test</td>
<td>Unit tests</td>
<td>Test coverage</td>
<td>scripts</td>
<td>TER</td>
</tr>
<tr>
<td></td>
<td>System test</td>
<td>RGM, architecture</td>
<td>DDTS, GNATS etc.</td>
<td>Remaining Defects, TER, Reliability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reliability Models,</td>
<td></td>
<td>prediction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test coverage</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operational use</td>
<td>Reliability Growth</td>
<td>DDTS, GNATS</td>
<td>Defects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Models</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defects</td>
<td>Reliability Growth</td>
<td>SMERF3</td>
<td>MTBF, MTTR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Models</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7. Future Work

Various approaches are used to combine disparate evidence in order to make evaluation of the overall dependability of a system. The DATUM (Dependability Assessment of Safety Critical Systems Through the Unification of Measurable Evidence) project in UK (the Centre for Software Reliability, City University, London) investigated several formalisms used to model uncertainty [Falla 1998]. The methods considered were Bayesian probability, Dempster-Shafer theory of belief functions, fuzzy sets and possibility theory. Although no single formalism for uncertainty was found perfect, Bayesian probability was chosen as the most mature and well developed formalism at the time. The obstacle for using Bayesian probability in cases of multiple evidence is the complex computations. This problem has been mostly overcome with the development of algorithms, solutions and tools in support of this formalism – the Bayesian Networks (BN). The developments in network propagation algorithms make Bayesian inference computationally feasible for solving complex problems. Bayesian inference can also be used for “what if” analysis. Brief reviews and details for most of the available Bayesian Modelling tools can be found in [Anthony 2006] and [Murphy 2005]. It is interesting to note in this context the paper [Simon et al. 2006], where BN implementation of the Dempster-Shafer theory is used to model the reliability uncertainty.

BN were also used by the FASGEP (Fault Analysis of the Software Generation Process) project to determine the fault propensity of software processes [Falla 1998]. The direction taken by the DATUM project has being followed in a series of projects within the RADAR (Risk Assessment and Decision Analysis) Group in Queen Mary University of London. [Wang et al. 2006] reported on the application of BN for project level estimation, where the accent is on the development and the test phases of the SDLC. [Perez-Minana et al. 2006] reported on the development of BN models for prediction of fault insertion and fault removal. The models were used as part of the software development process in Motorola, Toulouse. It is worth noting that the accuracy of the initial predictions of the generic models required a calibration based on measures from concrete projects. A procedure is suggested to calibrate the models and arrive at an improved BN. One approach was to vary the values associated with each node that are used as inputs to the intermediate nodes. The other approach (which produced better results) used linear regression and PCA to build the intermediate and the output nodes.

BN models have been used for scenario-based analysis and assessment of NFR (including reliability) [Sutcliffe et al. 2002] and [Gregoriades et al. 2005]. The models are plugged into the System Requirements Analyser tool. Scenarios are designed and depending on the selected tasks, the technology attributes, the human attributes and the environment variables, the BN model provides evaluation of the reliability for the scenario.

Based on the conclusions above, the following steps can be undertaken to provide practical input to the projects which are undertaken by the AOD AMS branch.

- Build on the existing architecture models to create estimates for the reliability of specific components and the whole software system.
• Build a BN model which considers evidence generated up to (and including) the architecture definition phase (from people, process and product):
  o identify relevant factors,
  o identify available tools which can help with:
    ▪ capturing
    ▪ processing
    ▪ presenting the collected data – metrics and evaluation
  o identify use cases for the model
References


[ARP 4754] Society of Automotive Engineers. Aerospace Recommended Practice 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems, November 1996


[DO-278] DO-278/ED109, Guidelines for the communication, navigation surveillance, and air traffic management (CNS/ATM) systems software integrity assurance, March 2002, RTCA & EUROCAE


http://www.comp.lancs.ac.uk/computing/resources/scs/


[Firesmith 2006] D.Firesmith, QUality Assessment of System ARchitectures (QUASAR), SEI CMU presentation, ver. 0.1, 2006
http://www.sei.cmu.edu/programs/acquisition-support/presentations/quasar.pdf


56


[Robinson et al. 2000] N.Robinson, P.Lindsay and A.Pitman, Extending the Integrated Capability Maturity Model (CMMI) for Safety-related Applications, TR 00-40, Software Verification Research Centre (SRVC), The University of Queensland, December 2000


[Spin 2007] Spin web page – C based static analysis tools, last update 17 April 2007


Dependability is an important quality attribute of software. As systems move towards complex system of systems and Ultra Large Scale systems, demand for methodologies for evaluating dependability will grow. The problem we aim to solve is how to evaluate the dependability of software early in the development cycle and before its deployment in order to assist Technical Risk Assessment, and to provide an additional assurance when monitoring the project health. This report surveys the existing approaches to software dependability.