Lappeenranta University of Technology
The Faculty of Technology Management
Department of Information Technology

ELIMINATING SOFTWARE FAILURES - A LITERATURE SURVEY

Licentiate Thesis

Supervisors: Professor Kari Smolander and D. Sc. Jouni Lampinen
Examiners: Professor Kari Smolander and D. Sc. Casper Lassenius

In Dublin, Ireland, 16 Mar, 2009

Kukka Rämö

ABSTRACT

Lappeenranta University of Technology
The Faculty of Technology Management
Information Technology

Kukka Rämö

Eliminating Software Failures – A Literature Survey

Licentiate Thesis, 2009

179 pages, 4 figures, 28 tables

Supervisors: Professor Kari Smolander and D.Sc. Jouni Lampinen
Examiners: Professor Kari Smolander and D.Sc. Casper Lassenius

Keywords:
Software error, software fault, software bug, software defect, software failure, fault prevention, fault elimination

Software faults are expensive and cause serious damage, particularly if discovered late or not at all. Some software faults tend to be hidden. One goal of the thesis is to figure out the *status quo* in the field of software fault elimination since there are no recent surveys of the whole area. Basis for a structural framework is proposed for this unstructured field, paying attention to compatibility and how to find studies. Bug elimination means are surveyed, including bug knowhow, defect prevention and prediction, analysis, testing, and fault tolerance. The most common research issues for each area are identified and discussed, along with issues that do not get enough attention. Recommendations are presented for software developers, researchers, and teachers. Only the main lines of research are figured out. The main emphasis is on technical aspects.

The survey was done by performing searches in IEEE, ACM, Elsevier, and Inspect databases. In addition, a systematic search was done for a few well-known related journals from recent time intervals. Some other journals, some conference proceedings and a few books, reports, and Internet articles have been investigated, too.

The following problems were found and solutions for them discussed. Quality assurance is testing only is a common misunderstanding, and many checks are done and some methods applied only in the late testing phase. Many types of static review are almost forgotten even though they reveal faults that are hard to be detected by other means. Other forgotten areas are knowledge of bugs, knowing continuously repeated bugs, and lightweight means to increase reliability. Compatibility between studies is not always good, which also makes documents harder to understand. Some means, methods, and problems are considered method- or domain-specific when they are not. The field lacks cross-field research.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Kukka Rämö

Ohjelmointivirheiden välttäminen – Kirjallisuuskatsaus

Lisensiaatintutkimus, 2009

179 sivua, 4 kuvaa, 28 taulukkoa

Ohjaajat: Professori Kari Smolander ja KTT Jouni Lampinen
Tarkastajat: Professori Kari Smolander ja TkT Casper Lassenius

Hakusanat:
Ohjelmointivirhe, ohjelmistovirhe, vika, häiriö, virheenetsintä

Ohjelmointivirheet ovat kalliita ja aiheuttavat vakavia vahinkoja, varsinkin jos ne havaitaan myöhäisessä kehitysvaiheessa tai käytön aikana tai niitä ei havaita ollenkaan. Jotkut virhetyypit ovat usein piileviä. Työn yhtenä tavoitteena on luoda aihealueeseen liittyvä yleiskuva, koska alalta ei ole viime vuosina tehty kokonaisvaltaista kirjallisuuskatsausta. Työssä luodaan perustaa alan jäsentämiselle; yhteensopivuuteen ja tutkimusten löytämiseen kiinnitetään huomiota. Työssä tehdään kirjallisuuskatsausta seuraavilta osa-alueilta: ohjelmointivirhetuntemus, virheiden ennaltaehkäisy ja ennustaminen, tarkastaminen ja analyysi, testaus ja virhetilanteista selviytyvien ohjelmien laatiminen. Jokaiselta osa-alueelta kartoitetaan yleisimmät tutkimuskohteet, ja näistä tutkimuskohteista keskustellaan. Lisäksi työssä keskustellaan kohteista, joita ei ole tutkittu riittävästi. Lopuksi esitetään suosituksia ohjelmistokehittäjille, tutkijoille ja opettajille. Työssä hahmotellaan ainoastaan tutkimuksen päälinjat ja pääpaino on teknisillä näkökohdilla.

Kirjallisuuskatsaus tehtiin suorittamalla hakuja IEEE-, ACM-, Elsevier- ja Inspect-tietokannoista. Lisäksi selattiin eräiden tunnettujen alan lehtien tiettyinä aikaväleinä ilmestyneet numerot, lähinnä viime vuosilta. Työtä varten tutkittiin myös joitakin muita lehtiä, konferenssijulkaisuja sekä muutamia kirjoja, raportteja ja Internet-julkaisuja.

Työssä havaittiin muun muassa seuraavia ongelmia ja keskusteltiin niiden ratkausukeinoista. Monet tarkastukset tehdään ja monia menetelmiä sovelletaan vasta testausvaiheessa, koska testauksen luullaan olevan ainoa laadunvalvontatapa. Monet staattiset tarkastustavat on lähes unohdettu, vaikka niiden avulla löydetään virheitä, joita on vaikea havaita muilla keinoilla. Muita unohtuneita alueita ovat ohjelmointivirhetuntemus, tietämys jatkuvasti toistettavista virheistä sekä helpot luotettavuuden lisäämiskeinot. Tutkimukset ovat usein yhteensopimattomia ja siten myös vaikeita ymmärtää. Joidenkin ongelmien, keinojen ja menetelmien ajatellaan liittyvän ainoastaan tiettyyn menetelmään tai sovellusalueeseen, vaikka ne ovat yleisempiä. Yhden osa-alueen tutkimuksissa ei yleensä oteta huomioon alan muita osa-alueita.

ACKNOWLEDGEMENT

This survey has been done as part of the intermediate (licentiate) degree between MS and PhD for Lappeenranta University of Technology. I thank my supervisos, D.Sc. Jouni Lampinen and professor Kari Smolander, for help, suggestions, and encouragement. I also thank Ph.D. Matti Heiliö and professor Heikki Kälviäinen for advice and encouragement.

In Dublin, Ireland, 16 Mar, 2009.

Kukka Rämö

TABLE OF CONTENTS

FIGURES

TABLES

1 INTRODUCTION

1.1 Why Are Bugs so Bad?

Software faults cause plenty of economic loss, particularly if they are not detected in early stages of software development. According to Research Triangle Institute, RTI (2002), the annual costs of software bugs are about 0.2 to 0.6 per cent of the GNP in the USA. The estimate is assumed to be too low. The study focused on testing and use of software as means to detect bugs.

The earlier the software fault is detected, the lower the costs are, see e.g. Boehm (1981), RTI (2002), Westland (2002), and Leszak *et al.* (2002) for details. Repair costs of a software fault are about 100-200 times higher in the maintenance phase than in the requirement specification phase (Boehm 1981). Faults are more difficult to correct in later phases of software life cycle than in earlier phases. For example, Naval Research Laboratory found in one project that 21 of the 22 defects that were moderately hard or hard to correct were discovered during the final 10% of the development life cycle (Fredericks & Basili 1998).

Some software faults lead to catastrophic consequences if detected only during the maintenance phase; there have been accidents caused by software faults, see e.g. Leveson (1995) and Ladkin (1994). In addition to this, some software faults are never detected, and those hidden faults may cause damage all the time. For example, people may make decisions based on the output of erroneous software and nobody ever detects that better decisions could have been made.

1.2 The Goals of the Thesis

Figuring out the *status quo* in the field, the current situation as a whole. Some areas of fault elimination get attention, and research is being done about them. The research is surveyed in the thesis. In addition, general features of research in the field are being figured out.

The field lacks up-to-date general surveys. Fault elimination is a wide topic, and partial surveys about some subareas have been made. In the 1970's general surveys were made, but the field was narrow at that time. As far as the author knows, a more general survey has not been done yet. The goal is to make a textbook about fault elimination, partly based on the material of this thesis. Such books probably do not exist.

Proposing some basis for structural framework for the field. Information in the field is hard to find. For example, in the field of compiler development there is plenty of information that could be used in bug elimination, too, but those who need information for bug elimination barely search it from publications that are intended for planning compilers. Also, the concepts and terms related to software faults are used inconsistently.

Surveying and increasing bug knowhow. In this work, information is collected about characteristics of bugs, fault classifications, fault proness, fault types, their temporal development, correlation between faults, and root causes for faults. Knowing about faults helps eliminating them. One often repeats the same faults all the time because he/she does not know about them. Knowledge about bugs can be used by teachers and researchers, too.

Surveying research about fault prevention, fault prediction, fault detection, and fault tolerance. The most common research issues for each area of fault elimination are discussed in this thesis. By studying what has been done, one can figure out what each area contains. Reviews of those areas help developers eliminate faults and teachers plan courses about fault elimination.

Figuring out what should be studied more. Some areas of fault elimination do not get as much attention as they should. One goal of this thesis is to reveal issues that would require more attention. Some fault types cannot be eliminated well with current means; identifying omitted research helps improving the situation. In addition, presentation about research trends and forgotten areas help research people choose their topics and teachers to plan their courses.

Encouraging for early fault elimination. According to several studies, software developers should eliminate faults in as early a phase as possible. Fault elimination is usually done later that it could be done. Late elimination is less efficient and more expensive than earlier elimination. This thesis presents some means for early fault elimination.

Presenting concrete recommendations. Recommendations are presented for software developers to eliminate faults; for research scientists to plan their research, and to improve usability, comparability, and understandability of results; and for teachers to choose course material.

1.3 Scope and Outlines of the Thesis

Because the topic is wide, only main lines of the existing research can be figured out here. Plenty of research has been done about organizational, managerial, and economical means for software fault elimination. However, the emphasis in this work is in technical means and in what can be done by means of software development.

Quality assurance is a wide topic. It covers, for example, efficiency and maintainability. Articles that do not involve fault elimination have not been investigated in this survey.

1.4 Surveyed Material

The survey was made by performing different fault-related searches in IEEE, ACM, Elsevier, and Inspect databases. In addition, a systematic search was done for material presented in table 1. Most references in the thesis have been journals, but there have been some conference proceedings, and a few books, technical reports, and web articles. Some other sources have been used, too. (Peng & Wallace 1993) is a web publication that has an overview about error analysis. Some information of the publication has been included in different parts of this thesis. No material published after February 2009 has been investigated in the thesis.

Table 1. Material for systematic searches in this thesis

Publication	**Volumes (last issue)**	**Times**
ACM Computer Surveys	1-41(1)	1969 – Dec 2008.
ACM Transactions on Computer Systems	1-27(1)	1983 – Feb 2009.
ACM Transactions on Software Engineering and Methodology	1-18(2)	1992 - Nov 2008.
Formal Aspects in Computing	10-17(2)	1998 - Aug 2005.
Formal Methods in System Design	10 – 27(1-2)	Feb 1997 - Sep 2005.
Information and Software Technology	37 – 51(2)	1995 - Feb 2009.
Journal of Systems and Software	28-82(2)	1995 - Feb 2009.
Science of Computer Programming	24 – 64(3)	1995 - 1. Feb 2007.
Reliability Engineering and System Safety	83-91(1)	2004 - Jan 2006.
IEEE Transactions on Software Engineering	1-35(1)	1975 – Feb 2009.
IEEE Transactions on Reliability	41-48(4)	1992 - 1999.

1.5 Structure of the Thesis

The following classification for fault elimination means is presented in (Avižienis *et al.* 2004) and followed in this thesis: fault prevention, fault removal, fault tolerance, and fault forecasting. Faults need to be detected before removal. Fault removal means are usually dependent on the application environment, so only the fault detecting portion of fault removal is investigated in the thesis. Bug fixes are investigated as a factor that causes new bugs. Fault forecasting is called fault prediction in this thesis. In this thesis, the concept of fault elimination covers fault tolerance, too.

Figure 1 describes the structure of the rest of this thesis in relation to fault prevention, fault detection, and fault tolerance. Chapter 2 is related to fault prevention, fault detection, and fault tolerance. In chapter 2, bug knowhow is investigated. Software bug types, bug classifications, and temporal development of bug types are inspected; the main goal is to make developers avoid known bugs. Features of bugs and fault prone software, correlation of faults, and root causes for bugs are also studied. Development framework and bug knowhow can be developed further with help of each other, and both are used in checking, proving, and testing software.

Chapter 3 investigates software development processes, theories, risk analysis, metrics, and defect prediction. Those means are mainly associated with defect prevention and prediction, but they have connections with fault detection and fault tolerance. Chapter 4 is about those means to look for faults that are not based on testing. It involves checks and analysis methods that can be performed for software in order to prevent and detect bugs. The goal of those checks is to make sure that everything is covered correctly in software. Rigorous proving is discussed, too. Many of those checks can be performed before testing. Faults can be both prevented and detected with analytic checks if the checks are done before testing. It is recommended that checks be done as early as practical. Chapter 5 processes testing in order to detect software bugs. Chapter 6 is about fault tolerance. Even if there is software development framework and bug knowledge and software is thoroughly analyzed and tested, there can be bugs. The chapter contains means to prevent harm if faults exist. Means of fault tolerance often involve fault detection. For example, recovery can often be done after the detection of a fault, and faults can be detected by self-checks. Chapter 7 contains summary, conclusions, and recommendations, and Chapter 8 is a closure; those two chapters have been omitted from figure 1.

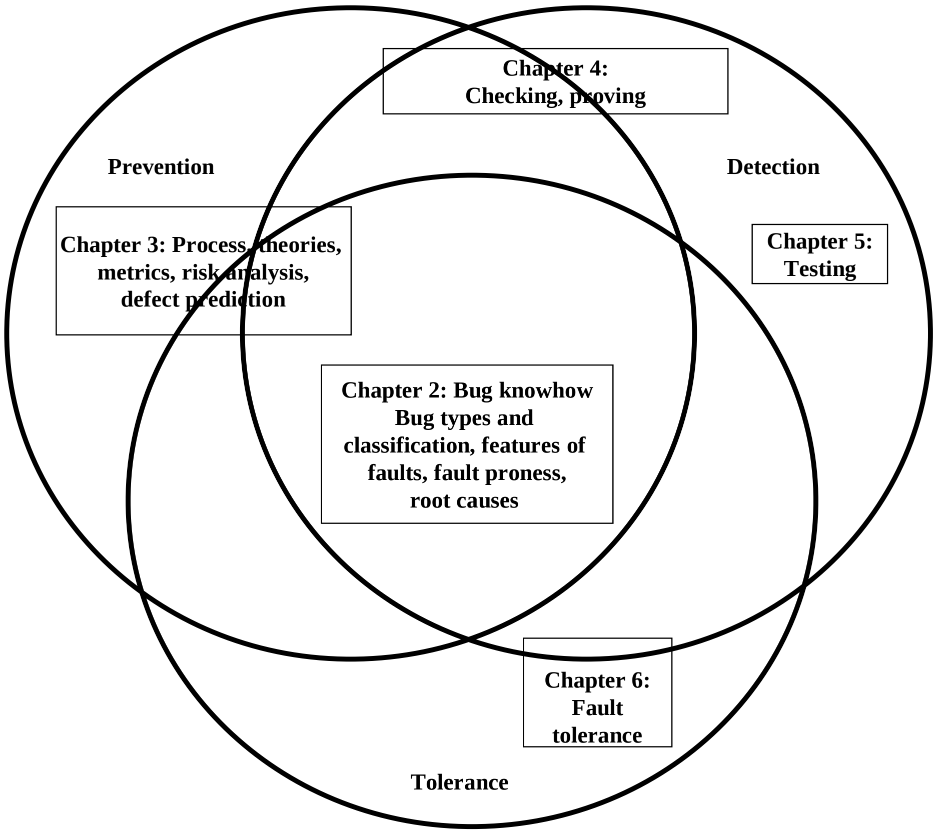
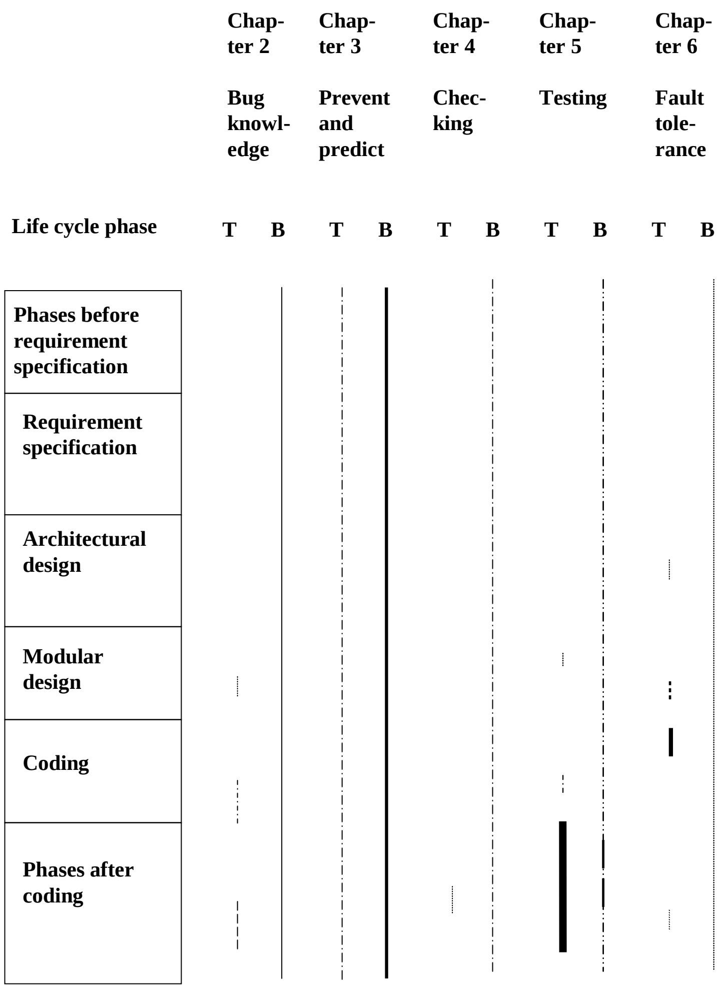

Chapter 4:
Checking, proving
Prevention
Detection
Chapter 3: Process, theories, metrics, risk analysis, defect prediction
Chapter 5: Testing
Chapter 2: Bug knowhow Bug types and classification, features of faults, fault proness, root causes
Chapter 6: Fault tolerance
Tolerance

Figure 1. Means for eliminating bugs related to the structure of this thesis

Figure 2 answers the question about in which phases of the software development life cycle the topics described in chapters 2-6 of this thesis are applied. The leftmost column describes the life cycle phases. Under each chapter column, the left pillar describes the phases where the topics are typically applied, and the right pillar describes the phases where they should be applied. The thickness of the pillar describes how much the topic is being applied.

The topics in the software development framework are applied in all phases, although they should be applied to a greater extent. Typically, a little checking is performed after coding, if at all, although checking should usually be performed most of the time during all phases of the software development cycle. Testing is often considered a long-lasting stage after coding, although testing should be performed during all phases most of the time, in addition to the main testing phases after coding. Prevention methods are used all the time, but they could be used more. For example, process maturity models and statistic process control could be applied more. During all phases of life cycle, prediction methods like risk analysis and metrics could be used more often than they are being used. Defect prediction models could be applied more often, too. Bug knowledge and fault tolerance are applied randomly if at all. They should be applied all the time when software is being developed.



Chap-ter 2 Bug knowl-edge
Chap-ter 3 Prevent and predict
Chap-ter 4 Chec-king
Chap-ter 5 Testing
Chap-ter 6 Fault tole-rance
Life cycle phase
T B T B T B T B T B
Phases before requirement specification
Requirement specification
Architectural design
Modular design
Coding
Phases after coding

Legend:

T typical:	**In what phases of the software life cycle the topic of the chapter is typically applied**
B better:	**In what phases of the software life cycle the topic of the chapter should be applied**

Figure 2. Chapters of this thesis related to the software life cycle

1.6 Basic Definitions

Some definitions are explained below that are frequently used in software engineering.

Error is a discrepancy between the computed, observed, or measured value or condition, and the true specified or theoretically correct value or condition (IEEE 1990). In this work, specification errors are included. Definitions of fault, failure, and mistake are commonly used as a definition to error, but fault tolerance discipline distinguishes between all those definitions (IEEE 1990). In fault tolerance analysis, error is the amount by which the result is incorrect (IEEE 1990).

Fault is an incorrect step, process, or data definition (IEEE 1990). See e.g. Abbott (1990) about problems in defining the notion of fault.

Failure is an inability of a system or a component to perform its required function within specified performance requirements (IEEE 1990).

Defect may mean error, fault, or failure.

Mistake is a human action that produces an incorrect result (IEEE 1990).

Safety critical software is software whose failures may have very serious consequences. There is unanimity about that software is safety critical if its failure can cause deaths and serious health losses. Other health issues and evident physical discomfort, too, are often included in the definition of safety critical software. Sometimes software whose failures cause significant damage to property is regarded as safety critical; according to IEEE standard (IEEE 1990), critical software is software whose failure could have an impact on safety, or cause large financial or social loss.

2 AVOIDING KNOWN BUGS

Knowing which bugs are common helps stop repeating them. This chapter presents some common bug types. The chapter also involves characteristics of bugs, causes for bugs, features of fault-prone software, and correlation of bugs. This information helps in eliminating bugs and preventing the repetition of the same bugs. It also helps in improving the software developing process and bug-related metrics, which are analyzed in chapter 3. Methods for risk analysis and defect prediction can also be developed accordingly; those methods are investigated in chapter 3. Chapters 4-5 of this thesis present different means to detect software faults. The information in this chapter can be used in applying those means, as well as in using fault tolerance means presented in chapter 6. General information of different knowledge areas like mathematics, computer science, and computer engineering (SWEBOK 2007), can be used in eliminating bugs. The information in this chapter is specifically involving bugs and could be regarded as a branch of computer engineering.

In the first subchapter, common fault types and fault classifications are introduced. Also, the temporal development of bugs is studied in the subchapter. In subchapter two, bugs are presented that are typical to specific application domains and environments. In subchapter three, features of fault prone software and characteristics of hidden bugs are presented. In addition, reasons why bugs are hidden and correlation between bugs are investigated. In subchapter four, causes for faults are discussed. Subchapter five is a summary of bug knowledge surveyed in this thesis.

2.1 Fault Types

This subchapter investigates fault types. In the first part, different fault types are presented. In the second part, fault classifications are discussed. In the last part, the question about which bug types have been present during different decades is investigated.

2.1.1 Typical Faults Found in Software

There are numerous lists of bug types, and they originate from different decades. For example, Foster (1980) has classified code faults, and Lutz and Woodhouse (1996) have classified specification faults. Table 2 presents some usual bug types; the same bug may belong to several of those bug types.

Table 2. Examples of general and special bug types

General bug types
Constant/value/sign: Erroneous value for constant or variable, or wrong sign (Foster 1980).
Wrong padding: error in padding of a field (FS Networks 2005).
Unit: using wrong units (Peng & Wallace 1993).
Expression: fault in boolean, arithmetic, or relational expression (Foster 1980).
Associative shift: erroneous association in boolean expression (Kuhn 1999).
Operation: operator (pointers included), operand, or even special character (Foster 1980), (Podgurski & Clarke 1990).
Negation: inverse of e.g. operator or number (Foster 1980), (Lau & Yu 2005).

Continued on next page

Reference: error in reference to a variable or an operator (Foster 1980).
Delimiter: wrong, extra, or missing delimiter, see e.g. (Duck 2005).
Sequence/calculation: wrong order of calculations in an expression (parenthesis in a wrong place, precision[1] lost due to wrong order of computations), or a wrong sequence (Foster 1980), (Howden 1980), (Darcy 2006).
Data errors: e.g. absent data, incorrect data, timing error, and undesired duplicate of data (Lutz & Woodhouse 1996).
Structural: structural faults like missing paths (Howden 1976), or cycles where there should not be cycles (Holmberg & Eriksson 2006).
Logical: neglecting branches, forgetting cases or steps (Schneidewind & Hoffmann 1979), (Dupuy & Leveson 2000).
Precision: see subchapter 4.1.2, and e.g. (Darcy 2006).
Error accumulation: accumulation of error in repeated computations due to e.g. precision limits (Goldberg 1991).
Convergence: e.g. assuming wrong convergence point. Some convergence and coherence problems are introduced in (Bastani *et al.* 1988).
Conversion: converting elements from one type or format to another (Dupuy & Leveson 2000).
Type mismatch: e.g. performing inappropriate type conversions or copying incompatible objects may cause mismatch (Spohrer & Soloway 1986a), (Sullivan & Chillarege 1991).
Overflow/underflow: A number is too large or small for the space reserved for it (Peng & Wallace 1993).
Value out of range: e.g. value of a variable, a parameter, or an argument is too large or too small, or is not inside the range of the function; or the divisor is zero (Peng & Wallace 1993).
Off by: e.g. incorrect processing of an extra element, or performing a loop one extra time or one time too few, e.g. (Smidts *et al.* 2002).
State faults: missing state, extra state, missing transfer, extra transfer, and erroneous transfer (Laycock 1993).
Missing statement: missing program statement (Schneidewind & Hoffmann 1979).
Statement order: wrong order of statements (van der Meulen *et al.* 2004).
Initialization or reset: not giving values to data elements before use; using old values when new ones should have been given; or wrong, incomplete, or undesired initialization; e.g. (van der Meulen *et al.* 2004), (Glass 1981), (Endres 1975).
Duplicate name: the same name is unintentionally used for two different objects that can be mixed with each other (Fruth *et al.* 1996).
Side effects: There is a lot of research about side effects. Examples of ways to cause side effects are evaluating a macro several times without considering side effects when there may be side effects (Seacord 2007), or altering a global variable in one function in a way that is unexpectable in another part of program (Fitzpatrick 2006).
Consistency: e.g. inconsistency in use of global variables, or between software states (Peng & Wallace 1993).
Interface: faults in interaction with other system components (Lutz 1993).
Encapsulation: faults related to hiding elements from other parts of the system (Sinha & Harrold 2000).
Exception bugs: bugs related to exceptions, e.g. (Yi 1998).

Continued on next page

[1] See subchapter 4.1.2 about precision.

Memory: e.g. allocation (Chou *et al.* 2001).
Violation of standards: e.g. violation of the standard (e.g. DO-178) that should have been applied for the software, or violation of an internal standard in a company (Peng & Wallace 1993).
Application dependent bug types
Timing Software is becoming more time dependent, which increases the number of timing bugs. One instance of timing bugs is occurrence of simultaneous events when there should be only one event. Leveson and Stolzy (1987) discuss the following time faults: a desired event does not happen, an undesired event happens, there is wrong order for events, two incompatible events occur simultaneously, or there is erroneous timing or duration for events. Atlee and Gannon (1993) discuss simultaneous occurrence of events that should not occur simultaneously. Lutz and Woodhouse (1996) also mention timing and order faults, like too early or too late arrival of data, repeated occurrence of an event that should occur only once, or intermittent occurrence of iterative events that should occur regularly.
Object oriented programming There are type errors e.g. due to inheritance. Rine (1996) studies preventing unsafe sharing of objects by stronger type structures. Peleg and Dori (2000) observe model-related faults like faults related to aggregation, links, or assignment; cyclic model not modeled as cyclic; or different types of confusion, e.g. between an event and a condition, or between an action and an activity.
Neglecting differences Differences between different computers, operating systems, and e.g. compilers often causes software malfunction. The following list contains common differences between machines: • Alignment and padding (Hakuta & Ohminami 1997). • Whether fields are assigned left to right or right to left. Hakuta and Ohminami (1997) discuss if words are stored with the most significant byte in the top byte or in the end byte of the word. • Storing numbers in memory (Hakuta & Ohminami 1997). • Representation of data types, e.g. floating point numbers and negative numbers (Hakuta & Ohminami 1997), and whether sign extensions are used for characters (Iwata & Hanazawa 1993). In many implementations, float and double are different types, and extra digits may be arbitrary in many implementations if a number is converted from one type to one that has a greater number of significant digits (Darcy 2006). • Methods for performing truncation and rounding (Cannon *et al.* 1990); particularly for negative numbers, see e.g. (Seebach 2006). • Zeros in comparison (Cannon *et al.* 1990). • Order of computations and/or assignments can be left to right, right to left, or nondeterministic; for example, C compilers re-arrange commutative and associative operations arbitrarily (ARM 2003). • Ranges for data types (Hakuta & Ohminami 1997). • Limits like those of nesting levels and arguments in a function call (ARM 2003). • Hakuta and Ohminami (1997) discuss processor architecture differences and other portability factors, including other system environment factors.

Gerhart and Yelowitz (1976) have surveyed software errors in specifications, software, and proven software. Typical faults in specifications were incompleteness and the situations where something that had been intended to be unaltered had been altered in programs. There were also other kinds of errors related to misunderstanding. Termination-related errors were common in proven programs.

2.1.2 Fault Classification Schemes

There is no one single classification schema for software faults, regardless of the range of efforts to make it and long history of related research. Developers often aim at removing the subjectivity of the classifier and creating orthogonal components. The more choices there are for any defect, the more accurately the developer can choose among the types, but more choices make classification harder and more time consuming. (Kelly & Shepard 2001).

Different classifications have been developed for different *purposes*, e.g. for making decisions during software development, tracking defects for process improvement, guiding the selection of test cases, or analyzing research results (Kelly & Shepard 2001). One application of making decisions during software development could be to realize different bug types and knowingly avoid them in software design. IBM has had quality assurance goals related to bug tracking and classification. One is to characterize or understand attributes in development environment (Fredericks & Basili 1998). Another is to provide feedback (Kelly & Shepard 2001) (Fredericks & Basili 1998). Knowing where the defect is helps improving the process (Fredericks & Basili 1998). IBM has a knowledge base about common defects (Fredericks & Basili 1998).

Sometimes it is hard to put a bug *in one* specific *class*. There are different ways in which the bug can be classified (El Eman & Wieczorek 1998). Kelly and Shepard (2001) describe situations where multiple interpretations of defects and categories are possible. The classifications have probably not been defined perfectly (Kelly & Shepard 2001). According to El Eman and Wieczorek (1998), different people usually put a fault in the same class if they use the same classification schema.

Table 3 presents general and special classification criteria and examples of bug classifications.

Table 3. General and special bug classifications

General classification criteria
Bug type Fredericks and Basili (1998) present classifications of bug types.
Breadth Lau and Yu (2005) present expression faults based on erroneous part of the expression. For example, a literal, a term, or the whole expression may have been negated.
Qualifier Fredericks and Basili (1998), and Kelly and Shepard (2001) describe existing classifications with keywords like missing, extra, duplicate, incorrect, incomplete, unclear, ambiguous, changed, and better way.
Trigger Fredericks and Basili (1998) describe classifications about what triggers the failure caused by the fault.
Source Fredericks and Basili (1998) describe classifications about the part of the system or an environment that caused the bug to be born, i.e. the source of the misunderstanding.
Location Kelly and Shepard (2001) describe classifications about the location of the fault; some examples are structure, expression, assignment, input, and output.
Life cycle phase when born Fredericks and Basili (1998) describe classifications about the software development life cycle phase where the bug was born.
Legacy level when born Fredericks and Basili (1998), and Kelly and Shepard (2001) describe classifications about the legacy level where the bug has been born. Examples are new software, modified software, bug fix, and re-fix.

Continued on next page

Activity where observed Kelly and Shepard (2001) describe bugs based on the activity that was being performed when the defect was detected, e.g. review, inspection, or testing.
Action in a failure situation Kelly and Shepard (2001) describe classifications about what the system does in a failure situation due to the failure, i.e. consequences of the failure.
Amount of damage Fredericks and Basili (1998) discuss about attributes like damage level, number of affected states, number of affected modules, whether the failure region (set of inputs that cause the failure) is connected, and repair effort.
Consistency Whether the failure occurs always or only sometimes, and whether the software behaves in the same way every time the failure occurs (Avižienis *et al.* 2004).
Volatility Gray (1985) classifies faults as transient faults and permanent faults.
Dependencies Jeng (1999) classifies faults as path dependent/independent, and data dependent/independent faults.
Special criteria and classifications
Examples of classifications Fredericks and Basili (1998) survey five famous classifications. Chillarege *et al.* (1992) describe orthogonal classifications and an environment that enables the metering of cause-effect relationships. Avižienis *et al.* (2004) classify faults and failures.
Classifications in domain testing Howden (1976) classifies faults as domain faults and computation faults. Domain faults are either missing paths or path selection faults (*ibid.*). Path selection faults are either predicate faults or assignment faults. Harrold *et al.* (1997) extend the classification so that faults are classified as statement faults and structural faults that cover more than one statement.
Classifications in HAZOP (Hazard and Operativity Analysis) Reese and Leveson (1997) describe keywords used in HAZOP. Original keywords were: "none", "more", "less", "as well as", "part of", "reverse", and "other than". Software oriented extension contains component and system oriented keywords. There are hazard-related keywords for each of those. For example, signals may drift. There are also data flow- oriented keywords for output, its timing, and detectability of output faults. (Reese & Leveson 1997).
Hierarchical classifications Lau and Yu (2005) present a fault class hierarchy that relates literal, term, operator, and expression faults and insertion, omission, reference, and negation faults. Those components are not completely orthogonal; e.g. term operation fault is located between literal insertion fault and literal negation fault in the study. Nakajo and Kume (1991) have a three-level class hierarchy, see subchapter 2.4. Okun *et al.* (2004) compare classes of logical faults in specification based testing and calculate relationships between those classes.

There are numerous studies about looking for and classifying faults or failures *automatically*. Execution traces are analyzed (Podgurski *et al.* 1999). Multivariate analysis like clustering is sometimes used in forming classes automatically (Podgurski *et al.* 1999). Dependencies between elements are sometimes used (Francis *et al.* 2004), and invariant properties that cause faults or failures are sometimes searched for (Hangal & Lam 2002). Some studies involve hierarchical methods and stratified sampling (Podgurski *et al.* 1999). Francis *et al.* (2004) develop tree-based methods for refining failure classifications when software is being executed.

Knowing which bugs are common and which bugs are present together helps stop repeating them. Berglund (2005) studies *communicating* about bugs. There are studies about the content of bug reports. Marick (1997) discusses guidelines about what a good bug report must contain.

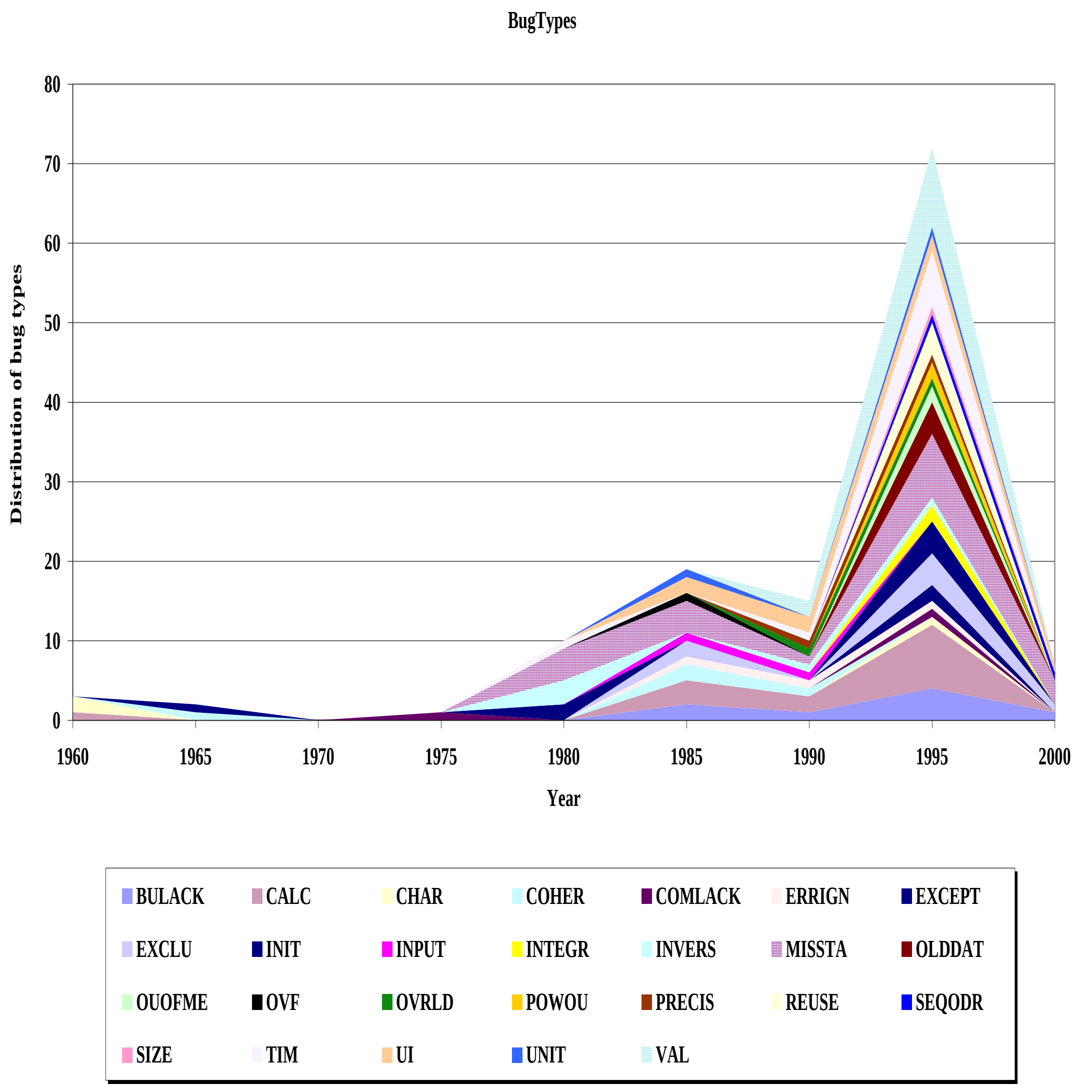
There are *databases* about common faults in order to increase bug knowledge; e.g. Fredericks and Basili (1998) discuss IBM database; Card (1998) discusses about a fault report database for cause and impact analysis, and for analyzing defect time and detection time; and Clapp *et al.* (1992) present a method where database contains data about test runs, test scripts, faults, repairs, and source code. Kim *et al.* (2006) discuss bug fix database for building bug patterns and thus stopping repetition of the same bugs in the project. There are public bug databases, e.g. Bugzilla database on https://bugzilla.mozilla.org/. National Institute of Standards and Technology in the USA has developed an EFF tool for collecting and maintaining bug databases (Wallace *et al.* 1997). Liu *et al.* (2008) study how to make it easier to find failures caused by the same bug.

2.1.3 Temporal Development of Fault Types

Figure 3 presents a distribution of bug types for safety critical accidents found on the Internet. Approximately four decades are covered. The figure contains aircraft, car, elevator, nuclear power plant, train, and spacecraft accidents. The bugs processed have been observed in safety critical software when it had already been in use; non-public bugs found in early phases of safety critical software development have not been included in the figure. There have usually been many more causes for the accidents; software bugs have been only one factor. In some cases, more bugs have been found later than just those that affected the accident. Besides known bugs, some cases like (NEAR 1999) contained malfunctions in simulation, the reasons of which were not found.

The accidents have been collected from the following references: (Adler 1998), (Arida 1999), (Brader 1987), (Dershowitz 2007), (Finkelstein & Dowell 1996), (Ganssle 1998), (Huckle 2005), (Jacky 1987), (JPL 2000), (Ladkin 1994), (Ladkin 1996), (Ladkin 1999), (Leveson 1995), (MCOMIB 1999), (Modugno *et al.* 1997), (NASA 1999), (NASA 2003), (NASA 2006), (NASA 2007), (NEAR 1999), (Neumann 1985), (Neumann 2007), (Nussbacher 1992), (Reid 1995), (Rushby 1993), (Santor 2007), (Sheffield 2001), (Sogame & Ladkin 1996), and (Strobl 2000).

The figure contains only the accidents and incidents where some details about the bug are available. On the Internet there are numerous accident reports without those details: either a software bug is suspected but not found, or a bug is known in more or less detail but the report includes no details about it.



Explanations of the legend: BULACK = lack of backup, CALC = calculation, CHAR = character fault, COHER = coherence, COMLACK = lack of communications, ERRIGN = continuing operation when something was wrong, EXCEPT = exception fault, EXCLU = excluding important states or features, INIT = initialization, INPUT = input, INTEGR = integration, INVERS = inversion, MISSTA = missing state, OLDDAT = too old data, OUOFME = out of memory, OVF = overflow, OVRLD = overload, POWOU = power out, PRECIS = precision, REUSE = reuse, SEQODR = wrong order in sequence, SIZE = size, TIM = time, UI = user interface, UNIT = unit, VAL = value.

Figure 3. Temporal development of bug types

Some observations can be made from the descriptions:

- There have been character faults at early times, and a few of them have been present later.
- Calculation-, coherence-, and initializing bugs have been common all the time. Calculation bugs may be computation errors or ignored conditions. For instance, the orbit around the sun was ignored in one piece of Gemini V -software (Neumann 1985).
- There have been some errors involving values or units.

- Inversion bugs have been present from the 1970's.
- From the 1980's when software became more complicated, there have been faults related to ordering of operations within an expression, ordering of sequences, exception processing, overload, timing, and missing states, as well as input faults and user interface faults.
- From the 1980's, sizing and overflow faults have started to appear.

Lack of integration between system factors combined with *insufficient user interface* has been a factor in many accidents. For example, in Nagoya 1994 accident (Sogame & Ladkin 1996) there was contradictive action between different aircraft parts, and in Cali 1995 accident (Ladkin 1996) there was a mismatch between a chart value and a database value. In both examples, the user interface, too, could have been better.

There are *contradictive bug reports* from the 1960's, when many bugs were character errors. For example, there are different versions about the reason for the disaster of Mariner 1 spacecraft in the 1960's. Character-, line-, or operation-related bugs are suspected, and even different descriptions about the same type of bug are contradictive. Jacky (1987), Brader (1987), and Strobl (2000) discuss the problem. It is not clear how many of the reported bugs appeared and if they were independent (Jacky 1987).

In some cases, it has been hard to decide what is being considered as a *missing state*. For example, X-31 crashed because ice caused wrong input, and the computer could not compensate it (NASA 2007). Mars Polar Lander software had value, calculation, and logic faults, and was ignoring transient states (JPL 2000). Its backup could stop working when the spacecraft was put into a safe mode (*ibid.*). When the system ignores special conditions, it is hard to know whether those conditions have been unintentionally ignored or if a trade-off had been made, see e.g. (Leveson 2001). For example, autopilots do not take all situations into account, which can be a factor in aviation accidents where pilots rely too much on autopilot or become distracted when using an autopilot; see e.g. (Sogame 1999) and (NTSB 1980), which are not included in the graph of figure 3. Sometimes there has not even been an opportunity for manually changing the action of the computer system, see e.g. (Ladkin 1994).

Part of a temporal development is *how long a bug lasts*. Zelkowitz and Rus (2004) studied defect evolution in a product line environment. Bugs were born all the way including mission preparation, and bugs were detected all the way including the mission. Some faults stayed for more than 10 years.

2.2 Faults in Specific Applications

In this subchapter, specific faults in different application domains and environments are being surveyed. The first part involves faults in different application domains. The second part investigates faults in different programming environments, and typical faults when using different programming languages.

2.2.1 Faults in Specific Application Domains

Sometimes research related to software faults concentrates on a specific kind of systems. For example, many studies are limited to database systems; real-time systems; or safety-critical, large, complex, or concurrent systems. Those studies can involve one or several phases of software life cycle. Many studies examine individual programs or many programs belonging to the same application area. Table 4 includes only some examples of the numerous studies that have been performed. Some of those studies contain a survey of research about bug types found in specific application domains or environments.

Table 4. Examples of studies about bug types in different application domains

Application	**Study**	**Common faults and remarks**
Development of several small programs by one programmer in Algol W for IMB 360/67, 173 errors	(Schneidewind & Hoffmann 1979)	Design faults like neglecting extreme conditions, forgetting cases or steps, and faults in loop control; representation faults (writing something else than desired); syntax faults; and manual faults. Complexity measures and errors had correlation.
Recognized failures of medical devices that were recalled due to repeated defects during years 1983-1997, 342 failures	(Wallace & Kuhn 2001)	Logical and calculation faults.
Long-term use of scientific Fortran software (N replicas)	(Hatton & Roberts 1994)	One-off faults.
DB2 (database), IMS (database), MVS (os)	(Sullivan & Chillarege 1992)	Undefined states particularly due to omitted logic. Some common triggers for faults were workload, unusual sequence (e.g. pressing cancel during a query), bug fixes, and recoveries. Assignment and checking faults are assumed to dominate late phases of database development.
Numerous relatively small real-time programs	(Rubey 1975)	Specification-related errors, particularly design consideration, and derivation from specification.
Satellite software (test)	(Dupuy & Leveson 2000)	A conversion fault, logic faults, omission of branches and conditions, value faults, missing functions, and faults in error processing procedures.
Spacecraft controller (SPIN used in experiment)	(Havelund *et al.* 2001)	Wrong task orders, timing faults; 6 faults analyzed.
Voyager and Galileo errors detected during integration and system testing	(Lutz 1993)	Interface faults were present. There were functional faults like behavioral faults, operational faults like omitted or unnecessary operations, and conditional faults like erroneous values on conditions or limits. Omitted operations caused inappropriate triggers (e.g. wrong input caused recovery instead of check of values). Conditional faults caused risk of triggering error recovery inappropriately or failing to trigger the needed response.

Continued on next page

Spacecraft system inspection	(Lutz 1996)	Value out of range, input arrived when it should not have arrived, delays in error response, and no path from high-risk state to low-risk state.
7 spacecraft, 199 anomaly reports	(Lutz & Mikulski 2004)	The most common triggering factor was data access or delivery (e.g. function/algorithm or assignment/limit), also recovery bugs were common.
ATM networks	(Hac & Chu 1998)	Header bit change, buffer overflow. Article presents a method for preventing and correcting buffer overflows.

In (Lutz & Mikulski 2003), *need for* new *requirements* for launched spacecraft systems arose when the software had to handle rare but high-consequence events like unusual paths, requests of data that had just become unavailable, overflows, or rare environmental situations. Another source for new requirements was for software trying to correct hardware failures. In another experiment, latent software requirements were revealed, and unexpected dependencies were identified (Lutz & Mikulski 2004). According to Littlewood and Strigini (1993), design errors, radically new systems, and discontinuous input-output-mapping are *problems* in safety critical software based systems.

Research is being done about *fault patterns*. According to Shull *et al.* (2005), patterns in defect classes have been found in classes of projects. Shull states that making hypothesis and empirically assessing individual studies is not as good a method as a more formal one. The article presents surveys for defects in a specific environment and/or application domain. For example, there are studies about the relation of interface faults and all faults; different studies have contradictive results and define interface in different ways. The study investigates e.g. the difference between new and modified modules, and questions like whether there are omission or commission faults, and how common cause of faults misunderstanding of specifications is. In the study, problems of concepts are discussed; different studies had different content for concepts, which caused problems when studies were compared. One example is the above mentioned concept of interface.

2.2.2 Typical Faults in Specific Environments

Some research has been done about what types of bugs are presented in some specific kind of software (see table 4), or in software made with a specific tool or *methodology*. According to Takahashi *et al.* (1995), structural methodology is more efficient and reliable than text-oriented design methodology. Data definition and interfaces were better understood by using a structural methodology, but those who used a text-based methodology understood specifications better in cases where relevant constituents were distributed over the documents. Gorla *et al.* (1995) compare merits of textual, graphical, and tabular tools, in understanding both tools themselves and process logic.

Yoo and Seong (2002) have analyzed the effect of specification *languages* on fault diversity. They analyzed a black-box language, a dynamic behavior language that used graphs, and a natural language. There were differences in fault types and density. For example, natural languages are inexact and graphs do not always express timing or scope of variables.

According to Sheil (1981), some programming language features are error-prone. The following examples are stated in the article:

- Untraditional operator precedence.
- Assignment as an operator rather than a statement.
- Semicolon as a separator rather than a termination symbol.
- Bracketing to close both compound statements and expressions.
- Inability to use named constants.

Table 5 presents typical faults for some programming languages.

Table 5. Common bugs for some programming languages and environments

Programming Languages		
Env	**Typical bugs**	**Study and possible comments**
C	First and last values, initialization, newlines, command sequence errors, calculation faults involving limit values and termination, order of data items	(van der Meulen *et al.* 2004)
	Pointer bugs, buffer overruns/overflows, mixing = and ==, misusing automatic variables, errors related to declaration and definition	Generally known
	Memory faults	(Xu *et al.* 2004), prevention method
C++	Allocation and deallocation bugs, buffer overruns/overflows, mixing = and ==, misusing automatic variables, and erroneous use of pointers	Generally known
	Memory leaks	(Levitt 2004), smart pointers for prevention
	Limits (even some unusual cases like processing only the last element), processing of special characters, duplicate processing, unimplemented functions, timing, and inexact documentation	(Smidts *et al.* 2002), C++ and waterfall
Java	Timing and synchronization bugs, pointer bugs, bugs related to equality of objects, bugs related to inheritance and overriding, and bugs related to exceptions, initialization bugs, and missing or erroneous checking of return value	(Hovemeyer & Pugh 2004), bug patterns and a detection tool
LISP	Shared variables and side effects	Generally known
FORT RAN	Argument passing, initialization, and overwriting of variables	(Hatton & Roberts 1994)
Cobol	Erroneous sequences, incorrect matching of statement groupings, missing conditions, and missing cases of input data	(Werner 1986)

Continued on next page

Operating Systems		
Env	**Typical bugs**	**Studies and matters involved**
Unix utility programs	Pointer- and array faults (e.g. related to range), initialization faults, faults related to signed characters, wrong assumptions, not defining something that resembles something else, bad address, omissions of checks for errors and end of file, and race condition faults	(Miller & Fredriksen 1990), (Miller, Koski, *et al.* 1995)
Non-Stop UX (basis for UNIX V)	E.g. bugs in device drivers, pointer faults, missing exception checks, and incorrect algorithm or code placement	(Iyer *et al.* 1996)
Linux	Faults related to blocking, pointers, allocation, bounds, and interrupts	(Chou *et al.* 2001), average bug age estimate based on data 1.8 years, logarithmic distribution except Yule for block checker
	The most common effects of fault injection in Linux kernel were reference to null pointer, page fault, invalid operation code, and protection fault	(Gu *et al.* 2003), latencies also studied. About 10% of faults propagate.
Tandem QUARDIAN90 OS	Neglecting unexpected situations (different kinds, e.g. timing, racing, or state), missing routine or operation, and the use of an incorrect constant or variable	(Lee & Iyer 1993), (Lee & Iyer 1995), About 72 % of the faults were recurrences
Maryland Naval	Omissions, bugs related to incorrect facts, and description table access faults	(Fredericks & Basili 1998)
Sperry Univac	Omissions and data definition faults	(Fredericks & Basili 1998)
DOS/VS environment	Configuration and architecture faults; faults related to communication and dynamic behavior, e.g. wrong command sequence, or missing steps like opening a file; faults related to functions offered, e.g. functions had been changed; faults related to calculation, logic, limits, reference, adderssability, or initialization; omitted commands; wrong values; and output faults	(Endres 1975)
IBM MVS	Storage corruption particularly due to allocation, pointer, and buffer overrun errors; and undefined states	(Sullivan & Chillarege 1991), boundary conditions were common triggers

2.3 Features of Faults and Failures

This subchapter discusses about what bugs are like. The first part discusses features of fault prone software, and methods to predict fault proness. The second part involves the question about why so many bugs are hidden. The third part involves failure interaction, fault regions, and the question about how many faults are usually needed to cause a failure.

2.3.1 Features of Fault-Prone Software

Table 6 presents research about features of faults and effect of different factors on fault density.

Table 6. Features of faults and effects of different factors on fault density

Fault proness of files According to numerous studies since the study of Endres (1975), there are fault prone files in computer systems. General research about fault-prone programs has been done. According to Vouk and Tai (1993), fault proness may oscillate as the function of time. According to Ostrand and Weyuker (2002), Fenton and Ohlsson (2000), and Pighin and Marzona (2003), the same files remain fault prone from release to release. However, files containing lots of pre-release faults do not seem to contain so many post-release faults (Ostrand & Weyuker 2002) (Fenton & Ohlsson 2000). Software systems produced in similar environments have roughly similar fault densities (Fenton & Ohlsson 2000).
Failure intensity and number of failure indications According to Shima *et al.* (1997), failure intensities can be different for different software faults and even for faults in the same module. Intensities for some faults can be identical (*ibid.*). One fault may have many failure indications (Munoz 1988).
Symptoms and detection mechanisms Generally speaking, sequences with more operands and larger value ranges reveal more faults than smaller sequences (Doong & Frankl 1994). Relative frequencies of add and delete operations is also a factor (*ibid.*). Howden (1986) investigated the relationship between a missing function and number of times a function is repeated. Lee and Iyer (1993) studied faults in Tandem QUARDIAN90 operating system. Address violation was a common detection mechanism. Most often the immediate effect when the fault was exercised was a single non-address error, e.g. field size, or an index. Symptoms of undefined problems were typically related to overlay or to data structures.
Effect of workload The level and the type of workload affect failures. Several studies indicate that system failures tend to occur during high loads. For example, Chillarege and Iyer (1985) show that this holds for latent errors. According to Woodbury and Shin (1990), high workloads increased the age of hidden faults. Chillarege (1994) studied software probes for self-testing. According to the study, software faults are often partial and/or latent; in the study, partial faults were defined as faults that do not cause a total system outrage. A Combination of latent faults may trigger in high workload and other special situations (*ibid.*).
Fault injection phase Mohri and Kikuno (1991) present a method where the development phase of a fault is found based on location and other information. In IBM, defect types were injected in a specific phase, e.g. assignment faults evolved during the coding phase, and algorithm faults evolved during low level design phase (Fredericks & Basili 1998). For HP, many faults were injected during detailed design and re-design; no formal review was performed after redesigns (*ibid.*). In (Leszak *et al.* 2002), the majority of bugs did not originate in early phases; functionality faults and algorithm faults frequently evolved during implementation phase. Many defects originated in component-specific design or implementation (*ibid.*).

Continued on next page

Fault detection phase Many IBM faults studied by Fredericks and Basili (1998) have been triggered by boundary conditions. Process inference trees were built about bugs detected in different life cycle phases in IBM (Fredericks & Basili 1998). For Sperry Univac, the majority of data handling faults were discovered in unit testing, whereas most data definition faults were found in functional testing (Fredericks & Basili 1998). Higher fault densities in function testing correlated slightly with high fault densities in system testing in (Fenton & Ohlsson 2000).
Fault content during different life cycle phases In Selby's and Basili's study (1991), pretty shallow phase of software development contained the greatest relative amount of errors. In addition, faults that were between initial states of program development and formulation of abstract data types were harder to correct than faults at those levels. The authors assume that it is due to the fact that programmers understand root and leaf levels better than other levels (*ibid.*). According to Selby (1990), effects of multiple testing phases on fault proness depend on the application.
Age of software New files often contain more faults than older ones (Ostrand & Weyuker 2002); see (Pighin & Marzona 2003) for contradictive results. Eick *et al.* (2001) study the risk factors and symptoms for decaying of code, and develop metrics. They also figure out reasons for decay, like inappropriate architecture, violation of design principles, and imprecise requirements. Hochstein and Lindvall (2005) assess how to diagnoze degeneration of code and modify the code so that it remains in conformance with the architecture.
Effect of modifications Modified software may have higher fault density than new software (Fredericks & Basili 1998) (Leszak *et al.* 2002), and the faults may be more difficult to correct (Fredericks & Basili 1998). According to research of University of Maryland Naval Research Laboratory (Fredericks & Basili 1998), modified or re-used modules had higher amount of incorrect or misinterpreted functional specifications than new modules. According to Naval Research Laboratory results, more faults were multimodular in modified than in original modules (Fredericks & Basili 1998). According to Selby (1990), reused components are less fault-prone than new ones, but reuse does not increase the reliability of the whole system. According to Thomas *et al.* (1997), modified components contain lots of interface faults.
Size and structure of fault region Not so many faults are multimodular (Endres 1975) (Fredericks & Basili 1998). Munoz (1988) studied how wide spread bugs are. In the study, the scope of a defect was determined by combinatorial testing. In (Cohen *et al.* 1997), a large number of faults were triggered by several combinations of parameters. In many cases, the set of contiguous input points tended to cause the same failure in (Dunham & Finelli 1990). According to Ammann and Knight (1988), small perturbations in input data may change drastically the probability to detect faults. Failure propagation is related to the width of bugs; it is investigated in subchapter 2.3.2. In the subchapter 2.3.3, research is introduced about how many software variables affect a software bug.

Knowing the *cause* of the faults helps in analyzing fault-proness. There are many studies about looking for factors that cause fault proness, see e.g. (Jacobs *et al.* 2007). Factors for fault proness are being searched with statistical methods. For example, Munson and Khoshgoftaar (1992) have performed discrimination analysis to detect fault-prone programs. Complex *metrics* of the program was data for this study.

There are some contradictive results about the correlation between fault-proness and complexity, see e.g. table 7 and (Subramanyam & Krishnan 2003). Sometimes the contradictions are due to different programming language (Subramanyam & Krishnan 2003). In addition, different complexity measures have different correlations with each other and with fault proness and other quality variables like change effort, Itzfeldt (1990) has a survey. There are numerous different measures for software *complexity*, see e.g. (Peng & Wallace

1993) and (Itzfeldt 1990). According to Güne§ Koru and Tian (2003), high defect modules are those that have almost but not exactly the highest complexity. According to Eaddy *et al.* (2008), if a concern in software (e.g. software requirement) is scattered e.g. across multiple classes or methods, the degree of scattering correlates with the number of defects.

Effect of metrics on fault proness has been studied. For example, Subramanyan and Krishnan (2003) survey experiments involving CK metrics for object oriented software. Vouk and Tai (1993) present metrics variables and discuss their ability to predict fault prone products and problems in defect prediction. There are studies about correlation between metrics variables, e.g. between CK variables, see e.g. (Subramanyam & Krishnan 2003). Fault proness depends on the application and on development methodology. Some comparative studies have been made, and there have been differences; e.g. Smidts et *al.* (2002) compared waterfall and formal development of C++ code. In (Leszak *et al.* 2002), different software domains had vast differences in defect attributes even within the same project. The study also involved correlation between defect density and both process compliance metrics and static metrics. Tian and Troster (1998) inspected tree-based defect models that link defects to a quality indicator. Table 7 presents some metric variables and their effect on fault proness.

Table 7. Examples of studies about effect of measures on fault density

Metrics	**Effect**	**Studies**	**Application /Environment**
Cohesion	No effect	(Briand, Wüst, *et al.* 2000)	Object oriented programs
High module strength	Lowers	(Card *et al.* 1986)	Functional programs
		(Selby & Basili 1991)	See footnote[2]
Global variables	Increases	(Card *et al.* 1986)	Functional programs
Coupling	Increases	(Selby & Basili 1991)	See footnote[2]
		(Succi *et al.* 2003)	2 C++ projects
	Generally increases, some measures decrease or have no effect	(Briand, Wüst, *et al.* 2000)	Object oriented programs
	Depends on language and depth of inheritance	(Subramanyam & Krishnan 2003)	C++, Java
Number of descendants	Increases	(Card *et al.* 1986)	Functional programs

Continued on next page

[2] An internal software library tool that contains several languages. The static source code metrics was constructed from the portion written in a PL/I -like high level source language.

Inheritance-, coupling-, and class-related measures, including depth of inheritance and number of immediate descendants	All measures increase, some (depth of inheritance and number of immediate descendants) decrease in other studies referred	(Briand, Wüst, *et al.* 2000)	Object oriented programs
7 CK metrics including inheritance depth and number of immediate descendants	Generally increase, depth of inheritance may decrease and number of immediate descendants decreases	(Succi *et al.* 2003)	2 C++ projects
Depth of inheritance	Increases, decreases, and then increases again as inheritance deepens	(Selby & Basili 1991)	See footnote[2], inheritance is nesting of processes
	Usually increases, depends on coupling	(Subramanyan & Krishnan 2003)	C++, Java
Number of methods in a class	Increases	(Subramanyan & Krishnan 2003)	C++
	Decreases	(Subramanyan & Krishnan 2003)	Java
Module size	Depends on application	(Ostrand & Weyuker 2002)	Compare different studies
		(Hatton 1996)	Compare different studies
	Increases	(Subramanyan & Krishnan 2003)	C++, Java

Bieman *et al.* (2003) studied *design patterns* and *change proness*. The change proness of classes used in design patterns was different in different cases. For example, the class size sometimes increased change proness. There are other studies about design patterns, too. For instance, Vokáč (2004) analyzed the correlation between the appearance of some design patterns and faults in C++ software. According to the study, there was a negative correlation. Some tools can look for fault patterns. Livshits and Zimmermann (2005) present methods and a tool for detecting new *fault patterns* using *revision histories*, and detecting their violations.

Table 8 presents classification methods for predicting fault-prone modules.

Table 8. Methods for predicting fault proness

Prediction method	**Authors**
Association mining.	E.g. (Chang *et al.* 2009)
Logistic regression, classification trees, and optimal set production. The latter study also involves pattern recognition to analyze data for software process planning.	(Briand *et al.* 1993) and (Briand *et al.* 1992)
Finding high-risk components with optimized set reductions based on software properties like number of global variables and nesting. The method is compared with trees and logistic regression.	(Briand *et al.* 1993)
Using continuous attributes in classification trees.	(Morasca 2002)
Logistic regression and rough sets are assessed as means of fault proness measurement, and a hybrid model combining both is built.	(Morasca & Ruhe 2000)
Non-parametric discriminant analysis.	(Khoshgoftaar *et al.* 1996)
Boolean discriminant functions.	(Schneidewind 2000)
Case-based reasoning.	(Khoshgoftaar *et al.* 1997)
Fuzzy decision trees.	(Suárez & Lutsko 1999)
Hyperbox algorithms in classifying software quality. Fuzzy box and genetic algorithms are presented.	(Pedrycz & Succi 2005)
A forest of learning decision trees.	(Guo, Ma, *et al.* 2004)
Some computer risk identification techniques are compared. Tree-based defect models are analyzed in identifying and characterizing fault-prone modules.	(Tian *et al.* 2001)
Statistical approach for measures of Java class fault proness. A model is presented, applying model to different software than the one it was made for is assessed, and ability of several methods like regression-based MARS is assessed.	(Briand *et al.* 2002)
Statistical dynamic bug searching for software with multiple bugs. The method in the study is based on making clusters of predicates that are true in bug situations.	(Zheng, Jordan, *et al.* 2006)
A model for finding files with largest number of faults and largest fault densities. The predictions are based on change history and fault parameters like file size.	(Ostrand *et al.* 2005)
Methods for building models for measuring fault proness for different applications.	(Denaro *et al.* 2002)
A project based measure about costs of misclassification. A table of used metrics variables is introduced.	(Khoshgoftaar *et al.* 2005)
An approach based on resources and events in development. The lack of experience of the programmer, failure history, late substantial modifications, software involved in the late design change, or uneasiness of developers, may be indications of fault-prone routines.	(Hamlet & Taylor 1990)
The combination of principal component analysis and neural network method to find sets of high-risk modules. It is stated that usual correlation and factor analysis-based methods result in too much correlation between classes.	(Neumann 2002)

2.3.2 Hidden Bugs

There are some classes of bugs that are hard to observe, the following list presents some such bug types. A *Bohrbug* is a bug that is solid and easy to detect (Gray 1985).

- A *context preclude* means that some configuration or set of memory constraints makes it impractical or impossible to use the debugging tool (Eisenstadt 1997).
- A *stealth bug* is a bug that consumes the evidence of itself (Eisenstadt 1997).
- *Heisenbug* goes away when one tries to look it (Gray 1985). For instance, it may go off when one turns on the debugging tool (Eisenstadt 1997), or it may be caused by a racing condition (Bourne 2004).
- *Mandelbug* is so complex that it is hard to predict the failure occurrence or non-occurrence, due to the activation or propagation of fault caused by either of the following two reasons presented by e.g. Grottke and Trivedi (2007). There may be a delay between failure activation and failure occurrence, so it is difficult to identify what caused the failure; repetition of the step does not necessarily cause the failure again. Another reason is that other elements of software system (e.g. operating system or hardware) can influence on software behaviour.
- *Schroedinbug* manifests itself only after unusual software use or use in a new situation or reading a source code, when the person who did the activity notices that the software never should have worked and it stops working for everyone until fixed (Raymond 2003).

Kelly and Shepard (2004) present problems in testing. According to the study, *multiple symptoms* may disguise the root cause of a problem. The authors also discuss the *cause/effect chasm*. The cause/effect chasm means that the symptom of a problem being far removed in space or time from the root cause (Eisenstadt 1997).

Hidden *bugs may survive tests*, even if the code is executed during the tests. A fault may remain hidden for a long time; see e.g. (Sullivan & Chillarege 1991). According to Sullivan and Chillarege (1991), executing a piece of code of a tested operating system in exceptional environment like rare prior state, loading, or input data, may trigger field failures. In (Cai *et al.* 2005), special combinations of input, noise, and failure detection process triggered common mode failures. Testing usually does not reveal external faults, see subchapter 5.1.1 and the introduction of chapter 5. In addition, some external conditions may modify for example heap and memory content, and this may affect test results (Whittaker 2001).

Systems do not always detect the first symptoms, and the errors may *propagate* (Sullivan & Chillarege 1991). In the study, memory corruption errors corrupted only few bytes, which made them harder to detect. In several studies, e.g. (Lee & Iyer 1993), a propagation concept "further corruption" means that consequences of a fault are first used without the fault being detected and later the fault is detected by a task not related to the one that accessed the first fault for the first time.

There are many studies about error propagation. According to Michael and Jones (1996), the errors in software code that affected data state behaved homogenously in the vast majority of cases: either all or none of them propagated to output. The amount of code executed after perturbation (after fault injection), the number of perturbations per location, and the extent to which the perturbed values differ from the original ones, did not have much effect on the homogeneity results. Voas (1992) presents a technique that contains analysis of probabilities that a fault affects specific section of software, probabilities that the section affects state, and probabilities that the state affects output. Also input error or data corruption may lead to erroneous output (Voas & Miller 1995a).

Models have been developed about bug propagation. For example, Okun *et al.* (2004) study propagation of logical faults. Wooff *et al.* (2002) describe graph models containing probabilities that a fault transfers to another node. Estimation of root node probabilities is presented and sensitivity analysis for software changes is discussed in the article.

Binkley and Harman (2004) study how many global variables and formal parameters a typical failure depends on. Voas and Miller (1995a) studied propagation by using information about variables that were altered by wrong input. Bishop and Bloomfield (2003) studied execution profiles of program statements and their effect on failure rate. There are studies where error propagation is analyzed by performing flow analysis, see (Murrill 2008). Bug detection by reasoning about logical dependencies and assumptions is a topic for research, see e.g. (Frankl & Weyuker 1993a). Bug detection by locations of variables is a related issue. Murrill (2008) does not recommend static flow analysis as an analysis method for error propagation.

Error propagation in operation systems has been studied. In (Lee & Iyer 1993), Tandem QUARDIAN90 operating system errors that were detected quickly either were bound to be detected on the first access, or the first error could be accessed without being detected and the problem was detected by the task that accessed the first error for the first time. In the study, errors with long latency were susceptible for causing further corruption. Overlay errors (errors that corrupt memory) often propagate and corrupt data in MVS operating system in (Sullivan & Chillarege 1991). A propagated error can often defeat the established recovery mechanisms (*ibid.*).

Even if there are faults in software, the output may be coincidentally correct. According to White and Sahay (1985), two or more inequalities may combine to impose an equality restriction on the input domain; this is called *coincidental equality*. White and Cohen (1980) presented *coincidental correctness*. Coincidental correctness means that there is a fault in software and output variables coincidentally are the same as if the domain and the computation had been correct. A good example of coincidental correctness is an equation x+x=x*x when x=2 or x=0. Software may also contain faults that cover each other's effect, so the result is correct (Abbott 1990).

Blindness means that an erroneous predicate may produce correct output when it is in touch with other context. Blindness is one form of coincidental correctness. Any linear combination of errors involving assignment and equality blindness may cause blindness (Zeil & White 1981). Table 9 presents common forms of blindness.

Table 9. Basic blindness

Blind-ness	**Explanation**	**These two cannot be distinguished**		**if this holds**	**Source**
Assign-ment	Functions may evaluate to zero when they are parts of expressions (Zeil & White 1981).	U+2*A-T>B	U>B	2*A-T=0	(Zeil & White 1981)
		X>0	A>0	X=A	(Zeil 1983)
Equal-ity	This type of blindness is due to equality restrictions (Zeil & White 1981).	C+D>1	C>0	D=1	(Zeil & White 1981)
Self	An expression cannot always be distinguished from its multiples (White & Sahay 1985).	X-1>0	X+A-2>0	X=A	(White & Sahay 1985)
		X+1>0	2*X+2>0	true	(Zeil 1983)

There is implicit *information loss* when two or more parameters give the same result. There is explicit information loss when variables are not validated; for example, if a module does not release information that other modules could potentially use. Function type is a factor in susceptibility to information hiding. For example, there is no information loss for function f(a)=a+1, where a is an integer argument. Modulo function f(a) = a mod b has information loss, and testability of the function decreases as b decreases. (Voas & Miller 1995b).

Voas and Miller propose specification decomposition to reduce cancellation of effects of faults. They recommend minimizing variable reuse, and increasing the use of out-parameters (output variables exclusively for testing). The study assumes that there is a single fault and it is in one location. The study also discusses mutations, executions, and a method to estimate the probability of infection. (Voas & Miller 1995b).

2.3.3 Number of Faults in a Failure Situation

Bugs may interact with each other. The amount of interaction and some reasons for it are discussed below. The portion of code that the bug may affect is discussed too; for example, the question about how many variables a failure is dependent on is investigated. The breadth of a bug has an effect on propagation. Propagation was discussed in the previous subchapter.

If random input is used, independent versions of software do not fail independently (Littlewood & Miller 1989). Bishop (2006) surveys experiments made about intentional diversity in design between different versions of software that all perform the same task. According to those studies, different versions have *common faults*. According to Bishop, the primary cause for common faults in final program versions has been inexact *specifications*. Yoo and Seong (2002) analyzed an N-version system where different versions had different specification languages. There were *matters that tended to cause defects in several replicas* (Yoo & Seong 2002). Diversity is useful in inspection, too: research shows that people who inspect software from different perspectives find different faults (and only few faults are found by several inspectors who have different perspectives), see e.g. (Laitenberger & DeBaud 1997). Correlated failures are discussed in subchapter 6.3, too. See also subchapter 5.1.2 about test coverage.

Brilliant *et al.* (1990) look for cases where programmers make equivalent faults. Apparently different logical faults may yield faults that cause statistically correlated results, and faults that seem to correlate do not necessarily correlate (*ibid.*). Some special cases in input space like cases of three points where two or more of them coincide or all are parallel, are subject to correlated faults (*ibid.*). Another type of correlated faults in the study was a precision bug; for example, cosines were compared instead of angles. In (Cai *et al.* 2005), common mode failures were due to e.g. initialization, computation, or precision bugs, or system coherence faults.

Dunham and Finelli (1990) introduce studies about characteristics for real-time failures. Typical features for real-time programs are repeated execution, fast iteration rates, and correlated inputs (*ibid.*). In the studies, widely varied failure rates were observed, faults *interacted* in non-intuitive ways, and some failures were caused by interaction of several faults (*ibid.*).

Table 10 describes research about how many faults are needed to cause a failure. At least some of those studies involve correlated faults.

Table 10. Studies about number of events or conditions affecting a failure

Fai-lures	**Field**	**Result**	**Study**
97942	N-version experiment	Rare situations cause severe errors. All software versions that failed under three rare faulty inputs had already experienced failures under two rare faulty inputs.	(Eckhardt *et al.* 1991)
365	Browser and server	Approximately 90 per cent of failures were caused by three or fewer conditions. None of the failures were caused by more than six conditions.	(Kuhn & Reilly 2002)
109	Medical devices	106 faults were due to values of 1-2 conditions, 2 had 3 conditions, and only 1 had 4 conditions. None of the faults had more than 4 conditions.	(Wallace & Kuhn 2001)
329	NASA distributed software, development and integration testing	1-way, 2-way, 3-way, and 4-way bugs represented respectively 68%, 93%, 98%, and 100% of bugs.	(Kuhn *et al.* 2004)
Not stated	Dynamic fault tree models	Modeling-feature interactions that arise only in cases involving 15 or more events are possible.	(Coppit *et al.* 2005)
Not stated, >100 000	Dimensionality model to characterize triggers for robustness failures	The input arguments were used as model parameters, each representing one dimension. Approximately 82% of all failures were caused by a single parameter in the study.	(Pan 1999)

Brilliant *et al.* (1990) studied *how many faults had to be corrected* to correct failures caused by multiple faults. According to the study, some such failures could be fixed by correcting only one of the faults, and some could be fixed by correcting two or more such faults. One NASA study had a similar result: fixing either of two different faults could correct the same failure (Dunham & Finelli 1990).

Some research has been done about *how many software variables* have effect on a specific software failure. According to Binkley and Harman (2004), a typical predicate in C-program studied depends on 72 % of formal parameters, and on 48 % of global variables that can potentially be used or defined by a call to the predicate's procedure, and on 2.4 % of global variables in the scope. The study also computed correlations involving formal parameters and global variables visible to a predicate; both were compared to the proportion actually used and the size of the function. There was no correlation between the number of formal parameters and that of global variables.

There is research about what types of bugs can be detected by *combinatorial testing methods* and how effective they are. See also subchapter 5.1.2 about test coverage. According to several research documents, pairwise testing is very efficient means of finding faults, see e.g. (Dalal *et al.* 1998) and (Cohen *et al.* 1997). Pairwise testing means that all 2-way combinations of parameter values are tested. Smith *et al.* (2000) compare testing of all-pairs (all pairwise combinations of parameter values are tested) and all values (all parameter values are tested at least once, and only one parameter has different value than in a test case that gives a correct result) for a remote agent planner software in Deep Space -mission. When all pairs- and all values –methods were both used, 88% of timeouts and correctness bugs were detected by those methods but only half of the interface and engine bugs were detected (*ibid.*). All pairs -method detected only 20 % more bugs than all values- method (*ibid.*). All values -method missed more timeouts than correctness bugs with respect to all pairs (*ibid.*).

Mutation testing means that small changes (mutations) are made to the module, and results of the original module and those of the mutated module are compared. If the results differ, it can be said that the mutant has been killed. Mutation testing is often related to coupling of faults. Some faults have very simple mutations and some do not (Howden 1982). Sometimes testing elemental mutants fails but more complex mutation testing may succeed (Howden 1982). Woodward and Halewood (1988) study an intermediate technique between using elemental and all mutations. (Offutt 1992) supports the hypothesis that test data sets that detect all simple faults detect large percentage of complex faults. K.S. How Tai Wah (2003) discusses a model to study coupling and studies the effect of killing mutants with one fault in mutation testing on mutants with more faults. Research is also being done about orthogonality of mutants, see e.g. (Lee *et al.* 2004) about orthogonality of mutants in object-oriented mutation testing. Saglietti (1990) investigates metrics of dissimilarities in input partitions between different versions of software.

2.4 Faults and their Causes

Knowing what causes bugs helps preventing them, so more and more attention is being paid to the causes of software faults. Possible causes of faults are investigated in this subchapter. In addition, classifications that involve causes for bugs are surveyed.

Many bugs are due to carelessness (Maxion & Olszewski 2000), (Nakashima *et al.* 1999) or lack of understanding. According to Spohrer and Soloway (1986a), many faults like type mismatches may be due to misunderstanding the semantics of the programming language. Some logical faults were due to erroneous reasoning (*ibid.*). Some other bugs may be due to misinterpreting specifications or mixing local and global (*ibid.*). Chou *et al.* (2001) studied causes for Linux-bugs. In the study, a common cause was that programmers do not understand the system well and do not know what all functions do in usual and more exceptional situations. There were copy-paste-faults probably due to ignorance or the fact that programmers trusted each other's work. In (Cai *et al.* 2005), common causes for faults were misunderstanding of specifications and lack of knowledge of application area or programming language. Changing only parts of the code caused bugs in (Sullivan & Chillarege 1991). Keeping track of cross references would help prevent those faults (*ibid.*).

Faults may be due to wrong assumptions. Table 11 introduces some types of wrong assumptions.

Table 11. Some typical wrong assumptions

Implicit assumptions. Jacobs *et al.* (2005) mention implicit assumptions not communicated to other projects. The study is about virtual teams.
People assume that other people test the software, see (Nakashima *et al.* 1999).
Programming environment. McKeeman (1975) studies how to prevent false assumptions about the programming environment. The study presents examples of false assumptions about the programming language as a cause of faults.
Evalutaion errors. Gerhart and Yelowitz (1976) found that evaluation errors were common in proven programs. For example, the programmer might have had assumed left-to-right ordering and the compiler may have performed computations in another or non-deterministic order (*ibid.*).
Type mismatches. Bug fixes in MVS operating system often changed message format or data structure organization, and other modules had implicit assumptions that contradicted those changes (Sullivan & Chillarege 1991).

Continued on next page

Special items. Maxion and Olszewski (2000) mentioned assumptions about special items, e.g. an assumption that there were a maximum number of rows/columns of data when there were not.
Independency. False assumptions about independency between sections of work products (sections can be, e.g. components, requirements, or constraints) are mentioned in (Conte de Leon & Alves-Foss 2006); the study involves detecting dependencies.
Allowing transformations which are valid only in some parts of their domain without recognizing those parts may cause bugs (Fateman 1990).

Some attention is being paid to the lack of understanding. IBM has had a level of experience in its fault trigger classification (Kelly & Shepard 2001), and Kelly and Shepard (2001) associated a level of understanding with each defect category in their bug classification. Spohrer and Soloway (1986b) studied how novices think when they make bugs. Jiang *et al.* (1999) studied what people with different orientations perceive as reasons for system failures; reasons were related to user and developer, processes, and organizational and technical environments.

Some design methods and programming languages support the concept of *mode*. Leveson *et al.* (1997) discuss mode confusions and reasons for mode confusion. Bachelder and Leveson (2001) introduce a model about a military helicopter system. By using the model, the following mode confusion errors were identified in the system: unintended side effects, indirect mode transitions, inconsistent behavior, ambiguous interfaces, and the lack of appropriate feedback. Operator authority limits are mentioned as the sixth class. According to Sullivan and Chillarege (1992), there are several causes for *missing state* bugs: some of those bugs are due to ignored cases, some are due to technical software functioning like setting state flags incorrectly, and some are due to situations where software misinterprets events and makes faulty transactions.

More and more studies have *classified* both bugs and their causes. Some studies find a cause for each bug type, and some studies classify bugs and their causes orthogonally, (Leveson 2001) is a good example of both actions. Classification methods can be based on some specified criteria (Fredericks & Basili 1998); or on data similarity, cluster analysis (Podgurski *et al.* 1999) is a good example of the latter. More and more attention is being paid to human errors. Nakajo and Kume (1991) had classified the type of each fault as either of three types: internal, functional, or interface fault. Cause-effect relationships of software errors are analyzed from data of observation points in a fault tree. The observation points form a three-level hierarchy of fault classification: fault type, human error, and process flaw that caused the human error. Typical human errors are misunderstanding of specifications or program function. According to the study, human errors may be due to a process flaw like inappropriate definitions and lack of methods to record and report them.

Lutz (1993) studied and compared faults in safety critical parts and non-safety critical parts of safety critical software. According to Lutz (1993), faults were often caused by misunderstanding specifications, and a common root cause for this was often communication errors between development teams. In the study, each root cause was associated with a process flaw or inadequacy in the control of system complexity, and with process flaw in communication or development methods used. Inadequate identification or understanding of interfaces or requirements, and interface design during system testing, were common flaws, particularly for safety-related faults. Other common flaws especially for safety critical systems were imprecise, incomplete, or unsystematic specifications; missing, unknown, undocumented, or wrong requirements; and insufficient design of requirements (Lutz 1993). An individual mistake was the most common human root cause and the lack of system knowledge was the second common human root cause in (Leszak *et al.* 2002). According to

Hansen *et al.* (1998), many failures in an air data system have been due to the insufficient understanding of interactions between air data system, flight control laws, and redundancy management software.

Leveson has studied common factors for software-related aircraft and spacecraft accidents. Software faults have been only one of the many factors in those accidents. According to the study, the vast majority of software-related spacecraft accidents involve incomplete requirements specifications and unhandled or mishandled software states or conditions. Leveson linked mechanic events to conditions or lacks of conditions that allowed them. Those conditions were linked to system factors. Leveson identified striking similarity in system factors in all those accidents. Leveson classified system factors as flaws in safety culture, ineffective organizational structure and communication, and ineffective or inadequate technical activities. (Leveson 2001).

Nakajo *et al.* (1993) studied methods for preventing the lack of communication in software development process. Those methods had originally been developed for hardware. Bug tracking tools help in sharing knowledge about bugs during and after software development. There are several tools for bug tracking, e.g. Bugzilla on https://bugzilla.mozilla.org/ is a public bug tracking tool.

2.5 Summary of Avoiding Known Bugs

Catalogues of bug types have been created. Different bug classifications are being developed for different purposes, and automatic classification methods are being developed. What a good bug report should contain is also being studied. There are bug databases for increasing bug knowledge. The temporal development of bugs has not got much attention by research people. Subchapter 2.1.3 contains a short study about the topic. Simple bug types like character- and calculation faults have been present all the time. There have been inversion bugs at least from 1970's. From the 1980's, software has become more complex and bugs related to e.g. stress, order of operations, timing errors, missing states, sizes of data elements, and user interfaces have been present. Correlation between different bug types could be studied more. There is empirical research about how long a bug lasts, but the problem is hard to solve empirically: there is a possibility that a bug is never detected.

Characteristics of bug types have not got much attention by researchers. Research is being done about fault patterns. Attempt has been made to find fault patterns by comparing different studies, but there have been problems in comparison; e.g. in Shull *et al.* (2005), there was no common definition for interface bug, which has made comparison difficult. The study is discussed in subchapter 7.1.2. Fault patterns could be looked for with other methods, too. Some statistical and other kinds of methods to look for fault proness have been surveyed in this chapter. How much these methods and other means help in detecting characteristics of faults could be a topic for research. Knowing characteristics of an individual bug type could help detect fault patterns, and vice versa.

There are studies about problems and numbers and types of faults found in different application domains. Omissions and logical faults are common in many application areas. What causes the need for new requirements is also studied. Safety critical systems have some specific problems. Research is also being done about error-prone features of programming languages and methodologies, and about frequent faults when using a specific programming language or operating system. Different methods, programming languages, and operating systems have different fault prone features.

There is a lot of research about fault prone files, predicting fault proness, effect of the life cycle phase on fault content and on easiness to correct a fault, effect of age and legacy on number and types of bugs, and when and where bugs are born. Size and structure of failure region for a fault is being studied. What software features or software development features correlate with bugs or help detect or correct bugs is being investigated. For example, complex sequences often reveal more faults than easy ones. Bug symptoms, fault detection mechanisms, and effect of workload on fault detection are also being studied. One bug may have several symptoms. When faults are detected and what triggers them are also studied. The effect of metrics on fault proness depends on the development environment and the application domain. Some studies involve design patterns. There are also statistical methods for looking for bugs, e.g. predicates that are true in failure situation are being caught.

Types and classes of hidden bugs are being discussed. Some bugs are hard to observe because they for example appear in rare situation or high workloads or give remote symptoms or several symptoms. Those bugs may be hidden for a long time. Not all hidden bugs appear in rare situations. There are studies about information that can be hidden from testers. In addition, testers do not necessarily detect fault symptoms. Also, the user may start to use the program in a new way, which may reveal hidden bugs. Research has been done about error propagation. For example, how dependencies of variables affect error propagation is being studied.

Software faults may also cover each other, or environment may cover bugs. Even if there are faults in software, the output may be coincidentally correct. For example, if the software multiplies two numbers when it should add them, the result is correct if both the numbers are twos or both are zeros. Some forms of coincidental correctness and blindness are presented in this thesis.

Researchers have not been emphasized that faults covering each other is an undesired phenomenon. There may be some situations where those faults do not cover each other. In addition, changes in software or in the environment may trigger those failures. According to some studies, fixing one fault may eliminate a failure caused by several faults. The studies do not critisize this approach even though it is very dangerous: the other faults still remain hidden and may actualize in other situations. When not all the faults are corrected, the remaining faults remain hidden and may be disastrous with other conditions (e.g. other input or different memory content) or after code changes.

Several methods should have wider application area than what they have. For example, more attention should be paid to the phenomenon that expressions may cover faults in other expressions. This is considered mostly in domain testing but could be considered in other connections, too. Blindness and coincidental correctness are investigated in connection to domain testing; they are rarely discussed outside the field of domain testing. They should be taken into account when investigating e.g. static analysis, risk-based testing, or fault-based testing.

There are cases where programmers make equivalent faults. For example, many geometric and trigonometric problems are like that. Faults may interact in non-intuitive ways. Generally speaking, one or two rare conditions are enough to trigger a failure, but sometimes even 15 conditions are needed. How many variables a failure depends on is also being studied. Some studies compare the fault detection ability of several combinatory testing methods. Types of faults found by these methods are being compared, too. Some studies assess a testing method that may reveal a connection between parameters or fault regions. How complex mutants need to be in mutation testing is also being studied.

As stated above, correlation between different fault types could be studied more. However, the number of faults in a failure situation has been studied a lot. Abstract models of fault correlation, too, have been studied quite a lot in the context of defect prediction or modelling of configurations of components. Fault proness has been studied, too, as stated above.

More and more attention has been paid to the root causes of faults. Typical root causes are carelessness, lack of knowledge, lack of communication, misunderstandings, wrong assumptions, and insufficient specifications. What software faults cause mode confusion or missing state is also being studied. Several classification systems classify faults and their causes. The root causes are often the same even when software faults are of different types. Human errors like misunderstandings may be caused by process faults.

3 DEFECT PREVENTION AND PREDICTION

Chapter 2 investigated information about software bugs in general. The rest of the thesis involves failure avoidance. In this thesis, the failure avoidance means are classified as means for fault prevention, fault prediction, fault detection, and fault tolerance, see chapter 1. This chapter involves methods intended for preventing and predicting software faults. The first subchapter involves means for fault prevention, although those means have connections with other areas of failure avoidance. The main topic is the disciplined software development process and its measurement, but some other topics are discussed briefly. The second subchapter discusses defect prediction like risk analysis, prediction metrics, and defect prediction models. The topics of the second subchapter, particularly risk analysis, have connections to fault prevention, fault detection, and fault tolerance, too. The last subchapter is a summary of this chapter.

3.1 Defect Prevention

This subchapter involves means for defect prevention. The first part discusses general means for defect prevention, like good software engineering practices (particularly the software development process), measuring, modelling, and reverse engineering. Some fundamentals of software engineering are also briefly discussed. In the second part, the relationship between bug prevention and different phases of the software life cycle is discussed.

3.1.1. General Means for Defect Prevention

According to a study of Tervonen and Kerola (1998), different people have different ideas about *what software quality is*. The study involves how to achieve co-understanding. There are studies about how to make quality assurance more practical, see e.g. (Koono & Soga 1990). Rai *et al.* (1998) classify software quality assurance into the following dimensions: technical, managerial, organizational, and economic. This thesis involves the technical dimension. Good software has different attributes like correctness, efficiency, and maintainabiltiy, and this thesis is about bug elimination, i.e. correctness.

Good software engineering practices like a disciplined development *process* and the use of appropriate methodologies, tools, and metrics, are an efficient way to prevent bugs. (SWEBOK 2007) contains a systematic framework for software engineering, including processes, methodologies, and best practices. It also includes managerial and organizational issues whereas this thesis is about technical issues.

Germain and Robillard (2005) studied effect of the process used on cognitive activities. The aggregate variation due to process was small and limited. Process used had an impact on control tasks like inspection and review.

There is a lot of research about process improvement. For example, *measuring and modeling* process quantities helps improve the process. According to the survey and experiment of Green *et al.* (2005), quality and productivity perception helps improve a software process. See (Rainer & Hall 2003) about analysis of factors that affect software processes. Research is being carried out about statistical process control. Statistical process control means looking for trends, cycles, and irregularities, and improving process or product based on those trends (Peng & Wallace 1993). Software quality metrics is discussed in subchapter 3.2.2. There are studies about software process evaluation like validation, see e.g. (Cook & Wolf 1999).

Some studies present *errors in theories* that are related to software engineering. For instance, Morris and Bunkenburg (2002) present inconsistency in theories of non-deterministic functions. Xia (1999) investigates flaws in software engineering like in definitions, interpretations and representations. For example, some circular definitions are criticized, and the inconsistent interpretation of the measure of length is discussed in the study. Inconsistencies in models cause software bugs and are a topic for research. Medvidovic *et al.* (2003) study discontinuity between models. In the study, connectors transfer or compare information between different models of the same or different artifacts. Different views of software and software inconsistencies can be modelled, too, see e.g. (Grundy *et al.* 1998) and (Liu *et al.* 2002). According to Grundy *et al.* (1998), there can be inconsistencies between views, between developers, and between development phases. Moynihan (2000) studies requirement uncertainty.

Some studies were found about the relationship between *reverse engineering* and bug elimination. Tian (1996) searches problem areas in testing; the search is executed by tree based analysis, where each node has a responding attribute and splitting is performed according to values of the attribute. Taghdiri (2004) derives specifications from code and detects errors in code by analyzing those specifications.

3.1.2 Relationship between Life Cycle Phases and Bug Elimination

The bug elimination means mentioned in subchapter 3.1.1 can be used in all phases of software life cycle. There are many methods that are used in one or more phases of the software development process. Ramamoorthy *et al.* (1986) classify methods according to weather they are phase dependent or phase independent. For example, risk analysis can be considered phase independent, although it may have different content in different life cycle phases. Review and testing should be executed in every phase of the life cycle. There are models about software development life cycle. For example, Boehm (1988) presents waterfall model and spiral model. Those models often contain requirement engineering, architectural design, modular design, coding, and testing.

Making goals should be the first part of software development process but requirement specification or risk analysis is often considered the first phase of it. It is hard to achieve goals if they have not even been specified. There are some studies about goals, and more and more research is being done about them. See e.g. (Fredericks & Basili 1998) about setting goals and measuring whether they are achieved. (Stallinger & Grünbacher 2001) is one example of numerous studies on software process -related simulation; the aim of the method in the study is to achieve convergence towards a goal. After making goals, later phases of software life cycle can be performed. Sometimes one can return to the previous phase or even an earlier phase than that during the development cycle, e.g. Boehm (1988) presents a spiral model for a software life cycle.

van Lamsweerde *et al.* (1998) present formal generic refinement techniques for making specifications from goals. Using this framework, conflicts between goals can be eliminated. For example, new goals can be set or goals can be weakened. van Lamsweerde and Leiter (2000) analyze exceptions for a single goal. All obstacles are formalized. Operations in the study are conditional input-output relationships over objects. Techniques for elaborating goals and generating obstacles from goal specifications and from domain properties are presented. Methods for resolving and preventing obstacles are introduced. The study also includes classifications of obstacles.

Requirement engineering is an important way to prevent bugs. Sommerville and Ransom (2005) detected a correlation between increased business benefits and improved requirement engineering process. According to the study, it is hard to know if there is a casual relationship. There is a lot of research about requirement engineering. The relationship between bugs and requirement engineering is investigated in chapter 4, particularly in subchapter 4.5.1.

Architectural Design Considering architectural design, safe memory allocation and deallocation is closely related to prevention of memory bugs, see e.g. (Gay & Aiken 1998) about memory regions and (Dhurjati *et al.* 2005) about memory safety. Ideal module size to reduce faults is discussed in subchapters 3.2.2 and 2.3.1. Godfrey and Zou (2005) study the detection of merging and splitting of functions.

Some architectural models consider failures, see e.g. (Perry & Wolf 1992) for a model that involves erosion due to violations over time, and drift over time due to insensitivity that leads to inadaptivity and then lack of coherence and lack of clarity of form (relationships between weighted properties). See subchapter 4.2.1 about modelling interfaces.

Modular design and coding are often considered as phases that cause bugs even though root causes are in earlier phases and general policies, see subchapter 2.4. However, bugs are born in these phases, too, as described in chapter 2. How to detect those bugs is investigated in chapter 4.

Testing could be performed during and after each phase of life cycle. Very often it is executed only after coding. Chapter 5 is about testing.

Maintenance and configuration management are usually not included in software development models, like V-model or spiral model. However, they are important parts of the development process. (SWEBOK 2007) has maintenance and configuration management as different topics in the description of phases of software life cycle. There is a lot of research about changes made to programs. Some examples are mentioned here. Phihip (1998) investigates issues like cohesion, proper program structure, scope of variables, and documentation of variable names. The article is about maintenance of event-driven software, but the same ideas help avoiding bugs in other life cycle phases, too. Elbaum and Diep (2005) study fault-, coverage-, and invariant detection in software by profiling released software. The study also involves improvements to profiling. There is research about looking for dependencies before making changes, see e.g. (Robillard 2008). Change impact analysis is also under investigation, see e.g. (Badri *et al.* 2005). There are lots of studies about change proness of software, see e.g. (Bieman *et al.* 2003). Also, there is software metrics that is related to maintenance, see e.g. (Peng & Wallace 1993). Cleland-Huang *et al.* (2008) study how critical goals are maintained throughout the lifetime of software system. (Rai *et al.* 1998) is a survey about maintenance research.

A Phase Review is recommended, at least at the end of each phase of the software development cycle. However, not many studies have been done about phase reviews. Dunham and Finelli (1990) present the software development process for NASA Langley laboratory, where DO-187A -standard had been implemented. In that process, verification, validation, and quality assurance activities were integrated in every life cycle step. Requirement traceability matrix had been used. In design review, it was checked that requirements had been translated correctly, no additional functionality had been added, and design standards had been followed. The code review included checks of interface, hierarchy, pseudo-code, and whether the coding standards had been followed. See (Ramamoorthy & Bastani 1982) about reviews like verification after testing phase. The article, like many others, involves metrics about testing, like number or faults remaining.

Tracing means linking different results. Tracing guarantees that goals are achieved. For example, requirements are often traced backward by linking them to goals, risks, and different kinds of models and documents where they originate, and forward to software architecture. The architecture can be linked to design and code. See e.g. (Gates &

Mondragon 2002) about tracing between and within life cycle phases. When tracing is used and a fault has occurred, forward search can be performed from failure modes to effects and backward search from hazards to contributing causes, see e.g. (Feng & Lutz 2005). Tracing can also be done between or within artifacts created during the same phase of life cycle (Ramesh & Edwards 1993). Gates and Mondragon (2002) survey tracing approaches and develop methods for tracing of constraints. Some tracing tools are presented in the study. Some studies involve how to find candidate links for requirement tracing, see (Hayes *et al.* 2006). Conte de Leon and Alves-Foss (2006) survey formal and other approaches to component traceability and introduce an own method.

3.2. Defect Prediction

In this subchapter, defect prediction is investigated. The first part surveys risk analysis. In the second part, aspects of process control and defect prediction are discussed. Measures and factors of reliability are involved. The third part involves defect prediction models. The fourth part discusses uncertainties in using defect prediction models.

3.2.1 Risk Analysis

Software projects sometimes fail, and they are often late. Software is often buggy, and bugs may cause a lot of harm. Risks in software development and how to avoid them is a topic for research. Risk analysis helps avoid known and unknown faults and their effects. There are models that aim at early avoidance of risks.

There are several risk *classifications*. Risk classification is usually based on the amount and likelihood of damage. According to IEEE *standard* (IEEE1540 2001) risks should be assessed individually, in combination, and along with their interactions with system and enterprise risks.

Pfleeger surveys risk classification grades, risk *items*, and common *mistakes* like having false precision or using values and use characteristics instead of distributions. One should not rely too much on science; models may be wrong. Confusing facts with values is a common mistake, e.g. one should not conclude the degree on risk solely from what has or has not happened before. A risk may exist even if nothing has happened before. Conversely, the risk can be small even if harm has occurred. (Pfleeger 2000).

A USA military standard (MIL-STD 882B 1984) involves risk analysis, which is performed in several phases of the software life cycle. User's views are often considered in risk analysis. See e.g. (Fields *et al.* 1999) for planning human interfaces in safety critical systems.

Accidents are often analyzed in order to stop repeating the same mistakes and to improve the developing process. A process for choosing the best method to describe an accident has been under research, see, for example (Munson 1999). Using a specific method has also been a common topic for research. Accidents can be described e.g. by cause-effect diagrams like fault-trees or event trees; see e.g. (Mulvihill 1988) for event trees. Research has been done about improving those methods and combining several methods, see e.g. (Xu & Dugan 2004) for combining dynamic fault trees and event trees. Description methods are being extended, see the fault tree column in table 12. Leveson (2004) criticizes event-based methods. According to Leveson, events and initial state are often chosen subjectively and the links between events may be subjective and biased. Usually, no root causes are looked after (*ibid.*). Leveson proposes a system-theoretic model for risk analysis. Table 12 presents that model and some other common methods of risk analysis. It was stated above that accident description methods can be combined. More generally, risk analysis methods can be combined, see e.g. (Tekinerdogan *et al.* 2008).

Table 12. Typical risk analysis methods

Method	**Typical use and research**
A system-theoretic model	A model is introduced for accidents that emphasizes system factors and models accidents as violations of safety constraints. The control system can react to component malfunction, external disturbances, and component interactions. (Leveson 2004).
HAZOP (Hazard and Operativity Analysis)	When using HAZOP, the faults are classified systematically (see subchapter 2.1.2). The leader hypothesizes an abnormal condition and asks questions to experts and determines whether and how the situation is possible and what effects it has on the system. HAZOP emphasizes component interactions. (Reese & Leveson 1997).
Deviation analysis	Reese and Leveson (1997) introduce a deviation analysis method, in which causality diagrams are built and deviation formulas of system variables are incorporated there, and deviations are evaluated with interval mathematics.
A method for analyzing failure mode assumptions and system dependability	Value error, timing error, and unsolicited service are possible. When using this method, failure mode assumptions are formalized as assertions on the types of faults that the component may induce. (Powell 1992).
Fault trees	Liu and McDermid (1996) perform safety analysis by understanding the physical model, building a fault tree, and checking a consistency of the fault tree. A support system is introduced in the study. In addition, fault trees for processes are being developed, see e.g. (Malhart 1995) and (Subramanian *et al.* 1995). Sohn and Seong (2004) assess testability by calculating output failure probability and importance of statements for output failure. Those numbers are calculated from a fault tree. See (Bobbio *et al.* 2003) about parametric fault trees to remove replicas and take only essential information into account. Fault trees that contain stochasticity, effect, and time intervals for events are being developed (Johnson 2003). See (Clarke & McDermid 1993) about weakest preconditions in fault trees where external disturbances are included. Ciarambino *et al.* (2002) use recursive operativity analysis as one HAZOP method, to prevent loops in fault trees. Sullivan *et al.* (2004) reverse-engineer specifications with an abstract fault tree. Coppit *et al.* (2005) study dynamic fault trees with priority and gates and functional dependence. Processing simultaneous events is discussed in the study. See (Ou & Dugan 2000) about sensitivity analysis of modular dynamic fault trees in respect to component failures. The fault trees in the study may contain static and dynamic modules.
FMEA (Fault Mode and Effect Analysis)	This is a method for taking all risk situations into account. In many versions of FMEA, a table is being made about all equivalence classes. SFMEA means software fault and effect analysis. SFMEA pays particular attention to hidden dependencies, e.g. unexpected interactions or unstated assumptions (Lutz & Woodhouse 1996).

Continued on next page

Formal and argument- based methods	Liu *et al.* (1995) survey the use of formal methods. Other safety techniques presented in the study are fault tree analysis, FMEA, failure propagation and transformation, and Toulmin argument form. (Armstrong & Paynter 2006) is one instance of the studies about how to construct and deconstruct arguments and justify them; this study involves safety related arguments.
Methods for requirements, e.g. forward and backward search	Smidts *et al.* (1996) present failure modes for requirements. Feng and Lutz (2005) present a safety analysis method, based on requirements, architecture, and scenarios. This method combines forward and backward search of failures and enables consistency checking between the results. The method finds missing and incorrect requirements. The study investigates safety analysis of product lines.
Other forward and backward search methods	For example, many state based methods include forward and backward searches between risk states and other states, see e.g. (Modugno *et al.* 1997). Tracing makes it possible to do searches between risks and other elements like causes, goals, requirements, and code.
Architecture-based analysis	Goseva-Popstojanova *et al.* (2003) study risk factors for connectors and components in UML analysis. Khajenoori *et al.* (2004) discuss knowledge-centered assessment patterns for safety concerns in software architecture.

3.2.2 Aspects of Process Control and Defect Prediction

Defects can be predicted by using bug knowledge and by using models and metrics. Plenty of research has been done about developing and estimating software quality metrics. Metrics makes statistic process control possible; statistical process control is discussed in subchapter 3.1.1. Many aspects of software quality can be measured, but this thesis emphasizes defect control. The following list contains examples of research involving information system *metrics in general*.

- Ontology of measures, e.g. units and attributes, (de los Angeles Martín & Olsina 2003).
- Axioms in metrics, see e.g. (Le Traon *et al.* 2003).
- Metrics problems; Munoz (1988) involves problems in correctness measurement, and (Xia 2000) is about coupling.
- Empirical evaluation of metrics, e.g. (Le Traon *et al.* 2003).
- Approaches for metrics (Cant *et al.* 1995).

Many studies *compare software to something else*. Among other things, such comparison makes it possible to understand program better or derive its characteristics. Software is sometimes compared to thermodynamic properties. For instance, Kirk and Jenkins (2004) investigate metrics; name-, flow-, and structural obfuscation; compression; and complexity. The concept of entropy is often used in models of quality assessment.

There are lots of source code *measures* like those describing program size, complexity, cohesion, and coupling, see e.g. (Peng & Wallace 1993) for a survey of source code measures and software quality metrics. Size related metrics has been developed, like a number of lines in code or metrics based on number of operations, see e.g. (Peng & Wallace 1993). (Gencel & Dermiros 2008) is a study about functional size measurement. See (Sarkar *et al.* 2007) for modularization metrics. Software complexity is a topic for research, because it has an effect on, among other things, the correctness and maintainability of programs. See (Lew *et al.* 1988) about what complexity consists of. Measures for complexity and for complexity information content are investigated in the study, for example for data structures, modules, and the whole system. There are studies about measuring design cohesion (Bieman & Kang 1998) and code cohesion (Lung *et al.* 2004). Software consistency (Murphy *et al.* 2001) and software process consistency (Krishnan & Kellner 1999) are topics for research. In (Krishnan & Kellner 1999), the measurement of consistency is discussed. According to the study, software defects can be reduced by improving software process consistency.

Zhang and Pham (2000) present 32 software reliability factors and their correlation and investigate ranking. (Wijnstra 2003) is an instance of studies involving *design aspects* like self-test, graceful degradation, error handling, operational aspects, initializations, etc. According to the study, quality attributes can be transferred to design aspects. Also, design metrics has been developed, and Zhao *et al.* (1998) have compared it to code metrics. Statistical analysis of non-stationary metrics is being studied (Pillai & Nair 1997), the article is about process metrics.

Finding relevant *factors for software reliability* is an important topic for research. Number of defects found early correlates with number of defects found later (Jalote *et al.* 2007). A correlation between defect density and project effort is a common topic for research. According to Selby (1990), density of fault detection as a function of testing time depends on the application. Plenty of research has also been done about effect of source code measures on defect density (see subchapter 2.3.1), effort (Itzfeldt 1990), and defect prediction (Itzfeldt 1990). Results about effect of source code metrics on defect density have been contradictive, see e.g. table 7 in subchapter 2.3.1.

Correlation between metrics and defect density is not the only criterion in choosing metrics. For example, some metrics make the defect prediction result more accurate and thus are used regardless of whether they correlate with faults, e.g. old defect densities can be used (Fenton & Neil 1999). There are also defect prediction methods and models that do not use metrics but use only e.g. faults found before, see e.g. (Briand, El Eman, *et al.* 2000) for capture-recapture models. Prasad and McDermid (1999) have developed a multivariable method for assessing whether a computer system works as the customer wishes.

There are studies about comparing *methods to predict metrics*. Gray and MacDonell (1997) discuss and compare prediction methods like least square regression, regression tree, case-based reasoning, neural networks, rule-based systems, fuzzy systems, and classification of decision trees. They also include robust regression analysis: the method to modify the model to correspond the data points. Hybrid neuro-fuzzy systems are also included in the comparison.

Tian *et al.* (1997) survey and present *tools* for capturing, analysis and presentation. Defect tracking, data collection (e.g. for measuring reliability), measurement, and modeling tools are involved. How to choose tools and how to integrate them are also problems discussed in the study.

3.2.3 Defect Prediction Models

Models are being developed for estimating reliability of software that has been tested with a specific method. Lyu and Nikora (1992) survey some models. Models are usually either *steady state models* or *reliability growth* models (Littlewood *et al.* 2001). Steady-state systems containing fixed failure and repair rates for faults can be described by Markov-models, see e.g. (Bukowski & Goble 1995). In reliability growth models, the number of expected faults can be derived from the number of total faults. The number of expected faults convergences towards zero or some other value (Littlewood *et al.* 2001). As an example of a reliability growth model, Goel and Okumoto (1979) present a non-homogenous Poisson process -based model for defect prediction, where failure-rate improves all the time.

Research has been done about this model, see e.g. (Bai *et al.* 2003). Another popular example is Musa's and Okumoto's (1984) logarithmic Poisson model, where earlier fixes cause more failures than later ones.

In addition to steady state and reliability growth models, there are *special use* models. Tian's study (2002) involves data clustering models, and data with different failure intensities. According to the study, these intensities can be used as a piecewise linear model or software reliability growth models can be fitted into these cluster models.

There are *numerous* defect prediction models. Distributions are being developed for faults left or for failure rate. For example, Hou *et al.* (1994) assume hypergeometric distribution for faults left. Failure rate is often assumed to be lognormal, see e.g. (Mullen 1998) for proof for the assumption. Another group of frequently studied models is the family of capture-recapture-models, see e.g. (Briand, El Eman, *et al.* 2000). Some models allow estimation before testing, see e.g. (Graves *et al.* 2000). Change history (Graves *et al.* 2000) and data from similar projects (Xie *et al.* 1999) are sometimes used in predicting defects. Wohlin and Körner (1990) model bug spreading; the model includes probabilities that a bug presented in a certain phase of life cycle is found during a specific phase of life cycle. In (Chillarege *et al.* 1991), defect subgroups with inflected growth curves are related to initialization defects, particularly missing initializations. There are surveys about models, e.g. Peng and Wallace (1993) present some models. Some models can take into account one or more of the issues presented in table 13.

Table 13. Special issues in defect prediction models

Distortions. For example, learning may slow down failure detection, see e.g. (Hou *et al.* 1994).
Differences in failures, corrections, and testing, e.g. different failure rates of faults, different severities of faults, erroneous or imperfect corrections of faults, and effect of testing on changes of failure probabilities (Peng & Wallace 1993).
Delay in repairs. Repairing failures takes time, see (Gokhale *et al.* 1997)
Common mode failures. Faults that have an effect on many components, see (Littlewood *et al.* 2001).
Operational profile when the software is in use, i.e. user inputs, see (Littlewood *et al.* 2001).
Components whose status is unknown. One does not always know if components are working or have failures, see e.g. (Tan 2007).
Several defect reduction cycles, e.g. test and correction, (Rallis & Lansdowne 2001).
Level of correctness, see e.g. (Weiss & Weyuker 1988) about a model containing a tolerance function for correctness.
Concentration of defects, e.g. Ostrand *et al.* (2005) present negative binomial distribution model that allows fault concentration.

Besides probability theories, different kinds of known information and belief are used when *estimating* reliability. Littlewood and Strigini (1993) discuss how to combine belief (based on e.g. design principles, test results, or design process), faults occurred, similarities with known systems, and other such factors when estimating reliability. Because the required level of dependability is high in safety critical systems, not enough can be learned in order to predict their quality well enough *(ibid.*). Pasquini *et al.* (1996) study reliability estimation based on input domain. According to the study, testing may affect neighbouring input values. See (Chang & Jeng 2005) about impartial evaluation in software reliability practice. The study investigates using prior information and prejudgements on software quality in connection with usage testing.

Lo *et al.* (2005) study reliability growth assessment based on individual components and system architecture. The study is an instance of sensitivity analysis studies; sensitivity analysis is investigated for parameters, components, uses, transition probability interactions, and relative error components. Some studies have also been done about calibration and robustness of defect prediction models, see e.g. (Briand, El Eman, *et al.* 2000) for those of capture-recapture-models.

There are several studies about absolute *bounds* or confidence intervals for reliability. For example, Bishop (2002) studies methods to calculate bounds for reliability for a specific execution profile of software statements and for calculating bounds regardless of execution profiles. There are also studies about bounds for faults that a system can tolerate, see e.g. (Santos *et al.* 2005).

3.2.4 Critique of Defect Prediction

Research has been done to compare different fault prediction models, see e.g. (Roper *et al.* 1997). Results of those studies contradict, some examples are mentioned in (Myrtveit *et al.* (2005). In addition, each technique has its relative strengths and weaknesses, and absolute and relative effectiveness depend on the nature of the process and the nature of faults (Roper *et al.* 1997). It is hard to find an appropriate method for a specific system. It is recommended that several models be combined in defect prediction (Neumann 2002). There is a recent tendency to use learning and adaptive models for defect prediction, Kiran and Ravi (2008) have a survey and an experiment. Neural networks are sometimes used in defect prediction, see e.g. (Neumann 2002) about combining neural networks with other methods. Some combined models are being built for defect prediction, see (Fenton & Neil 1999) about Bayesian networks. Menzies *et al.* (2007) study data mining in order to build defect predictors. Many recent defect prediction studies, like this one, make use of Bayesian methods. (Lessmann *et al.* 2008) is about benchmarking classification models for defect prediction.

Many studies criticize software defect prediction models. Models may be based on wrong assumptions. According to Butler and Finelli (1993), it is often assumed that different systems fail independently, although based on former experiments, it should not be assumed.

According to Fenton and Neil, models cannot cope with the unknown relationships between faults and failures; faults or their severities may be hidden, and it cannot always be known which faults lead to failures. The metrics variables cannot always be interpreted in terms of software features. Relationships between variables in models are not always known. Sometimes only a part of the problem is modeled and even that part is misspecified. Casual effects are often omitted. Goldilock's conjuncture of module size is investigated in the study. Also, there are wrong assumptions of the significance of variables, distributions etc, and data points are removed in models. Many models lack the way to integrate views between components structural complexity and within component structural complexity. This integration could explain why different problem or design decomposition approaches might result in more or less defects. (Fenton & Neil 1999).

Uncertainty factor in structural models are that one does not know components or dependability, and parameters are hard to estimate. An uncertainty factor in reliability growth models is that the growth is not always steady even if it is assumed to be steady. In addition, there is general uncertainty: the model and the reality are different. A stress test may help in that respect, and a model should be as continuous as possible. Unknown dependability between components and between consecutive inputs results in uncertainty, particularly if a change of one part may affect other parts. Non-obvious error propagation should be avoided by e.g. isolating the path or by a safety monitor. (Littlewood & Strigini 1993).

Briand *et al.* (1992) present restrictions for modeling software process or its features. Also Kitchenham *et al.* (1994) criticize quantitative quality assurance research. Myrtveit *et al.* (2005) look for reasons why different studies have contradictive results about which fault prediction model is the best. According to the article, many studies are made by using single data sample and samples are often small, different accuracy indicators give different results, and the same accuracy indicator may give different results depending on how it is used. The authors discuss previous results about accuracy indicators not indicating what they should.

According to Dunham and Finelli (1990), models that take into account real-time characteristics are needed. According to Rai *et al.* (1998), not much research is being done about connections between different quality assurance areas. There is no framework for selecting tools and techniques (*ibid.*). Technology changes should have to be taken into account (*ibid.*).

The software development process has a significant effect on failure rate, see e.g. (Dunham & Finelli 1990). According to Dunham and Finelli, the description of e.g. product and process data is important in identifying bugs. Effectiveness and efficiency evaluations for development process and individual tools and techniques are important since development methods contribute to reliability (*ibid.*). Investigation of the fundamentals of failure process is important (*ibid.*).

3.3 Summary of Defect Prevention and Prediction

Research is being done about software processes and their modeling. Some models describe the developing process; waterfall model and spiral model are common examples. Metrics used in improving software development processes is a topic for research.

Errors in development framework may result in software failures. Some studies present errors in theories that are related to software engineering, and there are studies about inconsistencies in models, and about metrics of software process consistency. Obstacles for goals and contradictions between goals may result in software bugs, and they are being studied.

Models of the software development life cycle often contain specification, design, coding, and testing phases. In research material studied for this thesis, issues related to requirement specification, coding, or testing, were more often related to bug detection than issues related to architectural design; architectural design issues were more often related to bug prevention than those involving other phases of life cycle. Some bug-related architectural issues were memory allocation, deciding about module size, and planning good interfaces. The software life cycle also contains setting of goals, maintenance and configuration, and phase reviews. Tracing is a common way to guarantee that requirements of earlier phases are fulfilled in later phases of life cycle. Tracing can also be used within one specific phase.

Many risk prediction methods have been developed for hardware analysis and retailed for software risk analysis. Plenty of research has been done, for example, about modified fault trees and failure mode and effect analysis. Some new methods have been developed for software risk analysis, like deviation analysis. Choosing and using a description method for risks and accidents is a topic for research. Mistakes in risk analysis are also being studied. Risk analysis and static fault analysis can sometimes be connected; for example, risk states can be analyzed in state based static analysis. Risk analysis methods are being improved; for example, more and more attention is paid to root causes of failures.

Plenty of research has been done about quality factors and how to understand and measure software quality. Also, there is research about fundamentals of measuring software attributes, about creating and evaluating measures, and about predicting metrics. There are many quality attributes, and they are being studied. For example, research is being done about complexity measurement and interaction between different factors of complexity. Also, cohesion measurement is being developed. Correlation between different quality attributes is being studied, as well as correlations between quality attributes, defect density, defects found, and development effort. Different methods are being developed for predicting metrics.

Models about the software development process and the software development life cycle were discussed above. Models are used in many other areas of software development, too. Predicting defects with models has been one common research area. Models may use information e.g. about quality attributes and/or faults found. What information should be used in defect prediction is a topic for research. Different applications need different models. Defect curves for different applications have different shapes, and factors affecting the curve shape are being studied. Some models can take into account differences between faults, special issues, and/or different sources of uncertainty. Bounds of reliability are studied, too. Sometimes input domains or data from known systems are being used in defect prediction. There is also a tendency to combine defect prediction models. There is research about mining factors that are important in defect prediction, and about prediction models that are based on artificial intelligence.

Defect prediction decisions and deciding which copies have a correct answer in N-version programming (see subchapter 6.3) may have a lot in common, but I have not found research that compares those problems. More generally, there is not much research about connections between different fields of software engineering. For example, according to Rai *et al.* (1998), not much research is being done about connections between different quality assurance areas, as discussed in subchapter 3.2.4.

Defect prediction models have been critisized. Research has been done about comparing different defect prediction models. Those studies give contradictive results, and researchers have found numerous reasons for these contradictions. Some researchers have mentioned factors for uncertainty, incorrectness, and incompleteness in defect prediction models and given means to mitigate them.

More generally, research models are incomplete and contain uncertainty. One reason for this is that models differ from reality. Defect prediction models have been critisized of being different from reality, but the problem is more general: models are not the same as reality.

There are many other uncertainty factors, too. Research papers often mention lots of uncertainty factors about the study involved. Shull *et al.* (2005) is a study about defect classes, and it is discussed in chapter 2. However, Shull presents lots of uncertainty factors of his study and studies in general.

In presenting uncertainty factors, Shull has an emphasis on uncertainty in post-hoc comparison of studies. For example, mismatches between different studies in comparative research make comparison more uncertain. As will be discussed in chapter 4, according to (Miller 2000), studies about defect detection methods have been found incomparable. Miller states that one reason for this incomparabity is the great variation among those studies; for example, different studies cannot be compared to each other since those studies use different methods. Another problem is that results of studies cannot always be quantified since there are no common definitions that could be used consistently in each study; for example, Miller states that there is no common definition of bug type.

Using results is different from drawing a conclusion. According to (Shull *et al.* 2005), too wide conclusions are often drawn: a study may cover only a small number of projects but conclusion about a large number of projects are drawn anyway.

4 CHECKS DURING AND AFTER DEVELOPMENT

This chapter involves means for checking, i.e. those means to look for faults that are not based on testing. Testing means are based on executing the program and comparing the result (e.g. output values, output frequencies, execution time, or whether the program terminates correctly) to some reference (e.g. to a desired result or a result of execution of another program). Some methods can be considered as checking or testing methods, depending on how they are being performed; examples of those methods are symbolic execution, log file analysis, constraint analysis, and cross-reference analysis. The means investigated in this chapter can be applied during or after development, and many of those means can be applied during both stages.

In chapter 2, features of bugs and particularly those of hidden bugs are surveyed. Chapter 3 discusses bug prediction, and bug prevention with e.g. risk analysis and a disciplined development process. Chapter 5 involves testing. All situations cannot usually be tested, and testing phase is usually not the best phase to eliminate bugs. Static and dynamic checks and proofs for code and documents may reveal faults that testing does not reveal, like faults that cause failures in rare situations or whose appearance is dependent on the test environment.

This chapter involves analysis of specifications, architecture, design, and code. The first subchapter investigates code and data based analysis methods, the second subchapter involves flow or dependence oriented methods, and the third subchapter focuses on methods primarily based on states. The fourth subchapter introduces and classifies logical systems; the application of some of them is introduced. Partiality, iteration, and termination are discussed, too. The fifth subchapter discusses formal methods in software engineering, including prerequisites and limits for proving. Because proving is complex, it is not much used even though it is an efficient way to prevent bugs. So the subchapter also discusses about how proving could be made easier. The last subchapter is a summary about checks during and after development.

Many analysis methods can have either code-, flow- or state oriented view. Type checking is a good example. In addition, type checks can be static or dynamic. Compilers do static analysis; they find many type faults. In dynamic typing, a type of a variable can be changed. In soft typing, the static analyzer generates type checks that are performed during runtime; see (Cartwright & Felleisen 1996). Static analysis of programs with dynamic storage and recursive data structures is being investigated (Landi 1992).

Abstract interpretation is used in many kinds of static analysis. Abstract interpretation of software means taking only some features into account. Accuracy consumes resources, so taking out unnecessary details is often worthwhile. What is needed should be preserved. Cousot and Cousot (1979) have built a general framework for abstract interpretation. Examples of abstract interpretation in the study are approximation of assertions and representing sets of states by lattices. Research is being done about comparing and combining analysis frameworks. The study (Filé *et al.* 1996) is about unification relations between abstract interpretations. Many abstract domains like intervals, octagon, and ellipsoid are used in static analysis, for example in preventing rounding errors and out-of-bound -faults (Blanchet *et al.* 2003). Cousot (1997) analyses types as abstract interpretations.

Software checks may be performed manually or automatically. Some static checkers draw upper limits for space and time usage (Ferdinand *et al.* 2006). Checking and proving can be performed constructively during development to prevent bugs, or developed system can be checked and/or proven retrospectively in order to increase understanding or directly detect bugs. In some dynamic analysis methods, probes can be used to collect information in order to detect bugs (Boujarwah *et al.* 2000). Some checking methods have complete coverage, but checking usually does not reveal external factors.

In checking, one makes assumptions, does a check, and draws a conclusion. An example of a pretty informal check is to perform code review to make sure that all case-statements have default-branches. During that check, one first assumes that if there are default branches, all situations are taken into account, then checks that there are branches and default branches, and draws a conclusion that all situations are taken into account, at the latest during the default-branch. Checking may have several degrees of rigour. Formal proofs are extremely strict checks. They include strict derivation of the conclusion. In proving, one creates a theory with premises and often checks that they hold, does derivation, and draws a conclusion. Logical methods are followed in derivation.

4.1 Document- and Code-Based Analysis

This subchapter describes checks that are based on analysis of software code. The first part discusses some typical methods to find several kinds of bugs by studying the code. The second part describes methods for analyzing values, ranges, and sizes of data elements, and for analyzing precision. Those methods are usually code-based.

4.1.1 Static and Dynamic Code-Based Analysis

Some typical methods that are based on reviewing software code are discussed in this subchapter. There has been some effort to classify those methods. According to Weinberg and Freedman (1984), inspection contains checks for specific issues, review is performed in the course of time, and walkthrough is a posterior design of existing code. Those terms are not used consistently, though, and they have different definitions.

Requirement parsing is a static analysis method for requirements. An unambiguous set of attributes is defined for each requirement. The set can contain e.g. initiator of an action, source, action, object, and constraints. This method may reveal inconsistencies between requirements. (Peng & Wallace 1993).

Comment analysis and making false assumptions are investigated in (Howden 1990). False assumptions are common causes for software faults, e.g. if information that was known was not made part of a program (Howden 1990). Comments may indicate false assumptions about other parts of the program (*ibid.*). Howden discusses comment analysis tools that read user specified facts and assumptions and check that the assumptions are based on facts. Some tools can check code against annotations (Jackson 1995).

Discovering program invariants usually has connections with proving and is discussed in subchapter 4.4.2. However, invariants are sometimes looked after without connections to rigorous proving. Ernst *et al.* (2001) present methods for dynamically discovering likely program invariants from execution traces. Besides more principal computation, pointer processing is discussed in the article.

Algorithmic analysis may contain e.g. re-derivation of equations; verifying numerical techniques; stability, truncating, rounding, and precision analysis; or timing analysis (Peng & Wallace 1993). Algorithmic analysis has strong connection with other analysis techniques. For example, with algorithmic analysis, incompatibility of data representation, e.g. units, incompatibility with hardware or software resources, nontermination of structures, or range faults may be found (Peng & Wallace 1993). For example, a trigonometric routine may only work on the first quadrant in the coordinate system; that fault may be detected with algorithmic analysis (*ibid.*). Memory and time requirements may also be checked; Peng and Wallace (1993) perform it dynamically and call it "Time and size analysis".

Code reading and walkthrough are methods to study algorithm. For example, *code reading* can be used for checking interfaces, comments, and compliance of standards, and for cross-checking of tracing analysis, and for checking that all paths are executed (Peng & Wallace 1993). Peng and Wallace (1993) present a long list about faults that code reading may reveal. Examples are nesting faults, erroneous predicates, missing items, array access faults, sequencing faults, and dead code. Endres (1975) also has lists that contain e.g. initialization and reference faults. In (Shimeall & Leveson 1991), code reading helped detecting calculation faults, missing checks, missing branches, and overrestriction. In *walkthroughs*, logic-, interface-, data-, and syntax faults are found easily (Peng & Wallace 1993).

Software inspection is often defined in standards as a static analysis method that follows a strict process. Fagan (1986) has developed the idea of software inspection. He used a checklist-based approach. There are other techniques, too. For example, Petersson *et al.* (2004) survey capture-recapture methods and discuss other defect prediction methods in inspection, Porter *et al.* (1995) study defect based approach, and Padberg *et al.* (2004) investigate neural methods to estimate defect content from inspection defect data. Laitenberger and DeBaud (2000) survey dimensions of inspection and present taxonomy. There is interaction between fault type classification (see subchapters 2.1.1 and 2.1.2) and inspection. Some studies compare inspection methods to each other, see e.g. (Laitenberger *et al.* 2000) that involves inspection of UML design documents. Comparisons of different inspection parameters are sometimes contradictive, and reasons for it are studied in (Porter *et al.* 1998).

The following list introduces some research areas of software inspection

- Managing inspection, e.g. (Thelin *et al.* 2004) is about performing pre-inspection to decide which software parts need most attention in inspection.
- Modeling inspection, e.g. (Porter *et al.* 1998)
- Assessing inspection parameters, e.g. team size (Bisant & Lyle 1989).
- Inspection methods, e.g. capture-recapture methods (Petersson *et al.* 2004) and defect classification (Kelly & Shepard 2001).
- Kind of software to be inspected , e.g. very large software (Porter *et al.* 1997).

Many studies compare different defect detection methods. Miller (2000) compares defect detection studies by metadata analysis. Defect detection methods involved are code reading, functional testing, and structural testing, although there is uncertainty even in this respect. The studies were not comparable, results differed, and many parameters were different in different studies.

In (Endres 1975), about an equal number of faults of each class were detected by inspection, testing by authors, and testing by other people. Simulation and proving did not reveal all types of faults. Specification and algorithm choice faults had more diversity between number of faults found by testing and inspection than implementation faults. Endres stated that there are numerous possible prevention means, including better understanding, clearer specifications, better specification languages and programming methods, making applicable algorithms available, and structuring programs better. Managing technical reviews is under investigation, e.g. Laitenberger *et al.* (2000) survey management of inspections.

How successful designers investigate the code that they modify is studied in (Robillard *et al.* 2004). According to the study and previous studies discussed in the study, successful designers have a plan for code reading. Also, successful changes are more scattered around the program, whereas unsuccessful changes are more located in one place (*ibid.*).

4.1.2 Values, Sizes, and Precision

Values, value ranges, and sizes of data elements are a factor in may software faults. Methods are being developed for *analyzing the sizes* of data elements. For example, overflows occur when values are too large in respect to the allowed size of the data item. In addition, *order* of computation may have an effect on results of a series of computations with limited sizes or precision, see e.g. (Goldberg 1991). What to do with values outside domains is discussed in subchapter 4.4.2. Kopetz (1975) discusses *input checks* before each function call. He recommends input range tests, too.

Interval arithmetic (Kulisch & Miranker 1981) and other areas of mathematics like convergence analysis are used in analyzing values, sizes, precisions, and ranges of data elements. Interval analysis of maximum and minimum values have been derived for operations for different data sets and types like real numbers and matrices, when maximum and minimum values of operands are known, see (Kulisch & Miranker 1981). There are studies about analyzing and model-checking intervals, like time intervals from one state to another, see e.g. (Hulgaard *et al.* 1995). Also, some systems based on probabilistic logic may process probability ranges of conditional probabilities (Lukasiewicz 2001). Rowe (1988) studied upper and lower bounds for set units and intersections; sets of variables, values, and their frequencies were inspected. The study involved resource usage of database queries, but the results can be utilized in size and range analysis to eliminate software faults.

Dependence between variables is often analyzed with interval arithmetic, see (Kulisch & Miranker 1981). Some other analysis methods like sensitivity analysis and deviation analysis are based on dependencies between variables. *Sensitivity analysis* can be used in size analysis and range analysis, too. Sensitivity means effect of the change or value of one or more variables on values of other variables. According to Bryant (1992), ordered binary decision diagrams of boolean functions can be used in sensitivity analysis of combinatory circuits. Sensitivity analysis can be used more generally, e.g. for intervals of values of variables in software, but no studies were found about the topic. Reese and Leveson (1997) present *deviation analysis* that contains causality diagrams for values of variables in formulas that software contains.

Precision faults are common software bugs. Research is being done about eliminating precision faults and improving accuracy of calculations. There are studies that relate to implementation of floating point systems, see e.g. (Goldberg 1991). Goldberg surveys characteristics of floating point arithmetic and eliminating related faults. According to Goldberg, rounding and truncating cause failures. Dunham (1986) discusses methods to improve precision in comparison, e.g. the use of gradual underflow with guard digits or chopping. According to Kopetz (1975), precision faults may be due to, for example, the fact that the results of the functions (e.g. trigonometric functions) may be too inaccurate for specific acceptable input data range. The *order* of computations also has effect on results. Sometimes when computing rounded or truncated numbers, the order of computation means even when it would not be significant with exact values. This is due to the fact that field axioms that hold for exact numbers do not hold any more when numbers are truncated or rounded (Darcy 2006).

Error accumulation means that imprecision may accumulate if a computation is performed repeatedly, see (Goldberg 1991). One common bug is to assume *convergence* to some point, e.g. zero, when the computation converges to another point or diverges. For example Bastani *et al.* (1988) study convergence problems, connected with proving termination. The article is about fault tolerance in distributed programs.

The precision of calculation is somewhat related to sizes of data elements, too. Interval arithmetic is used in precision analysis, see (Virkkunen 1980). In numerical analysis, precision is often calculated, see e.g. (Tropp *et al.* 2006). Some numerical algorithms detect non-convergence automatically (Troscinski 2003).

4.2 Flow and Dependence Based Checking

This subchapter describes checks that are primarily based on dependence between software artefacts, or on control or data flow of the software. The first part discussed modelling software artefacts and finding bugs by analysing those models. The second part investigates flow based checks. Those flow based checks are often related to dependencies between software artefacts, particularly dependencies between variables in software.

4.2.1 Modeling Software Artifacts

Many kinds of bugs can be prevented and detected by modelling software artefacts. Examples are interface bugs and specifications that have not been satisfied. Modeling software in the system is a topic for research, e.g. UML (unified modeling language) is widely studied; e.g. Jansen and Hermanns (2005) make extensions to UML statecharts and study them empirically. Integration of models is a trend. See (Wand & Weber 1990) for information system models; e.g. information system levels, modeling concepts, internal and external structure, the mapping between external (what) and internal view (how), environment, component decompositions, and system stability are involved in the study. Regnell *et al.* (2000) present model transformation and model expansion as means to integrate use case modeling with usage-based testing. Formalization principles of information systems are being studied, see e.g. (Ter Hofstede & Proper 1998).

There is research about modelling of aspects, see e.g. (Katara & Katz 2003). de Oliveira *et al.* (2004) focus on domain knowledge and define the concept of domain-oriented software development environment. In the article, to understand tasks to be decomposed, the tasks are described verbally, conceptually, and formally. Robillard (2008) studies topological means to look for dependencies. There is also research about concerns. For example, Robillard and Murphy (2007) study how to present concerns with a graph. Marin *et al.* (2007) study identifying cross-cutting concerns.

Many studies involve interaction between software components. Several models and languages have been built for this interaction, see e.g. (Liu *et al.* 2002). They help analyzing correctness and completeness. Examples of inconsistencies are situations where interactions are expected but not found and situations where unexpected interactions occur, see .e.g. (Keck & Kuehn 1998). Hepner *et al.* (2006) analyze conflicts among software components.

There is research about attributes of connectors, see e.g. (Navarro *et al.* 2001) and (Xia 2000). See also (de Lemos 2004), which investigates failure behaviour. There are also role-based approaches for removing unnecessary interaction, see e.g. (Colman & Han 2007). Wahbe *et al.* (1993) study fault isolation when models are coupled. See (Lopes *et al.* 2003) about high-order architectural connectors, which take connectors as parameters. Katz (1993) presents another way to adapt connectors. In the study, semantic abstractions of processes called roletypes are connected with actual parameters.

See (Bellman & Landauer 1995) about wrapping (machine-processable descriptions of resources) and validation and verification. See also (Ceri *et al.* 1988) about designing and prototyping program construction system using relational databases. In this study, relational algebra and interface subschema are used. There is research about automatic task construction from models (Wang & Shin 2006). There are studies about process visualization (program execution visualization), see e.g. (Moher 1988).

Static analysis can be done about software data and interfaces. Charts and models can be built based on the analysis. For example, entity relationship models can be built for static data analysis purposes. Research is being done about evaluating the quality of entity-relationship models, e.g. Markowitz and Shoshani (1989) evaluate the quality of an extended entity-relationship model. There are studies that compare different data models, e.g. Haugen (2005) compares UML and MSC (message sequence chart -model). Data analysis can be performed, e.g. for design (Markowitz & Shoshani 1989), verification (Haugen 2005), or testing (Kansomkeat & Rivepiboon 2003) purposes.

4.2.2 Flow Analysis

Knowledge of control flow and data flow can be utilized in several phases of software life cycle. In addition, the phrase "information flow" has been defined in many ways and used in many different contexts. Peng and Wallace (1993) define information flow analysis as an extension for data flow analysis, where data flows are compared with design intent. Many models of software architecture are based on data flow. In data flow analysis, control flow is analyzed focused on the use of variables (Peng & Wallace 1993). Here are some examples about the use of flow analysis:

- Documenting and understanding software (Moonen 1997).
- Transforming text-format requirements to graphic flows (Peng & Wallace 1993).
- Extracting objects (Guo 2003).
- Testing flow coverage (Frankl & Weyuker 1993c), see also subchapter 5.1.2 about flow based test coverage.
- Detecting flow faults like uninitialized variables, and inconsistencies in write-read-sequences like variables being written but never read before rewrite or end of program (Peng & Wallace 1993).
- Detecting dependency faults (Podgurski & Clarke 1990).
- Assessing effects of variables on a failure (Binkley & Harman 2004).
- Analyzing side effects (Yur *et al.* 1997).
- Detecting type mismatches, e.g. in call chains (Tip & Dinesh 2001).
- Analysis of live variables (Allen & Cocke 1976).
- Detecting dead code (Bergeron *et al.* 2001).
- Detecting buffer overflows, Murata (1989) uses PETRI-nets.
- Some tools detect race conditions (Beckman 2006).

Automatic static analysis utilizes flow, data, and interface analysis (Zheng, Williams, *et al.* 2006). Zheng, Williams, *et al.* (2006) study what kinds of faults can be found by automatic static analysis and by manual inspection. According to the study, a significant amount of critical null pointer faults and several other faults can be detected by automatic static analysis.

Dependence data can be used in deciding strategies to reduce state explosion problem. Zeil *et al.* (1992), and Jeng and Weyuker (1994) have made a simplifying observation: an interpretation of a predicate depends only on paths in front of it.

There are several representations for control and data flow.

> **Graphs** are frequently used in presenting control and data flow and scopes of software variables, see e.g. (Schmidt 1998). For example, program flow can be presented with flow charts and computation trees (*ibid.*). Flow graphs are being extended. For example, Mauborgne (2003) studies extending graphs to present infinity and infinity relations, and infinity representations can be used in flow graphs, too. Every flow graph that can be decomposed has a decomposition tree that describes how the flow graph has been built by composing (sequencing and nesting) other flow graphs, see e.g. (Canfora *et al.* 1998). Lano (1990) presents an N square method for presenting connected functions. Component interconnection is related to data flow, see e.g. (Ural & Yang 1993) for describing interprocedural data flow by directed graphs. Binkley and Harman (2004) investigate bubble and skyline visualizations for dependencies between predicates and formal parameters or global variables. Flow graphs can be model-checked, see e.g. (Schmidt 1998). Forward and backward analysis can be performed, e.g. to find out variables that affect or are affected by a specific variable (Allen & Cocke 1976). Detecting neglected conditions by studying dependence graphs is investigated in (Chang *et al.* 2008). Graph theory can be used in improving efficiency of flow analysis, see e.g. (Hecht & Ullman 1973).
>
> **Algorithms** can e.g. check that variables have been defined when they are used (Moonen 1997), calculate the definitions (of variables) that are valid in a specific program part (Moonen 1997), or they can calculate values for variables (Dor *et al.* 2004). Sometimes algorithms find those network nodes that are needed in flow analysis, e.g. for testing, see e.g. (Hong *et al.* 2003).
>
> **Flow equations and other relations** are also used in describing control and data flow. Inclusion and equality relations are often used, see e.g. (Palsberg 1998). There are studies about the relationship between flow relations and types, e.g. Palsberg investigates the relationship between control flow equations and recursive types. A simple way to describe flow is a decision table, see (Lew 1982). There are algebras for data flow, see e.g. (Fernandes & Desharnais 2007).

Reachability, dependencies between variables, and potential dependencies are essential concepts in flow analysis. There are studies about defining the importance of each node in a flow chart, see e.g. (Hecht & Ullman 1973) and (Kandara 2003). Loops, arrays, nesting, and recursion within or between procedures are challenges in flow analysis. Loop paths have been analyzed, e.g. numbers of iterations have been studied. White and Wiszniewski (1988) investigate recursive modeling of loops and computing loop paths for all such nested and concatenated loops where the number of iterations is known upon entry. Termination issues are also essential in flow analysis, e.g. Hong *et al.* (2003) take the termination or non-termination of paths into account in their analysis. Many studies analyze problems like processing pointers, and finding out where pointers point or may point to, or which element of a composite structure is being processed, see e.g. (Yur *et al.* 1999) and (Amme & Zehendner 1997). (Forgács 1994) involves flow analysis with inter- and intraprocedural recursion. Finding infeasible paths is one problem in flow analysis, see e.g. (Bergeron *et al.* 2001). Flow analysis methods are being developed for special systems like systems containing communication (Boujarwah *et al.* 2000), those with shared variables (Boujarwah *et al.* 2000), concurrent systems (Saleh *et al.* 2001), object oriented systems (Boujarwah *et al.* 2000), and time dependent systems (Bernardeschi *et al.* 1998).

In the beginning, many studies were about systems that had only one entry point and one exit point, but later there have been extensions, see e.g. (Dannenberg & Ernst 1982). Casati *et al.* (2000) study flows for changing and dynamic environments that may also contain temporary starts and stops and exceptional situations. Control flow analysis may have problems in making a difference between duration of a transfer and time to the next transfer; see e.g. (Baresi & Pezze 1998).

Kandara (2003) investigates *paths* in a flow chart. Relationships among paths are studied. For example, Kandara analyzes situations where all paths that go through one node y traverse through some other node x. Some concepts are defined and used in flow analysis in the study, and their application to test coverage is investigated. There are studies that discuss problems with path approach, see e.g. (Howden 1976).

Orso *et al.* (2004) present classifications of data dependencies. Methods for flow analysis are classified, too. Some examples of classifications are:

- Static/dynamic (Boujarwah *et al.* 2000).
- Interprocedural/intraprocedural (Ural & Yang 1993).
- Context-sensitive/context-insensitive, e.g. (Reps 2000).
- Path-sensitive/Path-insensitive (Dor *et al.* 2004).
- Incremental data flow techniques are often classified to elimination algorithms and iterative algorithms, but this classification is rough, e.g. Marlowe and Ryder (1989) present a hybrid strategy.

There are numerous studies about improving precision in flow analysis. For example, algorithms may eliminate unreachable paths, see e.g. (Snelting *et al.* 2006). As another example, methods for gathering alias information are being developed, see e.g. (Yur *et al.* 1999). They are needed particularly in pointer analysis, see (Yur *et al.* 1999). Burke and Ryder (1990) survey means for preventing precision faults in incremental data flow analysis.

The first page of this subchapter contained a list about the use of flow analysis. Here are some more examples about eliminating faults with flow and/or dependence analysis.

- Building path sets which form a minimal spanning set over possible entities in a subset of a graph (Marré & Bertolino 2003). In the study, the set is built for coverage testing.
- Cognitive study on how people search faults by reviewing entity-relationship diagram and data flow diagram (Hungerford *et al.* 2004).
- Algorithm to guarantee initialization (Strom & Yellin 1993).
- Constraints for states; they can be used in pathwise decomposition of a program (Huang 1990).
- (Olender & Osterweil 1992) is about interprocedural static analysis of sequencing constraints; the goal is to detect incorrect sequencing of events. The study involves flow and state analysis automation of constraint specifications.
- Detecting neglected conditions by studying dependence graphs (Chang *et al.* 2008).

The following list contains examples of the (Lacroix 2006) survey about memory protection related static analysis methods. The survey is primarily intended for language features, but those methods can also be used in other kinds of flow analysis.

- Monotonic operations for objects (e.g. variables, stacks, states), lattices of the control flow.
- Constraints on sets of values, and constraint-based subtyping (type1 < type2).
- Concepts of set theory like successor, predecessor, input, and output.
- Constraint propagation.
- Input error propagation.

Plenty of research has been done that relates logics and flow analysis. Table 14 contains some examples of this research.

Table 14. Examples of research that relates logic and flow analysis

Using logics in or with flow analysis, see e.g. (Hong *et al.* 2003) about using temporal logic in choosing test cases based on program flow.
Transformations between graphical presentations of data flow and logic structures, (Hong *et al.* 2003).
Definition of flow analysis. Schmidt (1998) defines data flow analysis as model checking.
Using constructive logic in building flow analysis algorithms and analyzing flow graphs (Lerner *et al.* 2005).
Relationships between flows and set constraints. In set based analysis, dependences between variables are abstracted as sets of values of variables; see e.g. (Heintze & Jaffar 1990).
Counters for variables (Corbett 1993): analyzing data flow near end transitions, constrained expressions about necessary conditions of an end transition are based on data flow, previous guards, and whether transitions occur.
Improving flow graph analysis of concurrent Java programs by supplying additional feasibility constraints (Naumovich *et al.* 1999).
The connection between consequence verification of logic programs and recursively defining flow chart computations forward or backward has been studied by Clark and van Emden (1981).
Predicate abstractions and use of weakest preconditions/strongest postconditions e.g. in looking for reachable states or values of variables (Flanagan & Qadeer 2002).
Reasoning about fault location based on control flow and/or data flow. Le Traon *et al.* (2003) have metrics about location, e.g. the number of tested components in a path is one variable.
Constraint propagation is analyzed in e.g. (Bessiere 2006).
Analyzing which variables share variables, and which variables are bound to other variables. Palsberg (1998) investigates flow analysis in relationships between variables in the abstraction and application operations in lambda calculus**.**

Slicing is a kind of data flow analysis. See (Weiser 1984) about program slicing: expressions or lines that have or may have an effect on values of variables in a specific point of software code, algorithm, or formal presentation, are the only expressions or lines that are investigated when slicing is used. Plenty of research is done about slicing; only some examples are presented in the next two paragraphs.

Cukic (1997) presents vertical slicing (grouping statements that affect a specific output variable) and horizontal slicing (grouping statements that affect specific input variables). Slicing can be static, dynamic, or hybrid (Gupta *et al.* 1997). There are several meanings for the concept of dynamic slicing, see e.g. (Wong *et al.* 2005) for differences between them. Harman *et al.* (2003) study amorphous slicing that does not necessarily preserve syntax but preserves semantic behavior. Slicing can be used, for example, in program understanding (Peng & Wallace 1993), reverse-engineering (Harman *et al.* 2003), building test cases (Hierons *et al.* 2003), debugging (Peng & Walace 1993), and as a checking method (Egyed 2003).

Slicing is often associated with concepts. For example, Tonella (2003) studies using a concept lattice of decomposition slices for program understanding and impact analysis. Gold *et al.* (2005) introduce unification of program slicing and concept assignment. Term rewriting, dependence tracking of variables, and slicing are used in locating type faults; e.g. Tip and Dinesh (2001) implement a prototype. See (Danicic *et al.* 2005) about a tool that can identify and remove a set of statements which cannot be executed when a condition of interest holds at some point in a program. In (Egyed 2003), slicing is used in a tool that detects new traces and conflicts and ambiguities between traces.

Sneak circuit analysis is one way to analyze control and data. According to Hansen (1989), software sneaks related to output are undesired output and undesired inhibit of output. An undesired output by virtue of its timing in relation to mismatched input timing is a timing sneak, and a situation where a program message does not adequately reflect the condition is a message sneak (Hansen 1989).

Desk checking is a dynamic way to analyze code. The programmer moves step by step through the code and tracks values of inputs (Zeil 1999). This term is used in many different ways. For example, (IEEE 1990) uses the words “desk checking” when it means a static review of documents or code. Many online glossaries, e.g. (Farlex 2009), define the term as manual testing of a logic of a program.

4.3 Software States

This subchapter investigates state based checking methods. The first part investigates representations of states, state space exploration, and state based model-checking. Software typically has a very large number of states. The second part involves means to mitigate state space explosion.

4.3.1 State Space Exploration and Representation

Unfortunately, state related bugs like missing states are common. Methods are being developed for exploring the state space of software. For example, D’Amorim *et al.* (2008) have developed an efficient method for exploring state spaces of object based software. More often, state space is explored by building some representations and analyzing them.

Describing software states helps reducing states and taking all states into account. States can be described with graphs and with decision trees, transaction trees (see e.g. (Madria *et al.* 2000)), decision tables, Karnaugh maps (see e.g. (Halder 1982) for large Karnaugh maps), and other charts. State machines are graphs that generally allow nondeterminism but not concurrency. Many extensions to state machines allow concurrency; some extensions to state machines are described in this subchapter. Biswas and Rajaraman (1987) study defining feasibility of decision tables. Moret (1982) surveys decision tables, decision trees, and other decision diagrams. Ordered binary decision diagrams can be used as symbolic representations of state machines (Bryant 1992). Giguette and Hassell (2000) present a relational database model of program states.

State machines help in taking into account all possible software states and all desired transfers. They can be used in describing and planning software function in different states and transfers (Walkinshaw *et al.* 2006). They are widely used in model checking specifications, design, or code. Specifications are often presented in the form of different sequence charts. Statecharts are stategraphs that allow hierarchy, concurrency, and communication of states (Harel 1987). UML sequence charts can be converted to statecharts (Latronico & Koopman 2001). In addition, elements like real numbers can be parsed with a state machine (Baker 1991). State machines are also useful in risk analysis; for example, risk states can be searched forward and backward with state machines and other networks (Modugno *et al.* 1997). Keck and Kuehn (1998) discuss using system knowledge when analyzing problematic states.

State machines of software must usually correspond those of specifications. *Model-checking* can be performed using logical formulas that contain software functions, see e.g. (Alur, McMillan, and Peled 2005). Atlee and Gannon (1993) present a method for state-based model-checking of event-based requirements. According to the authors, event-based requirements may have unexpected or ignored combinations or sequences of events and unexpected timing. Requirements may have logical constraints in the study. In many studies, requirements are transferred to state machines, and specific properties are verified, see e.g. (Nicollin *et al.* 1992). Sreemani and Atlee (1996) study converting tabular state-based requirements to a model that they verify. Heimdahl and Czerny (2000) perform an experiment of analyzing completeness and consistency when verifying state-based requirements. Petrenko *et al.* (2004) study confirming configurations with extended state machines. One problem involved in the study is to determine in a given state an input that does not cause output sequence from a state belonging to a given set or at least its maximum proper subset.

Research is being done about *identifying states* and transitions using state machines (Walkinshaw *et al.* 2006). There are *extensions* for state machines. State machines can, for example, be hierarchic (Kansomkeat & Rivepiboon 2003), stochastic (Chang *et al.* 1998), timed (Nicollin *et al.* 1992), or recursive (Alur, Benedikt, *et al.* 2005). States may overlap (Harel 1987). Alternating automata can have both deterministic and non-deterministic transformations, see e.g. (Kupferman & Vardi 2001). State machines can be combined to make product machines, see e.g. (Kumar & Vemuri 1992). There may be many entry and exit points in some types of state machines (Alur, Benedikt, *et al.* 2005). Return transforms can be used in failure recovery, see (Schultz & Cardenas 1987). See chapter 6 for recovery from failures. Pitt and Shields (2002) discuss the use of local invariants in state machines.

Networks are *classified* according to what conditions they have for nondeterminism and concurrency. Many complex systems can be analyzed with PETRI-nets (German *et al.* 1995). Different kinds of PETRI-nets are being developed. Different PETRI-nets may have different types of distributions for firing times, see e.g. (Trivedi *et al.* 1995). Some methods and tools involve transient states, too (German *et al.* 1995). Gerogiannis *et al.* (1998) classify extension of PETRI-nets into following categories: extension of tokens or places or arcs, modified semantics, extension of structural mechanisms, uncertain fuzzy information, and combining PETRI-nets and other specification methods. PETRI-nets are frequently used in analyzing complex systems. Bucci *et al.* (2004) study analysis of real-time systems with *special properties* like dense time domain and suspensions; extended timed PETRI-nets are used in the analysis.

4.3.2 State Reduction

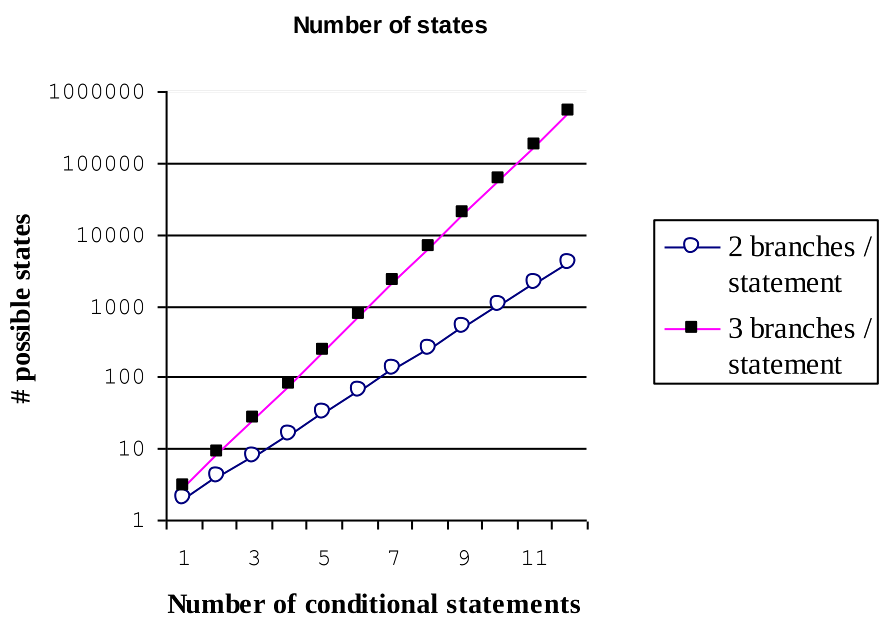


Figure 4. The number of condition combinations in respect to conditions

The term "state explosion problem" means the problem that appears when there are really a lot of states in large programs. Figure 4 presents the number of condition combinations based on the number of conditional expressions and the number of branches in each such expression. It is assumed that each conditional expression has the same number of branches. The vertical scale is logarithmic.

There are surveys about reducing states. Keck and Kuehn (1998) survey interaction of network service features and the state space explosion that follows from it. Keck and Kuehn discuss unexpected combinations. The survey contains a classification of means to fight state space explosion; the classification might also be adapted in other circumstances. Also, many of the methods mentioned in the article can be applied in other contexts, too.

Table 15 presents means to fight state space explosion. Studies are separated from each other by commas, unless stated otherwise.

Table 15. Means for fighting state space explosion

Setting rules as constraints Keck and Kuehn (1998) discuss the following methods: detecting necessary conditions for ambiguity or for reachability, using additional states, creating rules for interaction, using mutual exclusion, establishing priorities, and using heuristic methods. Assertions (Leveson 1991) and serialization (Flanagan & Freund 2004) can also be used in eliminating states. Ip and Dill (1996) investigate state reduction with reversible rules. Cheung and Kramer have developed a method for compositional reachability analysis with context constraints, see (Cheung & Kramer 1994) and later articles of the same authors about improvements of this method. Context constraints are behavior restrictions imposed on each process by its neighbours.
Abstraction and reduction: processing equivalent states as one state Symmetry (Keck & Kuehn 1998), isomorphism symmetry (Sistla *et al.* 2004), processing groups of states as one state (Sistla *et al.* 2000), rule-based selection among equivalent states (Keck & Kuehn 1998), partial order methods (Flanagan & Godefroid 2005), ordered binary decision diagrams (Bryant 1992), symbolic execution (Keck & Kuehn 1998), parameters (Bobbio *et al.* 2003), reduction (Klop 1992), hiding (Keck & Kuehn 1998), counter-example based abstraction refinement (Clarke *et al.* 2003).
Removing states by software architecture and design Modularization and encapsulation (Cukic 1997); using operators that preserve order, e.g. Phillips (1992) discusses monotonicity; placing diagnosis and recovery in ways that reduce state space, e.g. planning the depth of fault tolerant structures (Abbott 1990) and the level of redundancy (Boland & El-Neweihi 1995).
Removing states by using properties of data structures and/or algebras Modularity and compositionality with graphs or algebras by design or refactoring (Cheng *et al.* 2003); heuristic search (Santone 2003); minimalization of representations, e.g. graphs (Pop 2002), (Lee & Yannakakis 1996); minimal coverage (Marré & Bertolino 2003); constructing spanning sets (Marré & Bertolino 2003); contracting formulas (Meyer 2003); derivation of properties of unusual states from properties of algebraic sequences (van der Schoot & Ural 1998); lattice properties and process intervals (Alagar & Venkatesan 2001).
Removing unreachable states by data flow- and/or reachability analysis Reachability analysis (Keck & Kuehn 1998), reachability based behavioral equivalence (Cheung & Kramer 1996), slicing (Danicic *et al.* 2005), eliminating deadlocks and livelocks (Keck & Kuehn 1998), analysis of live variables (Allen & Cocke 1976), counters for variables (Corbett 1993).
Re-using values and/or subpaths, and/or partially processing the system with other methods than state-based ones • Executing simulation in model graph to some point, then generating a graph from there to some until-point (Stuart *et al.* 2001). • Prefix methods: use of a path prefix method where branch coverage is studied by using previous input paths (Prather & Myers 1987). • Re-use of previously selected paths (Chung & Lee 1997). • Combining data flow analysis and state machines (Cheung & Kramer 1994). • Fixpoint calculation (Phillips 1992), (Desharnais *et al.* 2000). • Symbolic representation with Presburger arithmetic formulas and approximation for systems with unbounded integer values (Bultan *et al.* 1999). • Performing an execution trace arbitrarily and dynamically collecting information about thread communication; this trace is analyzed to add backtracking points that identify alternative transactions (Flanagan & Godefroid 2005).

In addition, one efficient way to minimize states is to exclude undesired states. Not much research has been done about this powerful means. Denning (1976) surveys fault tolerant operating systems. Capability is part of the survey: a process can do only what is on a list.

4.4 Different Types of Logical Systems

This subchapter introduces methods for mathematically proving that a piece of software is correct. In the first part, different kinds of logical systems are discussed. The second part involves special topics like partiality, iteration, and termination. The following list contains examples about what proving covers.

- Processing formal models and specifications with logics and algebras, see subchapter 4.5.1.
- Constructing specifications with a formal system (Gerrard *et al.* 1990).
- Proving that the system satisfies specific properties, e.g. (Bravetti 2003) .
- Proving postconditions when specific preconditions hold and after a specific sequence has been executed (Gries 1981).
- Looking for necessary conditions for a problem, e.g. looking for weakest preconditions (Flanagan & Qadeer 2002).
- Looking for preconditions or deriving them from postconditions or other items (Gries 1981).

4.4.1 Logical Systems

Methods for software proving, particularly logic and algebra, are being investigated and developed all the time. There are numerous logical systems. Logical systems are being extended, e.g. van den Brand *et al.* (2003) extend some term rewriting systems. Many surveys have been done that present some logical systems. In addition, logical systems are sometimes compared with each other. For example, Bellini *et al.* (2000) survey some logical systems and compare some of them to each other. Some studies investigate general properties of logic and characteristics of logical systems in general, see e.g. (Morris & Wegbreit 1977) for induction and (Armstrong & Paynter 2006) for argumenting. Very often, the goal of logical studies is to develop an appropriate logical method for a specific application or application domain, see e.g. (Ozsoyoglu & Wang 1989) and (Whang *et al.* 1992) for algebra and languages that can be used in database applications. Filliâtre (2007) investigates total correctness: a formal proof of program is derived in the study with a tool. There are methods for proving total correctness (Babich 1979), (Pettorossi & Proietti 2004). Dawson (2004) formalizes general correctness. Collofello and Vehathiri (2005) discuss measuring correctness.

There is a tendency to build high order systems, where elements of simple systems may be parameters, see e.g. (Young 1997) and (Poigné 1992). *Abstraction* is a trend, based on common features of elements and relationships between elements. Abstraction can be based on, e.g. axioms (Kohlas & Stärk 2007) or category theory (Poigné 1992). Reduction is being studied. In some reduction methods, it is proven that the original system has a specific property if the reduced system has, see e.g. (Lipton 1975). Bobbio *et al.* (2003) study reduction in specific networks, particularly structural deduction.

Table 16 contains some types of logical systems. The same system may belong to several of those types. Studies are separated from each other by semicolons, unless stated otherwise.

Table 16. Logical systems

General philosophies Frege reference and sense, or concept and object (Frege 1892); logical models in argumenting (Chesñevar *et al.* 2000); combining logic and linguistics (Wondergem *et al.* 2001); constructing and deconstructing arguments and performing justification, e.g. (Armstrong & Paynter 2006).
Global theories Model theory (Makowsky 1992); category theory (Poigné 1992); Galois connection, e.g. Dawson (2004) uses Galois connection between weakest liberal precondition and strongest postcondition; domain theory (van Breugel *et al.* 2005).
Logics with different definitions of connectives Differences in connectives like implication, e.g. relevance logic, see e.g. (Goto & Cheng 2006).
Classifications based on truth values Truth-functional logic where the truth value of the compound sentence depends only on the truth values of the individual components (Payne 2005): 2 truth values, e.g. propositional logic (Payne 2005) and predicate logic (Romero 2005), truth-functional logic with >2 truth values (for elements with fuzziness, uncertainty, inconsistency, or undefined items) (Takagi *et al.* 1996), (Baroni *et al.* 2001), (Chung 1989), (Kifer & Lozinskii 1989), continuous truth values (Baroni *et al.* 2001); judgments are more complicated than such sentences, see (Jones 2007) about judgement forms; intuitionistic logic (Ferrari *et al.* 2005), e.g. constructive logic (Akama 1995) and refinement calculi (Yunfeng *et al.* 1999).
Algebra Tucker and Zucker (2002) present universal algebraic specifications. They study e.g. algebraic structures like groups, rings, and fields, algebraic specification of computable functions, abstract algebras, and universal properties common to algebraic systems. See e.g. (Bravetti 2003) about process algebras.
Multiple conclusion logic Logical systems with multiple conclusions, see e.g. (Miller 1994).
Deontic Logics Reasoning about obligation, permission, and prohibition. (Cheng 2006).
Modal logics Modal logics involve necessity and possibility (Bellini *et al.* 2000). Examples are temporal logic (linear or branching time) (Bellini *et al.* 2000). See (Kifer & Lozinskii 1989) about episthemic logic.
Order, amount, and monotonity Non-monotonic logic, e.g. default logics involve beliefs that can be changed (Doyle 1979); global partial order logic (Alur, McMillan, and Peled 2005); declarative partial order programming systems (Parker 1989); interval logic (Ravn *et al.* 1993).
Probabilistic logics Logics that contain probabilities. See e.g. (Lukasiewich 2001) for a brief survey on probabilistic logics and reasoning about systems containing uncertainty, and for probabilistic logic programming with conditional constraints.
Set based Set theories, e.g. (Lubarsky 2006) ; multisets (Frankl & Weyuker 1993c); systems using set operations (Ozsoyoglu & Wang 1989); set based analysis (Heintze & McAllester 1997).

Continued on next page

Type systems E.g. recursive types, subtyping, or polymorphic types; (Palsberg 1998), (Naumov 2006).
Constraints Constraint logic (Dantsin *et al.* 1997); constraint solving, e.g. partial order and lattices (Georget & Codognet 1998), the article is about semiring-valued constraints; Bistarelli *et al.* (1997) introduce a semiring-based framework for solving constraint satisfaction and optimization problems; set-constraints (Dovier *et al.* 2000); non-linear constraints and constraints that define bounds (Hentenryck *et al.* 1998).
Inductive reasoning Morris and Wegbreit (1977) examine subgoal induction. In the study, inclusion and equivalence relationships between computational induction, subgoal induction, inductive assertion, and structural induction are analyzed. Subgoal induction is a going backward-approach and inductive assertion is a going forward- approach (*ibid.*).
Deduction Systems with equivalences (Gries 1981), natural deduction (Maghrabi & Golshani 1992); reduction systems (e.g. rewrite systems including term rewriting and graph rewriting) (Klop 1992); sequent calculi (Maghrabi & Golshani 1992).
Analogical systems for proving termination Fundamentals of computability were studied a lot during 1950's and 1960's, see e.g. (Shepherdson & Sturgis 1963). There are many analogical systems for proving termination. Computability, lambda-calculi, Turing machine and its variations, fixed-point calculus, PETRI-nets, etc. (Potgieter 2006), (Bouziane 1998), (Badendregt 1992), (Phillips 1992).
Combinatory logic Lambda-calculus without abstraction, using different combinators (Meunier *et al.* 2005).
Non-determinism Processing non-determinism by algebras and logics, e.g. (Desharnais *et al.* 2000). (Walicki & Meldal 1997) is a survey about algebraic approaches to non-determinism.
Tabular verification Properties, operators and relations for specifications (Sekerinski 2003); algebraic composition of function tables (von Mohrenschildt 2000); formal semantics for tabular expressions (Janicki & Khedri 2001), expressions are guards or values in the study.
Software-related logical systems Studies about properties of recursive programs and how to prove recursive programs, e.g. (Phillips 1992); object oriented calculi, see e.g. (Yunfeng *et al.* 1999); relation calculus for specific static analysis methods – i.e. for static analysis methods that use over- and underapproximation (Schmidt 2007); logical formulas for state machines, e.g. equations about transitions (Alur, Benedikt, *et al.* 2005).
Properties of integers and counters Proving with properties of counters, e.g. in (Gunter & Peled 2005), based on behavior of program counters, paths are constructed that satisfy constraints for program variables. Siegel and Avrunin (2000) study improving a method of creating and solving integer equations for the existence of an execution trace that violates a specific property - if no solution exists, there are no violations; if a solution exists, there may be violations and some properties can be seen from the equations.

Continued on next page

Languages Algebraic languages for e.g. building models, and model-based languages like Alloy, VDM, and Z (Jackson 2002); processing continuity in programming languages (Gupta *et al.* 1998); formal languages for combining or refining specification descriptions (Feather 1989); use of semantics of languages in proving, e.g. axiomatic semantics (Bonsangue 2001) or denotational semantics (Desnarhais *et al.* 2000); abstract pseudocode (Aho *et al.* 1983).
Examples of abstraction Logical frameworks for several logical systems (Guerrini *et al.* 1997); abstract model theory for specifications and programming (Gougen & Burstall 1992); abstract data types (Aho *et al.* 1983); high order types (Poigné 1992); abstract reduction systems (Klop 1992); axioms in a combined algebraic structure for algebraic specifications, relational database, modules, constraint systems, and other structures satisfying specific conditions (Kohlas & Stärk 2007); relationships about the various conceptions of unification in different fields and what is common for them (Knight 1989).

4.4.2 Specific Issues

Some special topics in proving get a lot of attention by research people. Some important special topics are partiality, iteration, and termination of the algorithm. Those topics are discussed below.

Partiality

Proving something partial is often a topic for research. There are studies about problems involved and means to perform partial proving. Some means to work with partiality are

- Limiting the domain. There are studies about processing partial functions, see e.g. (Parnas 1993) for a simple method and its problems. See (Field *et al.* 1998) for partial equations.
- Supertotal functions. See (Boute 2000) about supertotal functions that are zeros outside their domain and problems with comparing their values.
- Many truth values and operation (Chung 1989).
- Type approaches (Poigné 1992).

Sometimes evaluation is performed partially even for total functions. For example, sometimes only critical portions are investigated with formal methods and their safety and liveness features are proven (Easterbrook & Callahan 1998). Moreover, evaluation is sometimes made for systems when only parts of them have been implemented; see (Avrunin *et al.* 1998). Stubs (Avrunin *et al.* 1998) and modular proving may help in partial implementations. Gannon *et al.* (1987) present the theory of modular proving, where details of modules can be ignored outside the modules.

Iteration

Many logical systems involve nesting and/or infinity. Plenty of research is being done about proving loops. Loops can be proven e.g. with help of rules, deriving hypothesis about loop function and proving loop against it, or by looking for invariants. See (Dunlop & Basili 1982) about proving loops with functional verification (loop functions). Inductive assertion and subgoal induction are also discussed in the study. Using fixed-point arithmetic in analyzing finite and infinite control structures is an important topic for research, see

(Desharnais *et al.* 2000). One research target is undeterministic loops; see (Desharnais *et al.* 2000) about verifying semantics of a candidate abstraction.

Heuristic iterative methods are being developed for proving loops, primarily for looking for linear loop invariants; see (Sankaranarayanan *et al.* 2004) for a brief survey. Invariants can also be detected by constraint-solving. For non-linear invariants, see e.g. (Sankaranarayanan *et al.* 2004); in the study, Gröbner bases and ideals are used for transferring the invariant generation problem to a constraint solving problem. Loop invariants may be generated automatically, see e.g. (Sankaranarayanan *et al.* 2004). Huang (1980) presents a stronger postcondition than loop invariant for proving consistency in loops. Backward subgoal induction is used in proving loops without using loop invariants; see (Morris & Wegbreit 1977).

See (Stavely 1995) about iteration over data structures, when number of items iterated and values to be iterated are fixed on entry. In that study, the data structure is verified against function and task. Matters like access and termination need to be proven only once for a data structure, not for every loop (*ibid.*). Basili and Abd-El-Hafiz (1996) discuss problems with different approaches for documenting loops and looking for invariants. They present a hybrid method where loop is decomposed, knowledge based methods are used in invariant generation, and algorithmic method is used for documentation.

Termination

One research area is computability properties of functions, and whether algorithms terminate and whether they terminate within finite time, see e.g. (Potgieter 2006). See (Hayes 2002) about termination of real-time repetitions. See (Negrini & Sami 1983) for loops and termination. Computability theories with recursive functions, Turing-machines, lambda-calculus, fixed-point operations for certain domains, and PETRI-nets are equivalent methods for proving termination or non-termination of algorithms (Potgieter 2006), (Bouziane 1998), (Badendregt 1992), (Phillips 1992). Bastani *et al.* (1988) study convergence problems in proving termination by analyzing rate of state change.

See (Verbaeten *et al.* 2001) about termination proofs for logic programs – programs in the study contain tables. Decorte *et al.* (1999) study constraint-based termination analysis for logic programs. Pedreschi and Ruggieri (2003) present a framework that always results in successful resolution for logic programs.

4.5 Formal Software Engineering

Formal methods are being applied during all phases of the software life cycle, and in many areas of software development. This subchapter discusses the use of formal methods in different connections. The first part discusses the use of formal methods during different phases of the software life cycle. Some special methods like semantic analysis and use of category theory are also discussed. Some methods like proving by contracts and processing floating point calculations are mentioned, and some application domains are investigated. The second part discusses real-time systems. Analysis tools are being discussed in the third part. The fourth part involves limits for checking methods, particularly for rigorous proving.

4.5.1 Software Development

Formal methods can be applied in all phases of the software life cycle and with numerous areas of software engineering. Both specifications and code can be produced formally, see e.g. (Yunfeng *et al.* 1999). There are many studies about ambiguities in structural analysis of requirements. For example, Baresi and Pezze (1998) discuss imprecision and ambiguity of one structural analysis method and give solutions for them. Writing requirement specifications based on intents and refining them is assumed to reduce faults (Leveson 2000). Some systems can rewrite specifications into logical expressions. Große-Rhode (2002) studies integrating different views of specifications with transformation systems. Testing can be performed formally, too (Gerrard *et al.* 1990).

Model-checking means property verification. For example, it can be model checked that code satisfies specifications or design requirements (Prasad 2006). Research is being done about model-checking by using a specific logical system, e.g. verification tools have been developed, see e.g. (He Jifeng *et al.* 2002). Model-checking in general is also a topic for research; for example, new methods are being developed, see e.g. (Cheung & Kramer 1996).

There are different approaches for program *semantics*, and they often have connections with static analysis or proving. Bonsangue (2001) has done a brief survey about program semantics. In the survey, the connection between different semantics has been investigated. The survey also analyses connection between different kinds of semantics and logic, and between different semantics and mathematical analysis. The monogram also processes connections with semantics and set theory, and with semantics and topology. Some other branches of mathematics and computer science, like domain theory and semantics analysis of types, have connections with program semantics (Fiore 1995). Concurrency, non-determinism, recursion, and timing, and are special features that have been investigated in semantic analysis, see (Bonsangue 2001).

Here are some examples of methods for static analysis. A static analysis method has been developed where program semantics is presented as equations on sets of states, and the least fixed point is solved by forward or backward deduction, see (Cousot & Cousot 1979). Invariant assertions need to be approximated to make the system countable (*ibid.*). The method can be used e.g. for proving postconditions and studying ranges for values of variables, as presented in the study. A systematic design of program *analysis frameworks* is investigated in the study. The method has been developed further in several later publications of the authors. Desharnais *et al.* (2000) abstract the input-output semantics of non-deterministic programs by elements of Kleene algebras. See (Sag & Wasow 1999) about event-based semantic analysis.

Formal systems are being built for *modeling* requirements (Bravetti 2003), and for architectural design (He *et al.* 2004). Logic and algebra are used in modeling, and several problems like some multitasking problems can be prevented with them (Ostroff 1992). See (Taibi & Taibi 2006) about a specification language for *design patterns*. There are logical systems for validating *timing* properties (Bravetti 2003). (Johnson & Malek 1988) is a survey of tools and models for evaluating reliability, availability, safety, and serviceability (in the article, serviceability has to do with the aspects of system design that ease diagnosis and repair).

Executable *specifications* are being developed and debated, see e.g. (Abbott 1990). Specifications are often presented with abstract models or with axioms (Gerhart 1984). Jones (1996) presents types of formal specification languages. Macqueen and Sannella (1985) present completeness results for proof systems for algebraic specifications. Kramer and Cunningham (1979) present the way to use invariants in capturing the behavior of structure and developing formal specifications. See (Sutcliffe & Maiden 1998) about domain theory for requirement engineering; in the study, generic models are used for modeling and critique for new requirements. Trace specifications are based on call sequences (Hoffman & Snodgrass 1988).

There are formal methods for proving consistency and other elements of correctness of specifications, see e.g. (Hoffman & Snodgrass 1988). Specifications can be verified statically (Prasad 2006) or dynamically (Gerrard *et al.* 1990). There are prototype languages, too (Belkhouche & Geraci 1996). Specifications are often built and verified constructively (Gerrard *et al.* 1990), or logical languages may enable definitions of consistency proofs (Hoffman & Snodgrass 1988). If specifications are being refined, decomposition, e.g. constraint-based one, is often done before refinement, and synchronization is often necessary after refinement (Go & Shiratori 1999).

See (Zave & Jackson 1996) about *multiparadigm* specifications. In the study, formal specifications are constructed for systems that map events to commands so that the same event can be used with many commands and vice versa. Consistency analysis for those specifications is studied, too, in the article.

The connection between *category theory* and functional programming is under research, see e.g. (Poigné 1992). Under Curry-Howard isomorphism, a proof of a formula computes a function that witnesses the formula (Makarov (2006) investigates the method); or propositions can be evaluated to types, and connectives can be interpreted as type construction operators (Naumov 2006). According to category theoretic point of view, propositions and types can be objects, and proofs and programs can be morphisms, see e.g. (Blute & Scott 2003). Methods are being developed for applying category theory to specification, design, and maintenance, see (Williamson & Healy 1999). More generally, properties of for example graphs, strings, automates, and partially or totally ordered sets can be investigated with category theory, see e.g. (Poigné 1992).

Hybrid systems contain both discrete and continuous elements (Avrunin *et al.* 1998). There is research about developing and verifying hybrid systems, including sensitivity and reachability analysis (Avrunin *et al.* 1998), (Barton 2000). Wang (2005) studies safety analysis of linear hybrid systems that may contain unbound variables and continuous variables that may change values at different rates.

The following list contains some formal *methods for some areas* in software engineering

- Multilayered approach to design and verification for trustworthy systems (Alves-Foss *et al.* 2004).
- Using category theory in configuring components, Vickers and Hill (2001) have a general approach.
- Bidoit *et al.* (1985) present a specification language and a program construction method; both contain exception handling.
- Proving that one program simulates the other (Birman & Joyner 1976). According to Birman and Joyner, one program could be a specification and another could be an implementation.
- Parametric temporal logic for measuring to what extent a reactive system satisfies a formula (Alur *et al.* 2001).
- Proving where contracts are preconditions, postconditions, or invariants, e.g. (Meyer 2003).
- Automatically looking for program invariants and their violations (Li & Zhou 2005).
- See e.g. (Appel 2001) about proof-carrying code. When proof-carrying code is executed, the proof of the code is checked automatically, and the code is executed only if the proof holds.
- Making specifications consistent with an invariant (Schewe & Thalheim 1999).
- Approximate correction checking with certifier checks (Jin *et al.* 1999). The accuracy is known.
- Formulas for floating point calculations (Virkkunen 1980) and formal systems for integer programming (Sarkar & De Sarkar1989) .
- A composite model-checker for multiple types (Bultan *et al.* 2000). It combines BDD and Presburger arithmetic representation, and can be extendable for other symbolic representations.

- O-Slang combines specification algebra with specification composition via specification-building operations, e.g. aggregation, inheritance, and communication, which are defined with category theory (DeLoach & Hartrum 2000).
- Testing of labelled Markov processes (van Breugel *et al.* 2005); domain theory and coalbegraic extension are used, and using finite branching and similarity in testing is investigated in the study.
- Localizing errors in counterexample traces (Ball *et al.* 2003).

Formal methods have been developed for specific application domains. Table 17 contains some examples.

Table 17. Applying formal methods within specific application domains

Application domain	**Examples of use of appropriate formal systems and issues**
Buffer consistency and overflow elimination	Model checking against file system errors (Yang *et al.* 2004).
Databases	Relational calculi (Ozsoyoglu & Wang 1989), languages and graphs (Beyer *et al.* 2005), and algorithms for deduction of formulas (Yang *et al.* 1989).
User interface	Formal proving (Brestel *et al.* 2005).
Concurrency	A chemical abstract machine (Berry & Boudol 1989); processing non-determinism (Atlee & Gannon 1993); state-based model checking of event-driven formalization (Atlee & Gannon 1993), problems with using logical disjunction in presenting states and transfers in concurrent programs – and solutions for the problems (Atlee & Gannon 1993); finding collections of actions that can be executed without interleaving them with actions of other threads (Flanagan & Freund 2004); nested transaction trees that process possible failures of sub- and supertrasactions (Madria *et al.* 2000). Winskel and Nielsen (1995) have done an overview about models for concurrency.
Communication protocols	Finding paths for desired states (Motteler *et al.* 1995). Motteler *et al.* (1995) present lemmas involving testing sequences. Shiratori *et al.* (1991) have a small survey about protocol verification.
Safety critical systems	Logics and analysis tools (Jacky 1995) , (Gargantini & Morzenti 2001).
Fault-tolerant computing	E.g. clock issues and type checks (Owre *et al.* 1995).

Many methods and tools *combine* proving and testing. For example, formal methods are often used when deriving test cases, see e.g. (Carver 1996), and some formal languages support design-time checks (Gerrard *et al.* 1990). There are methods that combine formal methods with less rigorous analysis like reviews, see e.g. (Traore & Aredo 2004). Some methods combine several formal systems. For example, Beauvais *et al.* (2001) study merging declarative formalism with events and imperative formalism. See (Kurshan *et al.* 2002) about combining software and hardware verification techniques.

4.5.2 Real-Time Systems

There are numerous logical systems for specifying and verifying real-time applications, including logic, algebras, and graphs, see (Falk 2004) for a brief classification. Some of the systems are extensions of non-real-time systems; for example, temporal logics are extensions of modal logics (Bellini *et al.* 2000). Logical systems for real-time systems often involve sequential and parallel processing, time intervals, guarded commands, and/or order constraints even for variables that are arbitrary close to each other (Bellini *et al.* 2000), (Nicollin *et al.* 1992). Some logical systems contain description about what happens when some or all processes terminate, see (Jahanian & Mok 1986). Some logical systems contain advanced temporal requirements; e.g. Ravn *et al.* (1993) present interval and combination related constraints like "at any time interval <= k, duration of (A and B) is < k_2". Many analysis methods calculate bounds for completion times (Ferdinand *et al.* 2006). In a real-time system, the time domain may be dense; e.g. Fränzle (2004) studies using decidable fragments of dense-time duration calculus for model-checking realistic real-time systems. Mok *et al.* (2004) present a specification model where events are instances and composite events and correlation can be specified. Luqi *et al.* (2004) present a document-driven methodology for real-time systems.

Synchronization of parallel actions in real-time systems is being investigated, see e.g. (Bravetti 2003). Problems with verification of embedded software are also discussed, see e.g. (Latronico & Koopman 2001). (Kopetz 2000) is a roadmap for real-time systems. It involves e.g. the composition of components which are as independent as possible, constructive integration, validity and upper limits for worst-case behavior, and implementation of generic fault tolerance. Architectures like smart sensors are also involved in the study. Neumann (1986) presents hierarchical design of computer systems for real-time systems.

4.5.3 Tools

Compilers usually perform different kinds of static analysis. There are other tools for static and dynamic analysis, too. The next paragraph presents some examples of analysis tools, and the following paragraph involves proving tools.

Hiller *et al.* (2002) present an environment for error propagation analysis. See (Trivedi 2002) about a defect prediction tool. Some tools assist in defect tracking and version control, see e.g. (Sanyal *et al.* 1992). Some tools can reason about fault location. For example, Korel (1988) describes a knowledge based tool that can reason about bug location based on program structure, execution traces, and user input. Deeprasertkul *et al.* (2005) present a pre-compile tool for error detection by parsing and pattern matching; the tool can do some automatic correction. Williams and Hollingsworth (2005) study automatic mining of source code based on change history. Dillon and Stirewalt (2003) study customizable and integrable analyzing components that are generated with a tool. Gregoriades and Sutcliffe (2005) present a requirement development tool that indicates problem paths and tasks and their causes (components, input values, and calculation of input values needed for the goal), and supports comparison of alternative requirements and designs.

Many tools can be used for example in property verification (Prasad 2006), proving theorems (Dawson 2004), or type-checking (Tip & Dinesh 2001). Tools are being developed for proving with pre- and postconditions and other assertions, see e.g. (Rosenblum 1995). Assertions may involve dependencies of variables, e.g. dependencies of function arguments on each other (Rosenblum 1995). Some tools detect conflicts between rules, or violations of assertions (Stonebraker *et al.* 1988), (Rosenblum 1995). Many proving tools are customizable; one can even generate custom verification tools with a HOL-tool and other languages (Shepherd 1992). User may define rules, see e.g. (Young 1997), but some tools can derive them. Li and Zhou (2005) present a general method to find implicit rules and detect violations. Schumann (1999) surveys some automatic provers. Young (1997) studies desired features in proving tools and problems in comparing proving tools.

4.5.4 Limits of Analysis and Proving

Many problems related to proving and testing are undecidable. There are undecidability theorems, some essential ones are presented below:

- General theorems by Gödel, Church, Turing etc., see e.g. (Wegner & Goldin 2003) for an overview.
- Rice theorem about undecidability of a non-trivial question, and theories about undecidability of associativity and commutativity analysis. (Charlesworth 2002).
- (Blass & Gurevich 2001): Decidable invariants do not suffice to verify single-loop programs even if pre- and postconditions are decidable.
- (Barber *et al.* 2003) and (Holzmann 1997): It cannot be model-checked that there is no unexpected system behaviour (missing or extra paths or services).
- According to Reps (2000), in general, context-sensitive structure-transmitted data dependence analysis is undecidable.
- Landi (1992) presents undecidability statements for static analysis.

In addition, the following things make rigorous proving to fail:

- Erroneous preconditions (e.g. misunderstandings).
- Invariants do not hold (models and reality are different; models usually have assumptions that do not necessarily hold).
- Erroneous command sequence (i.e. what the system does differs from what is being proven).
- Erroneous derivation (errors in performing the proving, e.g. sequence error in derivation, logic errors, or abstracting out something that would be needed in the proof).
- What is to be proven is outside the scope of the logical system.
- External disturbance (it is actually an instance of being outside the scope of the logical system). In control theory, there are equations for estimating system state when there are unknown inputs and disturbances, see e.g. (Chang *et al.* 1994).

Erroneous derivation usually makes rigorous proving to fail. In constructive development, the development fails if the derivation is erroneous. For example in building specifications constructively, derivation errors result in errors in specifications. Substitution inconsistencies are a topic for research, too, see (Cavalcanti *et al.* 1999). Morris and Bunkenburg (2002) investigate inconsistency in theories of nondeterministic functions, i.e. a flaw in the theory. (Jones *et al.* 1998) is an example of studies involving prerequisites of use of formal methods.

Gerhart and Yelowitz (1976) studied typical faults in proven programs. Logical errors, missing computation (e.g. returning an index instead of a value), and missing final task were frequent faults. Making wrong assumptions was also common, e.g. about the order in which the compiler makes calculations. According to the study, proving was not deep.

It would be desired that the use of formal methods be more common (Feather 1998). The following list presents some remedies that are under investigation:

- Partial application, e.g. proving only safety and liveness properties (Easterbrook & Callahan 1998).
- Lightweight formal methods, e.g. (Feather 1998).
- Automatic proving, e.g. (Schumann 1999).
- Fraser *et al.* (1991) investigate formal and informal specification languages and making formal languages from informal ones.
- Dijkstra (2000) introduces computation calculus for proving formalisms of intended interpretations. From intended operational interpretations, higher level of abstraction can be derived with the calculus.

4.6 Summary of Checks during and after Development

Many analysis methods can have code-, flow-, or state oriented view. Checks can be performed manually or automatically, and they can be performed during or after software development. Checks may be more or less formal, rigorous proving being extremely formal. Analysis, even code based analysis, can be static or dynamic.

Code analysis methods may be related to range, size, or precision of data elements; there is some research about these methods. Those methods sometimes involve dependence between variables. There are simple derivations for maximum and minimum values for some binary operations on different interval domains, but not much research about more complex sensitivity analysis was found. There are more general informal methods, too, like algorithmic analysis, code review, or looking for invariants in the programs. Not much research has been done about these methods. Some comparative studies and few other studies have been found. Those methods are very powerful and general. There are lists of faults that can be found with those methods. Many more types of faults can be found with those methods than what are on the lists. In addition, faults that are often found by flow related methods can also be found with these static methods. General methods like algorithmic analysis and code review can also be used in looking for range, size, and precision problems in addition to special methods like interval analysis.

Instead, a more formal general method, software inspection, gets a lot of attention among researchers. Some comparative studies relate inspection to testing. More such studies should be done since inspection methods and testing methods have a lot in common. For example, choosing checklist items and choosing test cases may have common features. However, choosing test cases is not investigated in inspection-related studies and vice versa. In addition, inspection- related studies do not use results of research in choosing test cases, and vice versa. There is a same kind of relationship between defect prediction in software inspection and properties of curves in general defect prediction models. In some studies mentioned in this thesis, one conclusion is that the field lacks cross-field research. The observations presented in this paragraph support the conclusion.

Uncertainty in research and particularly in comparison of different studies is discussed in the summary of chapter 3. According to Miller (2000), results of studies cannot always be quantified due to the lack of common definitions that could be used consistently in each study. For example, there is no common definition of bug type (*ibid.*). Another terminological problem is that some terms have definitions related to information technology, and those definitions may differ from either general definitions or definitions of the same terms in other fields. Some of those terms are included in IT standards. The word “inspection” has a general meaning, but in software engineering, the term has got a special meaning. In glossaries, software inspection is usually defined as a strict process with certain meetings, specific roles, and specific organization.

Some terms like “desk checking” have been defined in different ways in different documents. Some terms like "inspection" and "walkthrough" are used inconsistently. As stated above, the term “software inspection” has been defined in glossaries as a strict process. In some papers, the word often means methods like code review, code walkthrough, or algorithm analysis, which all are usually less formal methods. The definitions for those less formal methods are used inconsistently, too.

Some terms have a general definition but are unnecessarily redefined in some studies. Those redefinitions add part of the context of those studies or that of their domain to the general definition without stating it explicitly. For example, the definition of term “confluence” in a study about automatic generation of loop invariants using Gröbener basis (Sankaranarayanan *et al.* 2004)[3] includes local context. With reduction systems, the general definition of confluence only guarantees that different reductions from the same arbitrary object eventually lead to a common object (Klop 1992). In term reduction, it means that different reductions from the same arbitrary term eventually lead to a common term. The common term need not be a normal form[4] or lead to a normal form: there can be infinite reductions like cycles. Hence, the general definition only guarantees that if there is a normal form, equivalent normal forms are unique (Klop 1992); it does not guarantee that there is a normal form. In the study, in the narrow contexts where the reduction could be shown to be terminating, confluence of a reduction was defined as follows: “every term reduces to a unique normal form”. This is a definition in local context - in the context where all reductions terminate. Thus it is not a general definition and presented as such, may confuse those who have been using the broad definition.

There are many forms of representations for software and its environment, component interaction, data and its relationships, and for control and data flow. More and more research is being done about aspects and concerns. Graph theory is often applied when studying representations. There are analysis methods that use representations for finding bugs. Integration and paying attention to inconsistencies are trends in research involving models. Like there is research about understanding root causes of bugs and application domain, there is research about understanding the domain of models.

Flow analysis often involves dependences between variables. Flow analysis helps understand program and reveals several faults like type and dependency faults. Plenty of research has been done about making flow analysis easier and more precise. There is research involving path related problems and problems and challenges in flow analysis. Some areas of mathematics are sometimes used in flow based analysis. Some studies connect flow analysis and logic. How to detect bugs with flow and dependence methods is a topic for research. Flow analysis is sometimes combined with other methods to detect bugs. There are also studies about error propagation to output and constraint propagation.

As stated in the summary of chapter 2, different methods have different fault prone features. For example, graphs do not always express timing or scope of variables, as discussed in subchapter 2.2.2, see (Yoo & Seong 2002). A related problem in control flow was found in (Baresi & Pezze 1998): control flow analysis may have problems in noting difference between the duration of the transform and a time to the next transform.

State space exploration and state space representation methods are being developed. Software states are often presented with tables, trees, graphs, and networks, and those presentations are being extended. Identifying states in state machines is being studied. Missing states are common causes for software failures. State machines are often used for model checking or risk analysis.

The state space explosion makes state-related faults more common and model checking harder. A lot of research has been done about how to relief the state space explosion, and several relief methods have been developed. One means is reduction of graphs. There are studies involving general reduction of graphs and trees. Some of those studies are about state space explosion, and some are more general. Results of those more general studies can be used to mitigate state space explosion. States can also be reduced from other representations like tables.

[3] The study is discussed in subchapter 4.4.2.

[4] Normal form means a terminal: no more reductions can be performed.

Lightweight methods could be used in fighting state space explosion, but not enough attention has been paid to these methods. Examples of lightweight methods are modularity and easy elimination of unnecessary states. Examples of the latter are easy minimalization of graphs, and making the system go to error state if something unexpected happens.

Excluding and logging undesired states is a good means to fight explosion, but it has not got much attention. One related study was found about processes in an operating system. Minimalization could be performed by setting priorities, too. For example, erroneous input is usually rejected instead of being processed. For example, new input could be asked for if the piece of software is interactive and the user enters inappropriate input. If this is the case, erroneous input need not be partitioned according to values of other variables if partition testing is used. Sometimes it is hard to know if one wants the system to process unknown states. If they are excluded, they can be put into a log file. This way, the developers get to know that such states exist, and they may include the states if they wish.

Many studies of state space explosion involve telecommunication systems. However, state space explosion problems are present in numerous other application domains, too, and in systems where there is no concurrency. The results of the studies about state space explosion in telecommunication systems should be used when studying other application domains, too. However, people do not find those results when they could need them. As another example of using results of studies in other fields, results of studies for developing compilers are often related to software control and data flow. Results of these studies could be used in improving fault analysis methods that are flow-based.

There are numerous logical systems, and they are being extended. Both logic in general and special logical systems and application domains are being studied. Some fundamental theories about recursion and computability have been developed in 1960's. Also, partiality, iteration, and termination have been topics for research. Abstraction of logical systems is a trend. There are some studies about connections between logical methods and developing methodologies, e.g. between category theory and functional programming.

Formal analysis and proving methods can be applied in all phases of the software life cycle. Different kinds of logical systems, axioms, models, and languages can be used in proving. Some common topics of research are looking for invariants to capture the behaviour, methods for model-checking, different approaches of program semantics, and constructive development. Some formal methods and systems have been developed for some areas in software engineering, e.g. for making specifications consistent with an invariant, performing floating point calculations, or proving that one program simulates another. There is a tendency to integrate methods. Some tools can do automatic proving or reasoning about contradictions and failures. Some tools can correct faults.

Not much attention has been paid to prerequisites and limits of a logical method. Those limits are reasons why formal methods are not frequently applied. The research is more practically oriented: making formal methods easier to apply with means like lightweight or customizable methods and automatic proving are under research. There is a slight tendency for early validation and verification of software, but it should be stronger.

Some research is being done about how to make people apply formal proving more often. The question should be more general since checks are an efficient way to prevent faults and should get much more attention. Different kinds of checks have been developed, but checks have not been used very often, nor do they get much attention by research people, compared to testing. However, checking is very efficient method to reveal faults. For example, many checking methods have complete coverage, including unpredicted and rare situations, and unpredicted facts and fault types may be detected by checking.

5 TESTING

Testing is a very common means to detect software faults. This chapter discusses software testing as a means to detect defects. Testing may also reveal external factors that have an effect on software reliability, but testing does not reveal all external failures. Software may be tested for other reasons, too. For example, efficiency or user-friendliness can be tested. Those tests are outside of the scope of this thesis.

Plenty of research has been done about methods for choosing test cases, test coverage, and testing methods. Due to the large amount of methods and criteria and research involving them, it has been impossible to discuss them in detail. This chapter contains only collections of existing methods for choosing test cases and performing the testing, and collections of coverage criteria. The chapter also contains outlines of assessing testing methods.

The first subchapter of this chapter involves choosing test cases and estimating test coverage. The second subchapter presents different test methods and their classifications. Estimation of testing and some problems related to testing are also discussed, and testing tools are presented. The summary follows as the last subchapter.

5.1 What to Test

In this subchapter, the problem about choosing test cases is discussed. The first part involves examples about what to test. The second part investigates different types of test coverage, and methods for assessing coverage.

5.1.1 Items to be Tested

Table 18 contains examples about items to be tested. The first part presents common items and the second part introduces method-specific items.

Table 18. Items to be tested

General Testing
Risks Cases that are most prone to cause management problems or to cause timing problems (Kaner 2004), known faults (Kaner 2004), most frequent faults (Amland 2000), or faults that cause most damage (Amland 2000), and erroneous inputs like wrong values (Kaner 2004) or wrong command sequences (Leveson 1995).
Special items Missing element; missing function; blank; zero; leading zero; one; small absolute value; empty; special character like quote; first; last; border; close to border; output on border; discontinuity of a piecewise continuous function; discontinuity of a derivate; points where function stops being e.g. constant, monotonic, or linear; and items outside domains. (Redwine 1983), (Howden 1986), (Ostrand & Balcer 1988), (Wooff *et al.* 2002), (Clermont & Parnas 2005), (Goodenough & Gerhart 1975).
Hierarchy Howden (1986) proposes the following tests for arrays, but they could be executed for other kinds of collectors, too: zero value, zero row, zero column, and a case where some values are special and some are not.

Continued on next page

Predicates Arithmetic expressions, arithmetic relations with arguments in all orders, conditional commands, iteration commands, operation sequences, off by ones in relational expressions, smallest possible increments and decrements, sign and value, value and sign should not be the same in every test case, value and sign should not be the same as any other value or sign in all test cases, and in arithmetic operations variable should have a measurable effect on value and sign of the result; all those are in (Foster 1980).
Functions If input variables are names of subroutines, each type of subroutine should be tested (Howden 1980). If the number of appropriate steps is another input variable, some values for which the function converges should be tested, and some values for which the function does not converge should be tested (*ibid.*). Extreme values for calls and returns should be tested (Foster 1980).
Code Some command sequences and operation sequences (Foster 1980), loading (Krishnamurthy *et al.* 2006), relationships between variables (Foster 1980), maximum, minimum, and intermediate values (Howden 1980), combinations of maximum and minimum values (Howden 1980), maximum and minimum values in an expression (Howden 1980), identical and distinct values (Howden 1980), cases where all items have the same value (Howden 1980), cases where output values differ from input values for variables used in both input and output (Howden 1980), output out of range (Howden 1980), same paths with several values (Kaner 2004), repetition of a test case (Kaner 2004), all permutations for specific sets of values for some variables (Grindal *et al.* 2004), different sizes of data structures (Marinov *et al.* 2003), member of a family of related classes (Weyuker & Ostrand 1980), and paths with the highest number of conditions and operations affecting the result (Foster 1980).
Special situations File not open, read past end of file, overflow, and underflow are examples of error situations in (Westerfield 1992). Those situations could be tested. Extra item (Smidts *et al.* 2002) and wrong type (Spohrer & Soloway 1986a), (Sullivan & Chillarege 1991) are special situations, too. There are some new studies about testing of input validation; for example, Liu *et al.* (2009) have a path based approach.
Method or application specific items
Interaction The following items should be tested: possible interactions of components (Howden 1986); messages (Briand *et al.* 2003); parameters of functional unit, characteristics of each parameter, objects in the environments whose state could affect the functional unit's operation, and characteristics of each environmental object (Ostrand & Balcer 1988). In object oriented testing, objects, interfaces, and pure, inherited, and overridden methods should be tested (Alkadi & Alkadi 2001).
Mutation testing Mutant operations have been defined for expressions. Absolute value of a variable, the negation of the absolute value, and forcing x to zero are examples of mutants (Wong & Mathur 1995). Effect of omitting some mutants (selective determination) has been studied (Offutt *et al.* 1996). There are many other types of mutants; e.g. Emboss msbar generates same types of mutants for strings that are generated for DNA sequences in real life, see e.g. (Royce & Necaise 2003).

Continued on next page

Domain testing A lot of attention is paid to border shifts, coincidental correctness, and missing expressions, see e.g. (Clarke *et al.* 1982). Some errors related to boundaries are shifted bounds, tilted bounds, missing bounds, extra bounds, and closure faults (Zhang & Harris 2000). Stress-testing is related to testing boundaries for input, output, and loading, and values close to boundaries, see table 21. Research has been done about how to choose testpoints for different types of domains, e.g. domains with several inequations, those with linear and different kinds of non-linear equations, discrete and continuous domains, and situations where points exactly on the border cannot be tested, see (Jeng & Forgács 1999) for discussion about all those problems. Some studies involve choosing testpoints for given paths, e.g. (Jeng & Forgács 1999). White and Wiszniewski (1988) present the number of test points needed for loop patterns. Hierons (2006) studies finding test cases to avoid coincidental correctness in boundary value analysis.
Operational profile See (Chen & Yu 2001) on sampling strategies where test cases are allocated approximately in proportion to the size of subdomains; Ntafos (2001) puts it more exactly: probabilities are used instead of sizes because each input is not always equally likely to occur. Those sampling studies involve partition testing but the methods could be applied elsewhere, too. Naixin and Malaiya (1994) examine how operational profile should be considered in testing when failure probabilities and the number of test cases are known.

Research has been done about making *assumptions* in testing and building test cases. For example, some methods for selective regression testing assume that if a system is faulty, all faults are found by testing specific scope (Leung 1995). Test cases can be created *formally*. For example, Zhu (2003) analyses replacement systems in algebras that can be used in deriving test cases.

Test cases are often collected from some source. Sometimes a source is converted to another source for test case derivation, see e.g. (Kansomkeat & Rivepiboon 2003). Seeding faults to the source of test cases is another way to build test cases, see e.g. (Fu *et al.* 2005). Some studies compare sources for test cases, e.g. Zhu *et al.* (2002) compare different graph representations. Table 19 contains some sources for test cases.

Table 19. Sources for test cases

Textual scenarios. Textual descriptions can e.g. be structured and converted to state machines (Glinz 2000).
High level specifications. They can be of any format. Some examples are formal language, algebra, formal model (Zhu *et al.* 2002). Test cases can be derived e.g. from algebraic specifications by replacement e.g. (Zhu 2003).
Source code. See e.g. (Hierons *et al.* 2003) for conditional slicing.
UML diagrams. See e.g. (Kansomkeat & Rivepiboon 2003) for generating test cases from UML statecharts.
Models. E.g. Podgurski and Clarke (1990) present semantic models about faults and software behavior.
Graph representations. There are different types of graph representations. Methods for building minimum spanning trees and other minimum coverage selection methods can be used in path selection, see e.g. (Marré & Bertolino 2003). Flow graph is commonly used in looking for test cases, see e.g. (Hong *et al.* 2003). Gabow *et al.* (1976) reduce two problems to graph-theoretic problems. One is testing a specific set of statements. The other is finding a path which satisfies impossible path.

Continued on next page

Cause-effect-graphs. Tai *et al.* (1993) investigate fault-based testing of cause-effect-graphs; relation and boolean faults are looked after in the study.
Classification trees. Chen *et al.* (1999) propose improvements for classification trees.
Class vectors. Leung *et al.* (2003) present a method for generating test cases from class vectors.
Risk or fault regions. Risk or fault regions of the software can be used as sources for test cases, see e.g. (Amland 2000).
Information about similar systems. E.g. known problems can be used in building test cases, see subchapters 2.2 and 2.3.1. Performance of other systems, and performance testing benchmarks can be used in test case construction, see (Carrington *et al.* 2005) about assessing performance benchmarks.
Previous succeeded and failed test cases. E.g. old test cases can be used in regression testing, see (Rothermel *et al.* 2004).
Software behaviour. Bowring *et al.* (2004) study learning about software behaviour and clustering it into classes. According to the article, the information can be used in making plans about test cases.
Sampled user executions. User executions can be sampled for bug isolation (Liblit *et al.* 2003).
Constraints. Carver's study (1996) involves constraints that have been derived from an abstract program; besides testing, constraints can also be checked against abstract program (*ibid.*).
Properties of functions, or relations between functions. For example, CAVEAT tool deduces properties of functions and graphs and information about relationships between functions (CEA LIST 2004).
Other relations. Relations between conditional expressions are used in test case selection, see e.g. (Chen *et al.* 2003).
Algebraic specifications. Zhu *et al.* (2002) investigate choosing test cases from formal specifications, including algebraic specifications.
Operation sequences in a graph. Probabilities and probability distributions of operation sequences can be used as sources for test cases (Chang *et al.* 1998).
Any sources for known antigoals and antirequirements. See (van Lamsweerde 2004) about antigoals and antirequirements.
Input distribution. Input distribution is used as a basis for selecting acceptance testing cases (Kopetz 1975).
Domain features. Sinha and Smidts (2006) present a technique for taking domain features into account in testing. The study presents, e.g., types of system invariants and involves language features.
No source. Random data. See e.g. (Duran & Ntafos 1984) about random testing.

Howden (1986) introduces input-output, trace, and transform oracles for specification testing. See (Freedman 1991) about component testing, and how to make software observable and controllable. Observability means how input affects output. Controllability means how easily specific output follows from input.

Many failures are related to *rare events*. It is not likely that a specific rare event occurs in a random test set, nor have developers always taken rare events into account. Voas *et al.* (1996) do experiments and develop algorithms for making it more likely that rare events are selected for test cases in failure-tolerance testing. See (Haraszti & Townsend 1999) about rare-event simulation for software that has high complexity.

Software is *not the only thing* that should be tested. When software is run, external factors like the operating system, API, and other files, affect memory, heap, file handles, etc. The result of those other activities has an effect on software function, particularly on software input. Files may have been corrupted, and data may be erroneous. Other users may modify the data in files. Also bad user input causes errors in programs. Errors with external connections may be hard to reproduce. (Whittaker 2001).

5.1.2 Coverage of Testing

Coverage of test is one factor in estimating if sufficient amount of testing has been done. See also subchapter 2.3.3. Zhu *et al.* (1997) have a survey about test coverage.

Coverage criteria can be derived from several sources, e.g. from code, specification, or constraints to be satisfied. Frankl and Weyuker (1993c) survey data flow coverage methods, and Grindal *et al.* (2004) survey combinatorial methods. Table 20 presents some coverage criteria.

Table 20. Coverage criteria

Combinatory. All items, all n-way combinations of items, all values of some items, k-boundary, and k-perimeter are coverage criteria mentioned in (Grindal *et al.* 2004).
Value based. The tested item is a value for a variable (Grindal *et al.* 2004). Examples of using value based criteria are distinguishing each expression from all its strict subexpressions (Weiser *et al.* 1985), and testing each subexpresion with at least two values (*ibid.*).
Control based. The tested item can be e.g. node (Kansomkeat & Rivepiboon 2003) or entry (Frankl & Weyuker 1993c). Pohjolainen (2002) mentions e.g. branch, elemental decision, and all ways to execute a conditional decision as coverage criteria. Pohjolainen also mentions statement coverage, i.e. that all statements are executed. Tai (1993) mentions a method where all simple and compound subexpressions are tested with both truth values in a compound expression.
Control and value based. For example, all combinations of values of conditions are tested (Frankl & Weyuker 1993c).
Fixing some items and testing others. Grindal *et al.* (2004) mention strategies where some parameters contain default values and other parameters are tested.
Combined method. This method requires that every possible branch be tested at least once for each special value for each input and output variable (Howden 1986).
Orthogonal arrays and covering arrays. See (Grindal *et al.* 2004) and (Cohen *et al.* 1994) for orthogonal arrays, and e.g. (Grindal *et al.* 2004) for covering arrays. Yilmaz *et al.* (2006) present methods that use coverage arrays for testing different configurations and finding out features of the configuration subspaces in which bugs manifest.
States and transfers. Each state and/or transfer is tested, see e.g. (Fujiwara *et al.* 1991).
Flow criteria. Data flow coverage criteria involve e.g. definitions, uses, and/or paths from some or all definitions to some or all uses (Frankl & Weyuker 1988), (Frankl & Weyuker 1993c). k-tuple criteria involve definition-use-paths between a specific definition and use (Frankl & Weyuker 1993c). Further, uses can be separated to predicate- and computation uses (Frankl & Weyuker 1993c). See also (Jeng & Weyuker 1994) for more criteria. Hong *et al.* (2003) survey data flow coverage criteria. Podgurski and Clarke (1990) analyze dependence coverage.
Context. Context coverage involves paths that define variables used in a specific statement in (Frankl & Weyuker 1993c); Information Processing Ltd (IPL) (1999) investigates object specific context coverage (e.g. inheritance context decision coverage, or interaction coverage of inherited methods), user specific context coverage (e.g. thread coverage), and state based context coverage.

Continued on next page

Ordered context. Ordered context method requires that each ordered context be exercised by a path that visits the definitions in the given order (Frankl & Weyuker 1993c).
Orders. Orders of ordered data elements can be tested (Ntafos 1988).
Iteration. Paths that iterate loop k or less times and paths that fail the test on loop boundary can be tested (Ntafos 1988).
Mutation. Mutations can be tested (Frankl & Weyuker 1993c).
Specification-mutation. This is a coverage measure for mutants in a specification (Abdurazik *et al.* 2000).
Operational profile. Operational profiles for input can be tested (Chen 1998).
Couplings and Interfaces. Offutt *et al.* (2000) study coupling based covering in integration testing. In the study, similar measures are used for e.g. subroutine calls and use of external devices that are used when measuring data flow coverage.
Compound. E.g. Grindal *et al.* (2004) discuss compound strategies.
Fault tolerance coverage. Error and fault handling coverage and fault assumption coverage are presented in (Avižienis *et al.* 2004). According to the article, failure assumption coverage can be either failure mode coverage or failure independence coverage.

Many studies compare test *coverage criteria*, assess specific criteria (Abdurazik *et al.* 2000), or investigate methods for assessment. Relationships between coverage criteria are a common topic for research. For example, some criteria may subsume other criteria, see e.g. (Grindal *et al.* 2004). Feasibility affects the relationships (Frankl & Weyuker 1988). Frankl and Weyuker (1993b) discuss why subsumption relation does not always mean that errors are more likely detected. Frankl and Weyuker (2000) use failure regions and multisets of input subdomain for building relationships between coverage criteria.

Links between coverage criteria and properties of the program under test, e.g. number of bugs, are under research (Garg 1994). Plenty of research has been done about test *adequacy properties* and *relationships* of those properties. Weyuker (1988) presents axioms about adequacy properties. Parrish and Zweben (1991) have an axiomatic perspective for test data adequacy criteria. The study examines relationships between adequacy properties for different adequacy criteria and for different assumptions.

Chen *et al.* (2003) study relations among *choices* of *test cases*, e.g. embedding, and relations among test cases. Checking consistency among test case *constraints* and representing different types of those constraints is also studied in the article. Coverage *metrics* and coverage measurement tools have been developed (Pohjolainen 2002). There are studies about comparing test case allocation measures, e.g. (Leung *et al.* 2000). There are *fault class* hierarchies, see e.g. (Lau & Yu 2005) and (Okun *et al.* 2004) about fault class hierarchies in expressions. In those studies, hierarchies are presented where test cases that detect all faults in a larger fault class detect all faults in smaller class. Okun *et al.* (2004) also compare classes of logical faults in specification based testing and calculate relationships between those classes. Sizes of test input sites of some combination testing strategies are presented in (Grindal *et al.* 2004).

According to Roper (1999), the link between adequacy criteria and attributes of the program under test is missing. Also, there is no link between two criteria unless test cases for the more modest criterion are a subset of those of the more demanding criterion (*ibid.*). Frankl and Weyuker (2000) shortly survey research containing critique of relationships between coverage criteria. Hierons (2002) studies comparing test sets and test criteria by using test hypotheses and fault domains.

Algorithms and heuristic methods are often used in test case selection and *reduction*. Chang *et al.* (1991) present a strategy that uses heuristic rules to achieve branch coverage. Jeffrey and Gupta (2007) use coverage information in their heuristic method for reducing test suite.

There is research about reducing test cases by graph theoretic methods. Some examples are using previous paths (Prather & Myers 1987), looking for minimum coverage (Marré & Bertolino 2003), or algorithms for constructing minimum spanning set (Marré & Bertolino 2003).

There are theories about if there are test sets with which testing is certainly *adequate*. Adequancy problems with finite and infinite test sets, tolerance functions, etc. are under research, see e.g. (Li *et al.* 2004), (Garg 1994). How many test points define a subdomain is an important question, and it can be calculated, see (Zhu *et al.* 1997) for functional testing. According to Zeil and White (1981), number of test cases needed in path testing depends on the number of input variables and program variables. Cain and Park (1996) derive the number of necessary test points for finite domain vector spaces in testing the equality of functions. Chusho (1987) studies how to eliminate *redundant* test cases in respect to a coverage criterion, e.g. how to avoid re-testing a branch that has already been covered by another test case.

See (Dalal & McIntosh 1994) about *stopping criteria*. Large software and changing code are also involved in that article. There are many stopping criteria based on the probability of finding more faults. E.g. Littlewood and Wright (1997) present methods based on faults found during the testing period. The authors propose that the testing may be continued even if faults have been found and the test will eventually fail. According to Littlewood *et al.* (2001), the test can be deterministic so that it is known that the software certainly fails. Costs of failures are often taken into account when analyzing coverage and stopping criteria, see e.g. (Amland 2000).

Miller *et al.* (1992) study the problem about how to estimate the probability of faults if testing reveals no failures. A method is introduced that is based on prior information, and on assumptions about the operational profile. The situations are also covered where the assumptions about operational profile change. According to Butler and Finelli (1993), estimating the reliability of life-critical software requires so many test cases that it is impossible, regardless of whether the software is standard or fault-tolerant, and whether blackbox (input to output) or reliability growth models are used. See (Littlewood & Wright 1997) about stopping rules for operational testing of safety critical software, both discrete and continuous systems are inspected. The authors think that pessimistic rules are good, and Bayesian models (statistical models using a priori information) should be used. Also, one should stop after finding a fault (*ibid.*).

5.2 Test Execution and Evaluation

This subchapter involves methods and means for test execution and evaluation. The first part involves testing methods. Test evaluation and some problems in testing are discussed in the second part. The last part discusses some testing tools.

5.2.1 Testing Methods

Coverage criteria can be regarded as testing methods. About coverage criteria see subchapter 5.1.2. There are surveys and classifications about testing methods, see e.g. (Peng & Wallace 1993). Some of them cover other issues like fault coverage and test case selection methods, e.g. (Adrion *et al.* 1982). There is research about finding theoretical foundations for testing. For example, Hamlet (1994) has a survey about foundations of testing; it involves e.g. coverage, models, and dependability. Table 21 presents some typical testing methods. Studies are separated from each other by periods unless stated otherwise.

Table 21. Typical testing methods

Random testing. E.g. Duran and Ntafos (1984) assess random testing.
Risk-based. Common use test cases and test cases based on many kinds of risks like timeout in input, wrong number of input arguments, etc. may be tested (Kaner 2004). Amland (2000) has an overview. Some test cases can be based on previous failures, and some may be out-of-bound-cases (Kaner 2004). See also stress-testing row since matters like strong workload can be tested in risk-based testing.
Failure-based. See e.g. (Richardson & Thompson 1993) for test data selection.
Fault-based. Stamelos (2003) investigates associative shift faults and a model to detect those faults. (Tai 1993) is about predicate testing, including methods for detecting extra or missing predicates and operators. (Tai 1996) is an instance of studies involving the question about how many test cases are needed for eliminating each fault type in predicates (this study investigates eliminating all Boolean, relation, and/or off-by arithmetic faults). DeMillo and Offutt (1993) present experimental results about constraint-based testing, where faulty conditions are written as constraints. See (Morell 1990) about theories of fault-based testing, combinations of faults, and what to do if no faults exist. There are theories about alternative test sets and when they differentiate program from its alternatives. Equations that determine alternatives not differentiated by the test are analyzed in the article. In the study, expressions are replaced by symbolic alternatives, and system output is an expression in terms of input and its symbolic alternatives. In the article, those system output expressions are equated with the output from the original program.
State-based. Paths are often represented by trees (Lee & Yannakakis 1996). Different test methods like DS, UIOS, W, and Wp, have been developed for testing state machines (Dorofeeva *et al.* 2005), (Lee & Yannakakis 1996). Lee and Yannakakis (1996) survey problems in testing finite state machines, and several fundamental problems like state verification or identifying unknown initial state.
Flow based. Podgurski and Clarke (1990), and Laski and Korel (1983) study control and data dependencies and their use in testing and debugging, e.g. in detecting operator faults and dependence faults. Test cases can be built from program slices, too, see e.g. (Hierons *et al.* 2003) about conditional slicing to choose partitions in partition testing. Flow based dependence analysis and search methods have been developed for finding input for a specific statement, assertion, or path; see e.g. (Allen & Cocke 1976) and (Snelting *et al.* 2006).
Path approach. Path approach is a method where input is being iterated until a specific path is executed (Peng & Wallace 1993). Some studies involve recursive programs (Snelting *et al.* 2006); the article is about finding input for a specific path. There are other studies, too, about finding test input for executing a specific path, branch, or statement, see e.g. (Sy & Deville 2001). Howden (1976) studies reliability of path analysis and different kinds of path related errors. Howden (1986) analyses properties of path faults. Ntafos and Hakimi (1979) study path coverage problems in digraphs. Watson and McCable (1996) describe path testing methodology based on cyclomatic complexity of the control flow graph. Malevris (1995) presents means to restrain infeasible paths in testing all sequences and jumps. Zeil (1983) studies finding undetectable expressions for a test path when the class of error expressions is a vector space. There are new studies about paths in software. See (Ngo & Tan 2008) for a heuristic method for detecting infeasible paths.
Branch based. See e.g. (Howden 1980).

Continued on next page

Software/hardware integrated critical path analysis. A standard (MIL-STD 882B 1984) mentions the method but does not define the concept of critical path. According to Kundu (1978), critical path is a path where according to some complexity measure, there is the greatest number of e.g. statements, variables, or their dimensions. The article involves using groups in finding optimum critical path in a directed acyclic graph that can be used for software testing.
Class based. Plenty of research is being done about how to test object oriented software, particularly classes. Theories of behavioral equivalence are often used (Chan *et al.* 2002). There is research about object-oriented state based testing, see e.g. (Briand *et al.* 2004). Porwal and Gursaran (2004) study weak branch criterion evaluation for class testing; effect of the length of test sequences, nature of faults, and class features, on fault detection ability was studied for C++ classes. In the weak branch criterion, a pair of labelled edges is replaced by one unlabelled edge.
Event-oriented object testing. Event-driven nature of object based programming brings declarative aspects on integration and system testing (Jorgensen & Erickson 1994).
Antirandom testing. Antirandom testing means choosing test cases that differ most from each other (Malaiya 1995); Malaiya also presents metrics for this difference.
Mutation. Research is being done about choosing mutants, e.g. (Wong & Mathur 1995), and about coupling of mutants (see subchapter 2.3.3). Budd *et al.* (1980) study mutation analysis for programs, particularly programs with decision tables. Where to locate mutants and how to test them are being studied, see e.g. (Voas 1992). See (Delamaro *et al.* 2001) about interface mutation in integration testing; errors that have an effect on other functions and output, can be seeded to functions. Woodward and Halewood (1988) present problems in deciding whether a mutant is live or dead, and solutions for these problems.
Domain testing. When using this method, testpoints are chosen at or near the boundary. Boundary faults may be due to, e.g. incorrect branch predicates or erroneous assignments that affect predicate variables (Clarke *et al.* 1982). Research is being done about the nature of border shifts (Clarke *et al.* 1982). Test strategies are being investigated, e.g. what untested areas follow from choices of test cases is being studied, see e.g. (Clarke *et al.* 1982). White and Cohen (1980) inspect language features and troubles that they cause for domain testing. Jeng and Weyuker (1994) present a method to figure out executable paths. There are other problems, too, like loops and dynamic structures (Jeng & Forgács 1999). Research is being done about how to make domain testing more efficient, e.g. how to improve coverage or accuracy, or develop simpler strategies for complex situations, see e.g. (Jeng & Forgács 1999), (White & Wiszniewski 1988).
Combinatorial testing. E.g. Cohen *et al.* (1997) study combinatorial design in generating test sets. Grindal *et al.* (2004) survey combinatorial testing research and strategies. Many combinatorial strategies for choosing test cases are based on some combination-based coverage criterion (Grindal *et al.* 2004). Grindal *et al.* (2004) survey e.g. in parameter order -methods and their extensions.

Continued on next page

Partition testing. Goodenough and Gerhart (1975) present fundamental theorems of testing based on equivalences of test cases. They use decision tables in test data selection. Every value within an equivalence class is equal in the sense of the classification. If one is able to build homogenous classes, either all test cases in a class produce the correct state and output in respect for the specific fault, or all test cases reveal the fault (Weyuker & Ostrand 1980). According to Pasquini *et al.* (1996), an equivalence class may be scattered in many parts of the code. Partition testing uses all information and can reveal unknown combinations, particularly logical faults (Hamlet & Taylor 1990). It is an excellent method if partitions with high failure rate are small (*ibid.*). Kaner (2004) has a collection of errors that students make when they apply partition testing. There are studies about how to build partitions and choose representatives of them, see e.g. (Hierons *et al.* 2003). Weyuker and Jeng (1991) present strategies for considering all built partitions if partitions overlap. Ostrand and Balcer (1988) present constraints on partitions to eliminate contradictive and impossible partitions. Bastani's study (1985) involves hierarchic equivalence classes and probabilities.
Model-based. Paradkar (2005) surveys research about how to choose test cases from models. Models can be e.g. graphs or algorithms. Pretschner *et al.* (2004) translate a model into constraint logic programming code. Muccini *et al.* (2004) investigate testing software against architecture. Different abstraction levels are possible with relation to a specified view, and different implementations of architecture are possible (*ibid.*).
Comparing program to another program or a reference. Back to back testing is discussed in (Peng & Wallace 1993) and (Avižienis *et al.* 2004).
Testing by comparison. Avižienis *et al.* (2004) mention testing where outputs are compared with each other or output is compared to a reference.
Stress-testing. Stress-testing means testing factors like large sizes, large values, large and small frequencies, or premature input; (Peng & Wallace 1993), (Clermont & Parnas 2005). Oehlert (2005) studies fuzzing an application with unusual data, e.g. detecting buffer overruns with large input values, or detecting wrong signs by flipping the top bit of an integer. Krishnamurthy *et al.* (2006) study session-based workload generation for stress testing.
Performance testing. Testing system performance. See (Avritzer *et al.* 2002) and (Weyuker & Vokolos 2000).
Interface testing. See e.g. (Briand *et al.* 2003) about client-server class integration testing.
Integration testing. Integration testing can be e.g. top-down, bottom-up, or sandwiched (Peng & Wallace 1993).
Regression testing. Research is being done about techniques for choosing a method for regression testing, and methods have been surveyed and studied, see e.g. (Rothermel *et al.* 2004). The study investigates e.g. decision whether to reset some or all test cases when software has been modified, and the granularity of test suite. Li and Wahl (1999) survey regression testing and choosing test cases. Research is also being done about wrong and missing changes and about what should have been changed (Leung 1995). Leung also discusses fault detecting ability of selective regression testing.
Symbolic execution. Symbolic execution of loops has been studied (Adrion *et al.* 1982). Those situations are being discussed where the number of iterations is not always known in advance, see e.g. (Jeng & Forgács 1999). Symbolic execution trees can be used in testing (Adrion *et al.* 1982).
Structural analysis. In (Peng & Wallace 1993), structural analysis means testing structures with automatic tools.
Fault injection. See (Zeil 1983) about perturbations, and (Fu *et al.* 2005) about compile-time fault injection.

Continued on next page

Simulation. Research is being done about randomness (L'Ecuyer *et al.* (2007) mention some studies), making rare events more likely (L'Ecuyer *et al.* 2007), combining discrete, continuous, and analytical simulation (Donzelli & Iazeolla 2001), developing abstract simulation (Lee & Fishwik 1999), and integrating simulation with modelling (Lee & Fishwik 1999) (Donzelli & Iazeolla 2001). Ng and Chick (2001) study reducing input uncertainty in a way that reduces output uncertainty in simulations. McGeoch (1992) studies analyzing algorithms and reducing variance. See (Lee & Fishwick 1999) for multimodeling methodology for real-time simulation.
Debugging. Debugging can be e.g. event based (Lazzerini & Lopriore 1989), algorithm based (Stumptner & Wotawa 1998), trace based (Shapiro 1983), dependence-based (Strumptner & Wottawa 1998), or slice based (Wong *et al.* 2005). There are studies about fault localization and characterization, e.g. (Lawrance *et al.* 2006), and about simplifying and isolating failure-inducing input in testing (Zeller & Hilderbrandt 2002). Uchida *et al.* (2002) present a model for analyzing the reading strategies that can be used in debugging. (Stumptner & Wotawa 1998) is a survey about intelligent debugging. Nikolik (2005) presents convergence debugging, i.e. searching for test cases close to faulty ones by comparing how many times expressions are evaluated true and false. Debugging tools may contain automatic tractability (Pohjolainen 2002).
Log file analysis. See (Andrews & Yingjun Zhang 2003).
Constraint analysis. Constraint analysis is mentioned in (Peng & Wallace 1993).
Cross-reference list analysis. This method is mentioned in (Peng & Wallace 1993) and (MIL-STD 882B 1984).
Bounded exhaustive testing. In this method, all inputs are tested up to a specific complexity or size, see (Marinov & Khurshid 2001). For example, Sullivan *et al.* (2004) assess bounded exhaustive testing.
Mining. Song *et al.* (2006) study defect association mining and correction effort prediction. E.g. defects in a transaction are involved in the study. Li and Zhou (2005) introduce a miner for extracting rules and detecting their violations. Li *et al.* (2006) study mining copy-paste bugs.
Evolutionary or adaptive testing. See e.g. (Bergadano & Gunetti 1996) about inductive program learning. The study involves testing program and distinguishing it from other possible mutant programs by learning from a finite set of input-output examples.

Table 22 contains nearly-orthogonal classifications of testing methods.

Table 22. Classifications of testing methods

Structural testing / Functional testing. Test cases are built from design and code in structural testing, and from external specifications in functional testing (Adrion *et al.* 1982). There are other sources, too, like error logs (Andrews & Yingjun Zhang 2003).
View. Table 21 presents methods based on risks, faults, failures, coverage of elements (e.g. paths, branches, states, or classes), structure, model, or stress. Each view is presented in different row.
Entity. The entity that is tested can be e.g. unit, component, or integration of components (Peng & Wallace 1993). Testing can also be system testing or acceptance testing (*ibid.*). Elbaum *et al.* (2009) study using system test cases when choosing unit test cases.

Continued on next page

Life cycle phase. Testing can be performed during different phases of software life cycle e.g. during specification-, design-, coding- or maintenance phase (Adrion *et al.* 1982).
Static / dynamic. See e.g. (Adrion *et al.* 1982). Many kinds of dynamic techniques are discussed e.g. in (Peng & Wallace 1993), like those based on dynamic flow testing, comparing to a reference, fault injection, or debugging. Many methods have static and dynamic versions, for example data flow analysis can be static or dynamic (Boujarwah *et al.* 2000).
Time-related / no time-related. Many test methods can be performed independently of time or old versions of software. Regression testing (see table 21) is related to time.
Stochasticity. Test cases can be selected by deterministic or probabilistic basis (Thévenod-Fosse & Waeselynck 1993).
General **/ system part specific.** In table 21, for example path testing is general, and interface testing involves only interfaces.
Debug testing / operational testing / combination. In debug testing, existing faults are located, and in operational testing, the quality of software is assessed, and many testing methods have both goals (Frankl *et al.* 1998). Subcategories for debug testing are searching for likely bugs (Frankl *et al.* 1998) and tracking known bugs (Adrion *et al.* 1982).
Incremental / cross-checking / none. Le Traon *et al.* (2003) use this classification for data flow test methods.
Advance design / adaptive testing. In extensive testing, the test set that has been planned in advance is executed, and in adaptive testing, defects are corrected in the test cases (Munoz 1988).
Directed / representative. Choosing test cases based on a specific criteria suitable for detecting a specific class of faults is called directed testing and testing based on operational profile is called representative testing in (Mitchell & Zeil 1996).
Using one technique / combining several techniques. For example, directed and representative testing are combined in (Mitchell & Zeil 1996).
General / application domain specific. Table 21 presents general or widely used methods. Table 23 presents domain-specific methods.

Table 23 contains examples of domain-specific testing.

Table 23. Examples of domain-specific testing

Domain	**Examples of testing**
Database	Consistency, integrity, indirect access; Peng and Wallace (1993) discuss these features in connection with static database analysis.
Spreadsheet	Fault localization, e.g. faulty cells or variables (Lawrance *et al.* 2006).
Expert systems	Testing rule based systems (Kiper 1992).

Continued on next page

Concurrent systems	Graph based methods, e.g. (Taylor *et al.* 1992); comparing execution traces to specifications (Brockmeyer *et al.* 1996); replay of sequences (Tai *et al.* 1991); reachability analysis (Cheung & Kramer 1994). Some tools look for atomicity (Flanagan & Freund 2004).
Protocol testing	Motteler *et al.* (1995) investigate ways in which conformance testing may fail to catch faults. They also briefly survey studies about protocol testing methods.

Chu (1997) presents an evaluation framework for software testing strategies. Some studies test empirically or assess one or several test methods, see (Miller, Roper, *et al.* 1995) for a survey. One can test or assess e.g. ability to detect faults (Hamlet 1989), number of test cases needed (Dorofeeva *et al.* 2005), or maximization of coverage (Hamlet 1989). Vouk and Tai (1993) study estimating testing methods based on changes and e.g. test history.

There are comparative research methods to compare different testing methods based on their fault detecting ability (Hamlet 1989). Many comparative studies have been done, and Hamlet mentions some of them in the study. Empirical and analytical methods have often been used (Hamlet 1989). Test sets and fault criteria have been studied, and relationships between different testing methods have been constructed, see e.g. (Hamlet 1989) and (Hierons 2002). Comparison of test methods has been criticized, see (Hamlet 1989). Some modeling and comparing studies involve failure regions, see e.g. (Frankl & Weyuker 2000). Some studies involve bounds or confidence intervals for defect detection, see e.g. (Hamlet 1989). The results of comparative studies of testing methods sometimes seem somewhat contradictive, at least partly due to using different input subdomains, see (Weyuker & Jeng 1991). Miller, Roper, *et al.* (1995) discuss problems in evaluating test criteria. Hierons (2002) studies how test sets or test criteria can be compared in deterministic implementations in the presence of test hypotheses or fault criteria.

5.2.2 Estimating Testing

Marick (1997) discusses classic testing mistakes in organizational level. Sullivan *et al.* (2004) discuss errors in testing methods like errors in oracle, errors in specification where the test is generated, incompleteness, or limited focus. Kelly and Shepard (2004) present problems in testing. For scientific software, or other software where the application domain is complex, testing is difficult and can be inefficient, due to factors such as very long running times, multiple symptoms disguising the root cause of a problem, the lack of effective oracles, and the cause/effect chasm (Kelly & Shepard 2004). The cause/effect chasm means that the symptom of a problem being far removed in space or time from the root cause (Eisenstadt 1997). In (Clapp *et al.* 1992), a test generally revealed one fault; in one test, three faults were detected. A test technique that relies solely on some code cannot find missing-path faults (Jeng 1999).

Howden (1976) proves that no testing strategy can be constructed which is reliable for all programs. Reliability means in this context that if faults are not found when a program is tested, it has no faults. The prerequisites for testing being reliable are studied in the article.

How to supervise the quality of testing is under research. There are measures about failures, like MTTF (mean time to failure) or number of faults found (Hamlet 1994), (Peng & Wallace 1993). Some measures involve testing effectiveness. Cangussu *et al.* (2003) investigate statistics of faults left and rate of fault reduction. Test effort, complexity, quality of test process, and some coefficients are used as parameters. Sensitivity analysis is performed for those parameters in the study. Capture-recapture methods have sometimes been used in measuring testing accuracy, see e.g. (Ohba 1982), and see (Isoda 1998) for criticism. Nikolik (2006) studies measuring diversity in control flow and data flow between different test cases. Munoz (1988) studies product testing and discusses correctness measurement problems. Van Rompaey *et al.* (2007) investigate metrics for evaluating poorly designed tests.

Classification criteria are being presented for test quality measurement (Chen *et al.* 2004). Le Traon *et al.* (2003) present axioms for diagnozability in testing, and develop measures for diagnozability in flow based and design-by-contract testing. Testability indicates test effort and efficiency. There are studies about relationships between testability and inductive inference, e.g. (Cherniavsky & Smith 1987). Benander *et al.* (2000) have made a study that supports the hypothesis that in recursive algorithms, finding and fixing bugs is more likely and faster than in iterative algorithms.

There are measures about testability. For example, Sohn and Seong (2004) present a measure for testability and investigate its use in fault tree analysis. The measure is based on failure probability of a statement or a variable and entropy that describes the importance of the variable or statement. According to Bertolino and Strigini (1996), testability is not directly proportional to trustworthiness. The definition of testability is revisited in the article: the incompleteness of the test oracle and the possibility of observing faults in the absence of failures are taken into account, not only program structure and input distribution. The Bayesian inference level is more useful than classical confidence level because a-priori beliefs are crucial and some erroneous programs are inside confidence intervals (*ibid.*).

See (Wooff *et al.* 2002) about whether uncertainties should be quantified. The study involves Bayesian graphical models for software testing. A probability model for failures is introduced in the study, about probabilities that fault transfers to another node of a graph. Estimating root node probabilities is discussed in the article. Sensitivity analysis is done in the article for initial specifications about how changes affect pre- and post-testing reliability.

5.2.3 Testing Tools

Automatic tools have been presented for testing, proving, and estimating software properties. For example, Jalote (1989) studies automatic testing of completeness of specifications. There are tools for automatic generation of test cases, see e.g. (Pohjolainen 2002). Some tools transform program to something else, e.g. build graphs, and generate test cases from them, see e.g. (Kansomkeat & Rivepiboon 2003).

(Tian 1999) is a survey; some types of tools related to data capturing, analyzing, usage testing, or reliability assessment, are presented in the study. Pohjolainen (2002) has made a classification and a survey; the study contains e.g. tools for functional testing, test case generation, complexity measurement, coverage analysis, regression, load analysis, and test management. Many tools have several tasks (Pohjolainen 2002). There are classifications for some specific kind of tools, e.g. Shahmehri *et al.* (1995) classify debuggers. Some studies compare tools to each other, e.g. Horwath *et al.* (2000) compare SilkTest and WinRunner. There are also guidelines for choosing tools, e.g. (Johnson 2007). On the

Internet there are collections containing hundreds of tools, for example (QADownLoads) website contains numerous tools.

5.3 Summary of Testing

How to choose test material has always been discussed. At least since 1980's, lists have been made about items to be tested, and coverage criteria have been developed later. Those early lists cover e.g. data elements and their interaction, expressions, risks, unusual situations, and loading. Those lists are sometimes detailed but it is hard to make them complete. For example, maximum and minimum absolute values with both signs should usually be tested, but that has not been mentioned on detailed lists. There is research about how rare events will be taken into account in testing. There are many different sources for test cases. In fault injection based testing like mutation testing, faults are created; research is being done about where they should be located.

There are lots of general and special testing methods. Some methods like path and branch testing have been studied at least from the 80’s but some methods like performance testing have been studied for only about ten years. What is being tested depends on the method - maybe too much. In object based testing, object oriented features like inheritance are being tested. In boundary value analysis, attention is paid for testing expressions, paths, and borders. The error classification used in boundary value testing is barely used elsewhere, although it would be useful in other contexts, too. There is a question about if more attention should be paid to boundary errors when using other testing methods.

Constraints to limit test cases have been under investigation. When to stop testing is being studied a lot. Deriving the minimum number of test points for a specific method and minimalizing the number of test cases have been related to test coverage. Sometimes it is claimed even in research papers that exhaustive testing would guarantee that there are no faults, but that is not the case: all unexpected behaviours cannot be detected by testing. For example, missing paths are hard to detect by testing. In addition, software may have different environment at different times. For example, software can read from an uninitialized memory area. If the default values of memory areas are zeros, the values are most likely zeros during the test execution. Sometimes in run time when they are not zeros, the hidden bug actualizes.

In the field of testing, there is a lot of research about relationships. Some studies create relationships between coverage criteria, between test sets, between test methods, fault classes, or between test coverage properties and software properties like number of bugs. Some studies involve fault regions. Studies comparing coverage properties with each other or with software properties have been criticized. Some studies compare testing methods. As with comparative studies about defect prediction methods, results of studies that compare testing methods have been contradictive. Reasons for this are being discussed and research is being done about how testing methods should be compared. Frameworks, methods, models, and metrics are being developed for test evaluation. Problems in testing are being discussed. There are axioms involving testing. What makes testing reliable is being studied, too. Testability has been studied and discussed. Automatic integrated tools have been presented for static analysis, testing, proving, and estimating software properties. There are surveys about testing tools.

There are interesting relationships that should be kept in mind. Dependence of data elements has a relationship to the number of faults that have an effect on a failure situation. The relationship between data element dependencies and failures is discussed in subchapter 2.3.3. This relationship could be studied further. Flow dependence is related to test coverage, and dependence information is used in creating coverage criteria. Data flow dependence is related to dynamic dependence of data elements, which should be kept in mind in assessing test coverage.

As mentioned in the summary of the previous chapter, there is much more research about testing than about checks. In addition, there are studies about testing of special items and other unusual situations. However, only few studies involve paying attention to unusual situations already during development.

In addition, many methods are used in testing only, although faults could be eliminated earlier by using those methods during development. A lot of research has been done about using domain theory and category partitioning in testing phase but no research was found about applying those methods during other phases of software life cycle. As mentioned in the summary of chapter 2, blindness and coincidental correctness are rarely investigated outside domain testing, even though they have connections with other analysis and testing methods, too. Also, path-, branch- and data flow analysis are often performed in testing only even though they should have been performed during earlier phases of life cycle.

6 FAULT TOLERANCE

Even if software developers use methods presented in chapters 4 and 5, there can still be faults in software. One usually does not check everything, proving may contain errors, application of formal methods has restrictions, and everything cannot usually be tested. Particularly, rare situations and external factors are often omitted from testing. Often there is a possibility of an external failure, and there are unpredicted factors; testing reveals some of them but not all. This chapter involves preventing harm if there is a possibility of a failure. The probability of a failure in the presence of a fault can be reduced, and the software usually needs to have means to recover and continue when a failure occurs.

The first subchapter introduces some terms, problem areas, and general issues of fault tolerance and of related modeling. The three following subchapters involve the most common means for maintaining reliability. Failure diagnosis, N-version programming, and recovery, are discussed in consecutive subchapters. Several of those methods are often applied in the same system. For example, in N-version systems, the software may use recovery blocks in the components or after the result of the N components has been calculated (Torres-Pomales 2000). Also, multiversion software may perform self-checks (Torres-Pomales 2000), which are a common diagnosis means. The fifth subchapter involves some other reliability means, methods, tools, and development trends. The sixth subchapter is a summary of this chapter.

6.1 Introduction

Reliability issues have been applied for hardware. They have been adapted for software, and special features of software have been taken into account more and more. General surveys have been done about computer system reliability for several decades; (Strigini 2004) is a relatively recent survey.

The use of the most common terms in the field is not very consistent. Some most commonly used definitions are presented below.

Safety: Features and procedures which ensure that the product performs normally in normal and abnormal conditions, thereby minimizing the likelihood of an unplanned event occurring, controlling and containing its consequences, and preventing accidental injury, death, destruction of property, and/or damage to the environment, whether intentional or unintentional (Herrmann & Peercy 1999). Safety means that it is guaranteed that something bad never happens (Phillips 1992).

Reliability: The probability of failure-free operation of the software program for a specified time under specified conditions (Herrmann & Peercy 1999). Another definition for reliability is that it is a set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time (*ibid.*). Very often, reliability means preventing the damage if something bad happens anyway. If there still are faults in software, damage caused by them can be prevented by means of reliability.

Liveness: The system stays alive. Liveness means that it is guaranteed that something good eventually happens (Phillips 1992).

Damage: Loss or detriment caused by hurt or injuries affecting estate, condition, or circumstance (Oxford Dictionary IV 1989). Damage also means injury, harm, disadvantage, inconvenience, trouble, matter for regret, misfortune, or a pity (Oxford Dictionary IV 1989).

Harm: Evil, hurt, injury, damage, mischief, loss, grief, sorrow, pain, trouble, distress, affection, pity, or a pity (Oxford dictionary VI 1989).

Plenty of research is being done about error detection and recovery. Different *methods* are assessed and compared, see e.g. (Leveson 1991). Research is being done about how software can efficiently detect faults and recover from them. How to use information that is meant to be used for other purposes is being studied, see e.g. (Lomet & Salzberg 1991). One has often been able to achieve non-intrusive checkpointing and information collecting, see e.g. (Israel & Morris 1989). One *problem* is that recovery systems may have bugs see e.g. (Torres-Pomales 2000), and they cannot always detect other faults than what they have been planned to detect (Abbott 1990). Generally, faults are not detected either if detection and recovery mechanisms are in wrong *locations*. Where to place fault tolerant structures and checkpoints is also a topic for research, e.g. Torres-Pomales (2000) surveys the topic.

Different subsystems or modules may have independent or even *different* fault tolerance *techniques* (Strigini 2004). Functions, modules, or some other entities, or whole systems can be replicated; some studies compare different solutions, e.g. Boland and El-Neweihi (1995) compare component level and system level redundancies. Defensive programming has often been placed in different parts of software, but Kantz and Koza (1995) present a system where defensive programming has been built as an independent subsystem.

One area of interest in research is how to *fit software components* together in a fault tolerant way. Arora and Kulkarni (1998a, 1998b) study adding fault-tolerant components to increase the level of reliability and the number of fault classes that the system can tolerate. Sinha and Hanumantharya (2005) combine using of that method and prior use of category theory in composition. Also the *depth* of fault tolerant structures has to be decided when planning fault tolerant software (Abbott 1990).

There are studies that have mathematical *analysis* of fault tolerance, see e.g. (Wu *et al.* 1996). Some fault tolerance models include combinations of different fault tolerance means; for example, (Wu *et al.* 1996) contains systems with both N-version programming and recovery blocks. Some defect prediction models can take correlated failures and other special matters into account. For example, they may use information about numbers of correct programs, see e.g. (Littlewood *et al.* 2001). Fault tolerance models are usually used for defect prediction, reducing development or testing effort, or finding optimal configurations, see e.g. (Wu *et al.* 1996). Kim *et al.* (2004) present a framework-based approach for analyzing the reliability of an embedded system based on the framework and component reliabilities. Plugin components are possible, and separation of concerns is applied in the study. Some experimental studies *compare* different recovery methods, e.g. Bhargava *et al.* (1990) compare recovery with synchronous checkpoints with recovery with independent checkpoints.

6.2 Fault Detection and Diagnosis

For many accidents, ignoring the signs about that something is wrong has been a contributing factor, see e.g. (Leveson 2001). Doing something for those signs would have saved the situation. The same can be true with software, which makes failure detection and diagnosis important.

Self-checks are an important means for diagnosis. Complexity theory is used in the *theory* of self-checking, see e.g. (Wasserman & Blum 1997). Wasserman and Blum study developing good result checkers. Checkers can even be adaptive: a complex checker may generate more detail data if it does not have enough of it (*ibid.*). Staknis (1993) investigates n-way dependencies and independencies of a fault on input variables, and on different degrees of checks of those variables. Laws about relations between checks and about logical connectives of input checks are also derived in the study; e.g. conjunction between checks means performing all of them. Methods have been developed for testing and assessing *robustness*, see e.g. (Bennani & Menascé 2004), and characteristics for robust algorithms are discussed, see e.g. (Strigini 2004).

Nicola and van Spanje (1990) analyze *models* about checkpointing strategies. Checkpointing intervals based on Poisson process, transaction load, or time intervals are analyzed in the study. The study also investigates dependence of the recovery period on checkpointing strategy; in the article, the dependence can be deterministic, stochastic with distributions, or parametric. Some diagnosing methods use *partitions*. Preparata *et al.* (1967) have theories about how many faults can be diagnozed in partitioned systems where units can test each other, and those methods have been developed further. Jeon and Cho (2002) present an adaptive partitioning model for system-level diagnosis. Walter *et al.* (1997) investigate formal *verification* of on-line diagnosis. Guo, Mukhopadhyay, and Cukik (2004) study verifying result checkers. Huebscher and McCann (2008) survey autonomic computing, including self-checking, self-fixes, and self-healing.

Table 24 presents methods for defect detection and diagnosis, classified by view. Some methods contain automatic recovery. Different studies are separated by semicolons unless stated otherwise.

Table 24. Fault detection and diagnosis methods

Data structure redundancy Repeated dual links, other repetitions in data structures, extra rows and columns in matrices (Strigini 2004).
Code checks during runtime Type checks (Cartwright & Felleisen 1996); data structure consistency based on check and repair (Demsky & Rinard 2006); consistency in data structure processing, e.g. detecting cycles by monitoring path length (Giguette & Hassell 1999); repeating an operation and comparing results (Tewksbury 2002); comparing values of variables with previous values, with values of other variables, or to references or reasonable values (Torres-Pomales 2000); reading back output (Gericota *et al.* 2006); backward calculation (Torres-Pomales 2000); error correcting codes like checksums or codes for detecting transposed bits (Torres-Pomales 2000), (Gallian 1996); assertions (Leveson 1991), probes (Probert 1982); looking for unintentional redundancies (e.g. commands) - they may indicate faults (Xie & Engler 2003); protecting memory by checking e.g. return addresses, Chen *et al.* (1995) discuss protecting memory from operation system crashes by different checks; comparing results to those of other replicas (Torres-Pomales 2000); simulation (Lee & Fishwick 1999); analyzing log file, see (Andrews & Yingjun Zhang 2003) for analyzing log file for testing; monitoring performance and use of resources (Swobodova 1981), and system clock (SDL 2006).
Trace processing and partitions Slicing (Weiser 1984); partitions (Jeon & Cho 2002); building process trace tree and traversing it for searching for an incorrect, incomplete, or diverging sequence of procedures (Shapiro 1983); tracking a fault that is e.g. in output (Shapiro 1983).

Continued on the next page

Software intelligence Failure states (Kumar & Vemuri 1992); self-stabilizing algorithms[5]; automatic detection of convergence problems (Troscinski 2003); wrappers doing some checks, e.g. wrappers that intercept unintentional calls to functions or calls to defective functions (Strigini 2004); rule-based systems like algorithmic checks that find conflicts and other faults, see (Stumptner & Wotawa 1998) for debugging; Martin *et al.* (2005) study a query language to look for patterns related to sequences of events and objects - queries are converted to checkers that look for rule violations; neural networks may be used in detecting both known faults (He *et al.* 2000) and unknown faults up to external disturbances (Tewksbury 2002).
External checks Information flow, incremental and cross-checking strategies (Le Traon *et al.* 2003); watchdog (Torres-Pomales 2000); paying attention to symptoms like suspicious behavior or degraded performance, recognizing faults based on common symptoms (Lee & Iyer 2000), (Sheth *et al.* 2005); diagnostic messages (Hatton 1999); checks outside algorithms like monitors for programs (Strigini 2004), or programs that check other programs (Strigini 2004), Blum and Kannan (1995) present program checkers and theorems about checkers and checkability.

6.3 N-Version Programming

Avižienis (1995) discusses a design paradigm for N-version software, i.e. for making N copies of the software. Chen and Bastani (1992) study partial replication of software, where only some of the system data is stored in replicas. In the study it is assumed that the whole process can be restored from a partial data replica unless faults prevent the restoring effort. Control processes in systems with replicas are also under investigation, see e.g. (Bishop 2006). One problem is keeping the replicas consistent, see e.g. (Brilliant *et al.* 1989) and (Bishop 2006).

Software components may work as main, backup (active), or spare (standby) components. A great amount of research is being done about how small the copies should be, how many copies of software there should be, should the copies be of different types, and how many copies of each type there should be, and if some of the copies should be spare copies. Hocenski and Martinovic (1999) examine reliability factors of a system that has a hardware spare copy and checkpoints where data is transferred to the spare unit; they perform experiments for both redundant and non-redundant software. If there are more than one original components in a system that contains spare components, either there can be spare copies for individual components or spare components can be shared, e.g. Torres-Pomales (2000) presents both kinds of solutions. In warm standby systems, the probability for spare component failures can be anything from zero to that of active components. If a system has N components and still works when K or fewer of them are faulty, it is called K-out-of-N-system (N > K). See (Behr & Camarinopoulos 1997) about methods for comparing the reliability of incomplete K-out-of-N-systems, where not all paths are present.

[5] The word "stabile" means different things in different fields of mathematics and computer science. It is often related to the precision of computations. Self stabilization of algorithms often means that the system goes to a legitimate state in a finite amount of time, see (Strigini 2004). How to cope with transient faults is also being studied, see e.g. (Mossé *et al.* 2003). Self-stabilizing algorithms and their restrictions are under investigation, see e.g. (Das *et al.* 1996) for a self-stabilizing algorithm for directed acyclic graphs. Self-stabilizing algorithms should work even if the data has been corrupted (Strigini 2004).

According to Brilliant *et al.* (1989) and Littlewood *et al.* (2001), there may be individual differences in reliability: some 2-out-of-3 systems can be less reliable than some single systems. Sometimes the maximum number of faulty units is known or may be assumed. Xu and Randell (1997) analyze variations of deciding which units are correct if there are comparators and there can be at most T faulty units.

Models are being developed for systems with a specific configuration, e.g. for K-out-of-N-systems with identical components (Lai *et al.* 2002), or systems with non-identical or spare components, see below. Some models can take into account gradual degradation (Shmueli 2003), reparation times (Vermeulen *et al.* 1998), preventive maintenance (Vermeulen *et al.* 1998), component-based failure rates (Vermeulen *et al.* 1998), common mode errors, or uncertainty. That uncertainty often involves unavailability of spare components. She and Pecht (1992) analyze K-out-of-N warm standby systems. See (Bunea *et al.* 2005) about Bayesian models where information of other plants is used. Dai *et al.* (2004) present a model for correlated failures where failure distributions are not restricted to be of a specific type. In the model, different components may have different failure distributions, so can different N-way combinations of components, even for the same N.

Data transfer between replicas is one topic for research. Basic ways to transfer data are comparators and reciprocal monitoring, e.g. Torres-Pomales (2000) describes both kinds of mechanisms. Comparators are studied a lot; there is not so much research about reciprocal monitoring except for distributed systems. Data transfer means are often replicated, because they also may be faulty, see e.g. (Torres-Pomales 2000). There is research about how to get an *agreement* about the return value or system status. Examples of methods for estimating results of different components are comparison, voting methods, switching, self-checks, and acceptance tests (Littlewood *et al.* 2001). In voting, the results between replicas are compared to each other, and possibly some other factors are used, like fault histories of the replicas (Torres-Pomales 2000). Multi-stage voting and hierarchical voting are discussed, see (Vouk *et al.* 1990).

In voting, majority can be wrong (Bishop 2006). Another problem is to decide about when the results of different copies are regarded as different results (Brilliant *et al.* 1989), (Brilliant *et al.* 1990). In addition, results can be different from each other, but all may be correct. If the differences are due to inexact computation, this is an instance of the consistency comparison problem (Brilliant *et al.* 1989). Sometimes it is not even clear to decide whether results can be compared; for example, internal states may have an effect on results of individual copies for a short period of time (Brilliant *et al.* 1989), or if the software contains a number of related values, it is not sufficient to take a vote based on individual values (Bishop 2006).

See (Leveson *et al.* 1990) about *correlated failures* between different software versions, and about problems in voting and self-checks. Also, several faults that cause a failure (always or in specific situations) may have a common cause. There are experimental studies about how frequently common mode failures are present, see e.g. (Bishop 2006). In the vast majority of multiple processor halts for Tandem GUARDIAN operation system, the same fault halted primary and backup processors (Lee & Iyer 1993). Few halts were due to independent processor faults, and few halts were not related to faults on both processors (*ibid.*).

Plenty of research is being done about *diversity* between different copies. Different demands can be loosely coupled (Lee & Iyer 1995), or difference seeding can be used (Ammann & Knight 1988). Bishop (2006) surveys experiments about design diversity. Littlewood *et al.* (2001) discuss the question whether to choose several methods in order to cause diversity and which methods to choose. For example, the authors discuss problems in assessing strengths and weaknesses of different development methods on specific applications and conditions of operation. According to the authors, two methods may be equally good in average but those methods may be good for different sets of inputs.

Table 25 contains examples of means to cause diversity.

Table 25. Examples of means to cause diversity.

Different technical and physical actions, sources of inputs, and implementation technology for replicas of system functions (Littlewood *et al.* 2001). The study did not describe different implementation technologies for software but for example, some systems may be discrete and others may be continuous. See e.g. (Littlewood & Wright 1997) for reliability prediction for discrete and continuous systems.
Different input data values for different copies (Ammann & Knight 1988).
Different methodologies for different versions (Littlewood & Miller 1989).
Different specification languages for different copies. Based on their experiment, Yoo and Seong (2002) assume that a wrong description will produce unique faults with unique specification languages.
Different programming languages for different copies (Littlewood *et al.* 2001).
Converting input representation between different copies (Abbott 1990).
Algorithmic changes e.g. by changing expressions, data structures, or the order of storage allocation, or rearranging internal data (Ammann & Knight 1988).
Timing or causing external disturbances, Rushby (1993) discusses those as problems.
Action, state, and timing, e.g. different copies performing different tasks, memory state, race, timing, or random events (Lee & Iyer 1993).
Authors. Research has been done about if several copies of the same software version should be made by same or different people; some pieces of this research are introduced in subchapter 2.3.3.

6.4 Failure Recovery

Some systems stop when they fail, but it is usually desired that the system be able to recover. Recovery methods can be classified. A forward method brings the system into a new state in order to perform a function, or updates a file using change record data (IEEE 1990). A backward method returns the system back to an earlier state (IEEE 1990). The third fault recovery method is to include enough redundancy to be able to mask the fault (Avižienis *et al.* 2004). Avižienis *et al.* (2004) also classify means for preventing the failure from occurring again: the alternative means are diagnosis i.e. finding out the cause, isolating the fault, re-configuring the system, and re-initialization. Research is being done about these means, e.g. about how to do restarts and if one should do periodic restarts, see e.g. (Bao *et al.* 2005). One problem in recovery is that other failures, even those that are independent of the first one, may occur during recovery, see e.g. the study of Al-Saqabi *et al* (1996).

An architectural means for recovery is recovery blocks, where program components check their correctness and back up when they detect a failure. The software needs to store information about its state for recovery. Some methods use log files.

Negrini and Sami (1983) present graph- and dependence based analysis about whether the acceptance test in recovery block makes sense i.e. represents testing the system state. See (Mill 1985) about building a sufficiently correct state from a state in a recovery point. See (Wang *et al.* 1993) about progressive retry. There are checkpoints, and the deepness of the recovery and the number of participating processes are increased gradually until recovery succeeds. Method investigated in the study uses log files in recovery. Qin *et al.* (2005)

study recovery from faults by environmental changes; some other methods are discussed briefly in the article.

Plenty of research has been done about modeling recovery. The following list contains some examples.

- Failure distribution models when multiple version system contains checkpoints (Nicola & Goyal 1990).
- Validating recovery blocks by testing. The model is based on failure events (correct and incorrect results of alternates and those of acceptance test). The relationship between faults and fault correction is based on failure history and the amount of testing done. The faults are repaired and the repairing time is taken into account. (Pucci 1992).
- Including correlation between outputs of different modules (alternates) operating on a single input, between successive inputs, and between successive acceptance test runs on correct/incorrect module outputs, all in the same recovery block. (Tomek *et al.* 1993).
- Recovery blocks with nested clusters of failure points. When in failure cluster of the primary module, the input sequence encounters clusters of failure points belonging to the first alternate, the second alternate is invoked. (Csenki 1993).

Research is being done and surveyed for database recovery, see e.g. (García-Muñoz *et al.* 2007), a survey. There is also research about how to recover when there is non-determinism, see e.g. (Elnozahy *et al.* 2002). A lot of research has been done about recovery and checkpoints for co-operating processes. There can be e.g. atomic transactions, or recovery may be based on beginning and ending of a conversation (Romanovsky & Strigini 1995). There is research about modeling communicating recovery blocks, see e.g. (Berman & Kumar 1998), and about coordinating different recovery mechanisms, see e.g. (Tai *et al.* 2001). Al-Saqabi *et al.* (1996) present an algorithm that can recover a communication protocol from multiple failures and from failures in recovery process; no content of messages is lost but some messages are retransmitted automatically.

Recovery points and *exception* processing are examples of failure recovery. According to Maxion and Olszewski (2000), special situations like special items, wrong types, overflows, and precision errors cause exceptions. There may be situations where program analysis method does not notice exceptions, e.g. a subprogram may encapsulate exceptions (Sinha & Harrold 2000). Methods and models are being developed for static analysis of exception flow (Sinha & Harrold 2000) and for detecting uncaught exceptions (Jo *et al.* 2003). Some methods can analyze effect of exceptions on dependencies and control flow (Sinha & Harrold 2000). Handling multiple exceptions is one topic for research, see e.g. (Sinha & Harrold 2000). Maxion and Olszewski (2000) study eliminating exception handling failures with three methods: dependability case analysis for exceptions and their reasons, N-version programming, and collaboration. There are also studies about choosing the correct exception handler, see e.g. (Cui & Gannon 1992). Garcia *et al.* (2001) survey exception handling for different languages and present taxonomy for technical aspects of exception handling.

6.5 Other Reliability Issues

Research is being done about several other means to increase fault tolerance. Table 26 contains some of those means. Many of those means are very easy to use and do not need extra resources.

Table 26. Means to increase fault tolerance

Isolated software Software makes it possible to design complex solutions, but for safety reasons, they should be simple and isolated, and evolutive development and separation of concerns help (Littlewood & Strigini 1993). See (Alves-Foss *et al.* 2004) about multilayered approach for system design and verification.
Weak replication Replication means may also be weaker than building N versions. Kistler and Satyanarayanan (1992) present a system where cache is utilized to guarantee continuous file system function.
Data replication Research is being done about how to get replicated data files consistent when changes are made to the files, see e.g. (Lim & Hurson 2002). Taking backups (IEEE 1990) is a way to replicate data to be used to replace or restore data in a failure or a disaster.
Averaging Averaging can be performed to get better estimations of values and minimize the effect of a possible faulty value. The values can be, for example, values of different copies of the same software (Latif-Shabgahi *et al.* 2004) or values of a variable in the cause of time, see (Lander & Berbari 1989) about the latter.
Fault tolerance in programming languages Austin *et al.* (1994) investigate checks for spatial and temporal access faults, and state that UNIX typically provides storage protection on segment granularity. Some programming languages have safety features. For example, EC (Hatton 2005) and MISRA-C (Hatton 2004) are safe subsets of C. Hartel and Moreau (2001) survey research about Java safety. See (Phillips 1984) for safe data type specifications, where equality axiom is independent of implementations. De Florio and Blondia (2008) survey fault tolerant programming languages and adopting customer semantics for a language.
Safety margins Systems can sometimes be made to tolerate a more stressful environment than they are expected to have (Littlewood & Strigini 1993). This is one form of safety margins, which are one means of defensive programming. The possibility of the unknown should be taken into an account.

Fault tolerance is being built only for faults that are somewhat anticipated. It is harder to estimate the risk of the *unknown*. One cannot know exactly either, to what extent the unknown can be prevented with fault tolerance.

Abbott (1990) surveys *resourceful* systems, systems that set goals and select alternative plans if they are not achieved. Some system can even replace a function for another, and some even do reactive reasoning for an error and/or a change of environment (Abbott 1990). According to the study, real situations have uncertainties and the border between the system and its environment is fragile. Self-checks and self-protection are involved in resourceful systems (*ibid.*). Giguette and Hassell (2002) study the use of a recovery planning tree when planning recovery methods in resourceful systems.

Delgado *et al.* survey run-time monitoring *tools*. Many systems react to violations of specifications in a particular state. Use of automata and algebra is limited in tools, but logic and high-level language are common. Few tools can capture domain-level and design-level properties before implementation. Some tools have automatic recovery. (Delgado *et al.* 2004).

Lee and Iyer (2000) study SYMPTOMS-tool for rediagnozing failures based on common symptoms. See (Liang *et al.* 2004) about NT_SwiFT for Windows NT. The set of components provides components for automatic error detection and recovery, checkpointing, event logging and replay, and communication error recovery, and incremental data replication. Some tools collect dependencies between components. For example, on the webpage (Maples 2004), Windows Dependency Walker is presented. It collects information about dependencies between modules. According to the webpage, the tool can be used in error detection.

6.6 Summary of Fault Tolerance

There are some general problems involving reliability. Terms involving reliability are not always used consistently. Also, terms like safety and reliability have their special meanings, although the use of those terms is not consistent. Another general matter is that results of studies about availability could be used when studying reliability but they are not used very frequently.

There is research about methods and problems in recovery, fitting components together, and placing fault tolerant structures and checkpoints. There are different architectural and code-related solutions to achieve fault tolerance, for example self-checks, N-version systems, recovery blocks, and exceptions. For recovery blocks, the reliability of acceptance test and that of recovery are being studied. The reliability of different configurations of fault tolerant solutions is being studied; the components may be of same type or different types. Some studies involve concurrency, non-determinism, and failures during recovery of fault tolerance solutions. Some research has been done about the breadth and depth of fault tolerance mechanisms. Several fault tolerance means are often combined. Redundancy can be achieved at different levels, and it can have different degrees; more research could be done about partial or weak redundancy since it often is an easy way to increase reliability. For example, repetition of a function or a procedure could be a cheap way to increase reliability.

There are different methods for fault detection and diagnosis. Different types of self-checks are easy to use and could be used more often. For example, more attention could be paid to using multiple data structures for both increasing the reliability and making software more efficient. Other replications like repeated calculations increase reliability, too. As another example, robust algorithms could recognize input outside domain. What can be checked in software, effects of checks, and making checkers are topics for research. There are metrics and axioms for diagnozability and theorems about checks and about diagnosis. More research could be done about assessing self checks.

Control process for N-version replicas and the consistency of replicas are being studied. Data transfer between different copies is a topic for research. Comparators are studied a lot; there is not so much research about reciprocal monitoring except for distributed systems. Deciding the result of N-version computation is being investigated. Factors that cause differences between different replicas are being discussed. Sometimes differences are due to slight differences at the places and times when different units perform their tasks or get information, but I have not found any discussions about this source of differences. The possibility of correlated faults is a typical problem, too. Whether replicas should be different and ways to achieve diversity are being studied. One way to increase the difference is to make the fundamental structure different. For example, some replicas can be discrete and others can be continuous.

Other means for achieving reliability are isolating critical parts, weak replication, data replication, averaging different values, safe data types, fault tolerance in programming languages, and safety margins and other forms of defensive programming. Safety margins may mean, e.g., accepting greater input intervals than what the software uses, and making longer than necessary deadlines for events. Not very much research has been done about means mentioned in this paragraph, even though they are usually easy to use.

There are models to predict or optimize reliability; some models involve repair, correlation, nesting, communication, or other extensions, or use information of other systems. For N-version systems, research about models is concentrated on looking for the optimal configurations and their models. There are models about several means to achieve reliability, and some models combine different means to achieve fault tolerance. There is a tendency towards robustness and adaptability. Some systems have means for preventing the fault from occurring again, and some systems can recover by changing the environment. Some diagnozing methods contain automatic recovery. Some tools can reason about faults. Some tools that have been done for other purposes are used in reliability, too.

7 DISCUSSION AND CONCLUSIONS

In this chapter, the contributions of the thesis are presented in relation to the goals of the thesis. The thesis had the following goals:

1. Figuring out the *status quo* in the field.
2. Proposing some basis for structural framework for the field.
3. Surveying and increasing bug knowhow.
4. Surveying research about fault prevention, fault prediction, fault detection, and fault tolerance.
5. Figuring out what should be studied more.
6. Encouraging for early fault elimination.
7. Presenting concrete recommendations.

The contributions of this thesis are discussed in detail in the first subchapter. In addition to those investigated in the subchapter, one more contribution is to present recommendations to software developers, research scientists, and teachers. Recommendations are presented in the third subchapter. Between those subchapters, in the second subchapter, different methods for fault elimination are compared with each other. In the last subchapter, problems with doing this thesis are discussed and recommendations are presented for future work.

Plenty of research has been done about organizational, managerial, and economical means for software fault elimination. However, the emphasis in this work is in technical means and in what can be done by means of software development. The thesis investigates fault elimination only. Quality assurance contains many other areas like effectiveness and maintainability of software.

7.1 The Contributions of the Thesis

7.1.1 Figuring out *Status Quo*

Types of Research

Software fault elimination is a wide topic, and plenty of research has been done about it. Three common ways to do research have been the development of models and methods, execution of experiments, and writing surveys. Theoretical research has been done in the form of developing models and other formal entities. Lots of models have been developed e.g. for software development processes. Another area where models have been developed intensively is defect prediction. It has been widely admitted that models differ from reality, and the fact that reality is often unknown makes the situation worse. Participants for experimental studies have often been either companies or university students. Interviews and surveys have also been used as methods in experimental studies. See the subchapters 7.1.3 and 7.4 for discussion about surveys of different areas of software development.

Problems with the Difference between Research Environment and Reality

Many studies state lots of uncertainty factors. Stating them in a research document helps a lot in assessing results. There are differences between research conditions and reality. Controlled experiments are often done with university students, so they do not necessarily correspond to real-world development. Also, studying models has uncertainty because models differ from reality. For these and some other reasons, software engineering research contains a lot of uncertainty, and research methods need to be assessed. There are some studies that assess methods for software engineering research; some assessment studies are general and some are related to a specific field of software engineering, e.g. testing. Comparative research is discussed in subchapter 7.1.2.

Universal Problems are Sometimes Erroneously Connected with Only one or Few Methods

Some common types of software faults and some fault-related problems are studied only in relation to one or few specific methods, even though those faults and problems are more general. For example, blindness and coincidental correctness are investigated in connection with domain testing, but studies that do not involve domain testing usually do not take them into account. Unfortunately, software often contains blindness and coincidental correctness, regardless of whether one uses domain testing. They can often be detected with other methods, too, so studies that involve other test or analysis methods could take those types of faults into account. More generally, in some fields, the research and development approach is method-oriented, although the aspects of those studies could be taken into account in other fields and when using other methods, too.

Results of a Specific Study Could Be Used Elsewhere

Some studies concentrate on a particular application or application group like telecommunication system, a specific application type like real-time applications, or a certain phase of software development cycle. Some studies concentrate on a specific methodology like object based development, or a specific language like Java. The results of many studies are dependent on a specific system or application environment, or few of them. Very often those results could be used with some other systems and in some other application environments, too. In addition, application environments change and so does the usability of results of previous studies.

Attention should be paid to what the characteristics to the environments are where the results of the studies hold; it makes it possible to generalize the results to some extent. For example, many studies of state space explosion are related to telecommunication systems but the results could be applied elsewhere, too, since state space explosion is a quite general problem. However, people do not find those results when they could need them. People doing research for application domains outside telecommunication systems have hard to find the results that are only related to telecommunication systems and that do not even have keywords about state space explosion. As another example of studies that concentrate on particular application group, results of many studies about resource availability could be applied in increasing reliability. How a structural framework could improve the situation is discussed in subchapter 7.1.2.

Lack of Cross-Field Research

More cross-field research should be recommended. For example, items for software code inspection can often be chosen by same grounds and with same methods as test cases, and vice versa. Not much attention has been paid to this analogy, even though taking common features into account would help developing both fields. Also, defect prediction in software inspection and properties of curves in modes about faults found in testing have common features, but attention has not been paid to them. In some studies mentioned in this survey, the conclusion is that the field of the study lacks cross-field research. In those studies, only the area of the study is considered, but the problem is more general. On the other hand, there is a tendency to combine different methods. For example, several defect prediction models are being combined in assessing defects, flow analysis is often combined with other methods, formal methods are often combined, and one system may combine several means to increase fault tolerance. In addition, many researchers agree on the view that methods can be applied in different areas.

7.1.2 Proposing Basis for Structural Framework

An Outline for Structural Framework

The field of eliminating software faults is unstructured. Particularly, analysis and checking lack a position in framework. One reason for this may be that too little attention has been paid to many areas of analysis and checking. The table of contents in this work could be considered as a support when developing a structural framework. Some topics could be inserted into several categories, so the table of contents is intended to be for discussion instead of being a final classification. (SWEBOK 2007) may also be a big help when constructing a structural framework.

If there is a structural framework, studies that contain fault elimination could use keywords related to that framework, so the studies could be found more easily. A framework would also help figuring out research domains whose results could be used in specific areas of fault elimination even though their main purpose is not fault elimination. Some studies have another purpose but produce some as-a-side-knowledge about software faults. For example, plenty or research has been done about topics like logical programming in database design, data flow analysis in testing compilers, and availability and optimization issues. Some of the results could well be used in software fault elimination. Particularly, results of studies for developing compilers are often related to software control and data flow and could be used in improving flow-based error analysis methods. As-a-side-knowledge usually is unstructured and consists of occasional hints about fault elimination. In this survey, some of those hints are referred to when surveying the quality related content where they could be applied. Those hints are hard to find if the studies do not have keywords that are related to a structural framework of fault elimination. A structural framework would encourage the use of keywords related to the framework.

Improving Comparability of Different Studies

Comparison of studies has been problematic. Many studies compare different software development methods. Some studies have compared results between different comparative studies, and those results have been found contradictive. For example, different studies about defect prediction methods have different conclusions about which method is the best. The same is true for studies that compare testing methods. Reasons have been looked for those contradictions. In some fields, studies have not even been comparable. For example, studies about defect detection methods have been found incomparable, see (Miller 2000).

Reasons for this incomparability are discussed in the summary of chapter 3. Miller states that one reason for this incomparability is the great variation between those studies; for example, different studies use different methods and thus cannot be compared to each other. Another reason is that results of studies cannot always be quantified due to the lack of common definitions that could be used consistently in each study; for example, there is no common definition of bug type (*ibid.*).

It should be considered if recommendations could be established to improve comparability. However, compulsive standardization would hide individual differences. Those differences should be kept in mind. There should be a possibility to divert from standards, but diversions of standards should be explained in the documents where the diversions occur.

More Consistent Use of Terms

More consistent use of terms would make research papers, coursebooks, and software documents more understandable. It would also make comparing studies easier. There are some glossaries and development frameworks. For example, there are several online glossaries, and (SWEBOK 2007) contains good explanations of terms related to software development framework. Regardless of the existing standards and glossaries, the terminology is not used consistently in research papers. For example, many terms are not defined in standards and glossaries, some terms are defined differently in different documents, and some terms are not used consistently even if they have been defined consistently. Visser *et al.* (1997) define mismatches in the use of concepts, and those mismatches can be found, particularly between term, definition, and concept to be defined. The article also involves mismatches on classes and on relations, but the field of software engineering does not seem to have strict classifications of its terms. When writing the thesis, I found several matters causing confusion. Some examples of those matters are presented below.

Multiple definitions for terms in different glossaries cause inconsistency in the use of terms. For example, the term “desk checking“ has been defined differently in different glossaries. Many online glossaries state that it means a manual testing of a logic of a program. IEEE glossary (IEEE 1990) uses the term when it means a static review of documents or code. In some documents like (Zeil 1999), the definition is more detailed than in glossaries.

Arbitrary definitions of standard terms cause unnecessary problems since definitions in existing standards or glossaries could be used. Some terms like "inspection" and "walkthrough" are used inconsistently, even though at least some of them have been defined in glossaries. In glossaries, inspection is usually defined as a strict process with certain meetings, specific roles, and specific organization. In some papers, the word is used in the meaning of less formal methods like code review, code walkthrough, or algorithm analysis. The terms for those methods are not used consistently either.

Including local context in definitions sometimes causes troubles. Some terms have a general definition but are unnecessarily redefined in some studies. Those redefinitions add part of the context of those studies or that of their domain to the general definition without stating it explicitly. This can be confusing for those who have used the general definitions. For example, using the word “confluence” in a loop invariant study has been discussed in the summary of chapter 4.

Special IT related definitions for general terms may sometimes cause problems. On the other hand, there may be a legitimate need for information technology (IT) related definitions. Those definitions may differ from either general definitions or definitions of the same terms in other fields. Some of those terms are included in IT standards. For example, the term “inspection” usually means a strict organized process to investigate software code; inspection often contains meetings. Also, terms like safety and reliability have their special meanings, although those terms are not used consistently. Understandability and comparability may be improved by referring to the special IT related definitions and emphasizing that there are other more general definitions for the terms, too.

Arbitrary definitions of non-standard terms are often problematic. This is the case where one needs to define terms used in a study, and there are no general definitions to use. Results of comparative studies sometimes show that different definitions for the same terms make comparison of different studies difficult or impossible. For example, Shull *et al.* (2005) compare different studies about the presence of bugs. They observe that the content of “interface bug” has been defined differently in different studies, so it is hard to compare research results about interface bugs. Definitions in prior studies could be considered in order to improve comparability and understandability. If one wants to use a different definition than that in previous studies, it should be explained in the study.

As stated in the previous subchapter, too strict standardization may hide individual differences. So the general rule stated in the previous subchapter can be applied here, too: one should keep in mind individual differences and divert from general standards and definitions, but the diversions should be explained in the research document.

7.1.3 Research Areas of Subfields of Fault Elimination and Bug Knowhow

The main contribution of this thesis is to survey research about different fields of software fault elimination. A short summary of research done is given below. Bug knowhow gets special attention since it is an efficient way to eliminate bugs and not much attention is usually paid to it. Knowing about bugs helps stop repeating common faults. This subchapter is a summary of research that has been done in the field of fault elimination, and the next subchapter is about areas where more research should be done.

Subfields of Fault Elimination

Software faults can be eliminated by avoiding root causes, bugs, and bug interactions that appear frequently (chapter 2); taking fault prone features and characteristics of bugs into account (chapter 2); establishing a good software development process and improving it (chapter 3); performing risk analysis (chapter 3); applying appropriate methods during different phases of software development process (chapter 3); and analyzing (chapter 4), proving (chapter 4), and testing (chapter 5) software after each phase of software development cycle. As long as faults cannot be totally eliminated, means of fault tolerance are needed in order to prevent the harm (chapter 6). Plenty of research has been done about all these main topics. Chapters 2-6 survey research being done and discuss areas that do not get much attention. See also subchapter 7.1.4 for the most eyestriking areas that need more attention. In the following paragraphs, some main research areas are listed about each of those topics.

Bug Knowhow

In this thesis, bug knowhow is regarded as one subfield of fault elimination. It is investigated in chapter 2. There is research about characteristics of bugs, which fault types exist, fault classifications, fault proness, failure propagation, bugs types in specific applications and application domains, reasons for hidden bugs, number of faults in a failure situation, and root causes for faults. This research has been surveyed in this thesis.

There are different studies about fault proness. For example, some studies investigate factors of fault proness, e.g. features of fault prone software, fault prone features of programming languages, or fault-prone problems. There are also studies about fault proness of some specific language or environment, and about measuring and predicting fault-proness. There has been some effort in comparing different studies about fault proness and about fault types. Due to the lack on consistency in the studies, there have been problems in comparing different studies.

Because methodologies change, some bugs become more common or less common in the course of time. In addition, there are software bug types that have been repeated all the time. Frequent bug types are calculation and initializing bugs. Software has become more complex, so e.g. timing and interface faults have become more frequent. All situations should be taken into account in software development. Methods that guarantee this are becoming better and better. Anyway, missing software states have become a more common bug type. One reason may be that state spaces tend to be enormous, i.e. software has a large number of possible states. State space explosion is discussed in subchapter 7.1.4.

Some surveys have been performed about bug knowledge. There are surveys about fault proness, metrics for fault proness, typical faults in specific environments or application domains, number of faults in a failure situation, and precision of calculations.

Fault Prevention, Fault Prediction, Fault Detection, and Fault Tolerance

In the field of defect prevention and prediction, there is some research about fundamentals of software engineering, and issues like control and data representation. Graph theory is studied widely in software engineering research. There is research about processes and about assessing and improving a software development process. Methodologies get a lot of attention, so do models. There is research about risk analysis, risk representation, and requirement specification. All phases of software development life cycle get attention by research people. There is research about factors that have an effect on quality. Defect prediction gets a lot of attention. There is plenty of research involving metrics for development, for processes, and for defect prediction. Surveys have been done about general development like software engineering methodologies, control and data representation, software development processes, modelling, requirement analysis, process and quality metrics, defect prediction, and risk analysis. Small reviews have been done about some areas of foundations of software engineering.

In the field of checking and analysis, there is research about software inspection. There are also studies about limit analysis and about precision, and about modeling software and its environment. Flow analysis gets a lot of attention, usually connected with the function of compilers or with logic programming. There are some studies about flow analysis related to bug detection. There are studies about software states: for example, there are studies about state machines, often connected with model checking. There is research and development involving different logical systems for proving software, and methods for using those systems. Connections between logical methods and developing methodologies are also being studied to some extent. Termination of programs also gets attention. There are also other studies involving formal methods in diverse areas of software development. Each phase of software development life cycle has studies about applying formal methods, and many checks can be performed formally. Some topics for analysis surveys have been software inspection, software state description, state space explosion, data flow analysis, and proving methods.

In the field of testing, there are studies about choosing test cases. Early studies are more general and later studies are primarily method specific. Coverage criteria get a lot of attention, and test coverage in general is being studied. Relationships between coverage criteria are also studied a lot. There are lots of studies about testing methods. Some methods like performance testing have not got much attention before this decade. There is research about assessment of testing. Some topics for testing surveys have been test coverage, testing methods, and testing tools. Small surveys have been done about choosing test cases.

In the field of fault tolerance, there is a lot of research about configurations with full replication, and about modeling different configurations. N-version systems with full replication have got a lot of attention, so have recovery blocks. However, I have not found much research about reciprocal monitoring of different copies in N-version systems, but there may be some in other sources and with other keywords than the ones I have used. There is plenty of research about checks and diagnozing, but I have not seen so much research about assessing checks. Examples of reliability related surveys are surveys about fault tolerance in general, means of fault tolerance with configuration design and recovery blocks, fault diagnosis, and self-checks.

Temporal Development in the Field

There has been temporal development in the field, too. Some development studies like those that involve fundamentals of software development have been present during all decades, but some issues are more dominant during specific decades than during other ones. Here are some examples. Fundamental theorems were developed in 1960’s, many of path methodologies were developed from 1970’s to early 1980’s, and many new features and methodologies were developed during late 1980’s and 1990’s. At that time, comparative studies probably became more frequent. More and more attention has been paid to system factors in the 2000’s; however, there was some emphasis on them even during 1980’s if not earlier. More and more research is being done about software development processes. In the 1970's if not earlier, there have been studies about controlling development process by metrics, but more and more attention is being paid to it. More and more research is also being done about looking for features of fault proness. More generally, there is a trend towards statistic methods and looking for features e.g. by reverse engineering or cluster analysis. In the 2000’s more and more research has been done about integrating different methods and integration of different models, and about data mining. There is a relatively new trend towards adaptability.

Interrelationship between Fields of Fault Elimination

Fields of fault elimination are often interrelated. Integration between different fields of fault eliminations is a trend. For example, static analysis is sometimes compared to model checking, many tools have several tasks, and some methods and tools combine proving and testing.

7.1.4 Detecting Research Areas that Should Get More Attention

Each chapter of the thesis (chapters 2-6) covers a specific field of software fault elimination. Omitted research areas are mentioned in the appropriate chapters. A few omitted areas are briefly discussed in this subchapter.

The most eyestriking lack of research is about most software checking and analysis methods, except proving and testing methods. Flow analysis gets a lot of attention but it is probably due to the fact that it is used in compiler technology. It seems that studies of flow analysis generally have the view of language analysis. It should be kept in mind that flow analysis can also be used in bug elimination; some means to detect bugs with flow analysis are surveyed in this thesis. Formal software inspection gets plenty of attention. Less formal methods like time and size analysis, limit analysis, precision analysis, algorithm analysis, core review, code walkthrough, and comment analysis, are often easier to use and very efficient but do not get as much attention by developers or researchers as formal inspection. Precision analysis gets some attention but the view is often related to implementation. To some extent, the lack of research of these less formal methods may be due to the fact that there is a myth that only testing belongs to quality assurance. However, quality assurance also contains many other fields of bug elimination and many other areas than bug elimination.

One area where not much research has been done is partial replication or weak replication of software, as a lightweight method to increase fault tolerance. For example, difficult modules could be replicated, calculations could be done in several ways, repeated data structures could be used, or there could be redundancy inside a data structure. Instead, total replication by N-version software has got a lot of attention. However, I have not found much research about N-version systems with reciprocal monitoring, but there is probably research about it in other sources like architecture-related journals, and with other keywords than the ones I have used. Other lightweight methods to increase reliability, like safety margins, have not got a lot of attention either. Also, one possible area for more research is the assessment of software self-checks. This lack of lightweight method research may also be due to the fact that one thinks that QA is testing only. Because one only thinks of testing, one pays attention to fault tolerance only in extreme cases where there is a need for full replication.

State space explosion gets a lot of attention in relation to distributive and concurrent systems, but the problem is much more general. Not enough attention is being paid to lightweight methods against state space explosion, like state modularity and lightweight elimination of unnecessary states, and in including only desired states in software, even though lightweight methods could be easily used as a partial solution. Many states could be eliminated in early phases of software development by making the software e.g. go to error state in some situations. In addition, inappropriate input should be eliminated in as early a phase as possible. For example, new input could be asked for if the piece of software is interactive and the user input is out of range or illogical; this is an easy way to eliminate lots of erroneous states. Means to fight state explosion problem, e.g. in graphs and state machines, have been developed, and they are often based on abstraction. Also, simple methods like encapsulation and modularization often help. There are surveys about state space explosion. Some methods are being studied in detail but since the topic is wide, there is room for more research.

Correlation of different bug types should also be studied more; knowing the correlation helps avoiding those bugs. So more research could be done about which bug types correlate with each other. However, abstract models of fault correlation and number of faults in a failure situation have got plenty of attention by researchers. Plenty of attention has been paid to fault proness in general.

Characteristics of bug types and their temporal development have not been studied very much. Subchapter 2.1.3 contains a small analysis about temporal development of bug types, but the sample is small and more research could be done about the topic. Also, research about characteristics of individual bug types would help in understanding and eliminating them. There have been some efforts to find fault patterns by comparing different studies, comparing them has turned out to be problematic; Shull *et al.* (2005) is discussed in subchapter 7.1.2. There are other methods, too, to look for fault patterns. This thesis surveys some statistical and other methods to look for fault proness. How much these methods and other means help in detecting characteristics of faults could be studied. As stated in the summary of chapter 2, knowing characteristics of an individual bug type could help detect fault patterns, and vice versa.

7.1.5 Encouraging People for Early Elimination of Faults

In many studies and coursebooks it has been recognized that the earlier the faults be detected, the less costly they are. Regardless of that, too little attention has been paid to eliminating faults during development instead of testing. Different kinds of checks have been developed; these checks would help early fault elimination. Those checks are not performed very often, and only little research is being done about these checks compared to research involving testing. In addition, attention is being paid to the testing of special items and other unusual situations. However, only few studies involve paying attention to unusual situations in development and not only during testing.

The related problem is that many methods are used in testing only, although their use in development could result in earlier elimination of faults. A lot of research has been done about how to apply domain theory and category partitioning in testing phase. Similar approaches should be applied in all phases of software life cycle. For example in coding, category tables should be done about equivalent classes of values for different variables. It would help take all situations into account when writing software code. Now those tables are built only when testing the code, if at all. Nor is much attention paid to reduction of state space by ruling out impossible states before making a more detailed partition, see also the subchapter 7.1.4. In addition, path, branch, and data flow analysis in design and coding phases could be useful. This analysis is not performed very often. In the rare cases where it has been performed, it has been performed only during testing of code that had already been developed.

Methods presented in chapters 4 and 5 could be applied both during development and during testing. They should be applied as early as practical. Testing is recommended in all phases of software life cycle, but it is often executed only after all development work.

7.2 Pros and Cons of Different Fields of Software Fault Elimination

Each field of software fault elimination is useful but none is sufficient. The benefits of using processes, models, and methodologies have been discussed in chapter 3. Table 27 presents some benefits and insufficiencies of the other fields of fault elimination.

Table 27. Evaluation of fields of fault elimination

Field	**Benefits**	**Why is this insufficient alone**
Bug knowhow	+ A lot of prior information available + Special analysis of rare situations	- Knowledge is inherently used all the time within and besides concrete means like those in the following rows
Analysis, checks (rigorous proving excluded)	+ Unpredicted facts and fault types may be detected + Includes rare situations + Some methods have complete coverage + Some checking methods are good ways to counteract state space explosion + Many methods are easy to use	- State space explosion problems - External activities are usually ignored
Rigorous proving	+ Reliable (co-incidental correctness and coincidental equivalence are rare) + Covers the whole of what is being proved + All abstraction levels are possible	- Works only inside strict restrictions (e.g. scope of the system, correct initial conditions) - External activities are ignored - Takes time to use - Takes time to learn - Requires formalization
Testing	+ May reveal outside activities + Tester gets holistic feel of software and its use + Often easy to use	- State space explosion problems - Unusual situations hard to detect - Covers only cases that are being tested
Building fault tolerance	+ Unpredicted facts are covered + Rare situations are usually covered + External factors are covered	- When using probabilities, they must be small so that the combined probability is small - Common mode failures are possible - Does not remove all combinations of independent failures - Heavyweight means like full replication use resources

7.3 Recommendations

Table 28 contains recommendations for software developers, scientists, and teachers in order to eliminate software failures. By following these recommendations, one can understand failure elimination better, create more consistent development framework, pay attention to forgotten research areas, stop repeating the same bugs all the time, and use all available methods of fault elimination in as early a phase as practical. There recommendations also help people use information and methods outside their own research or development areas.

Table 28. Recommendations for software developers, scientists, and teachers

For software developers
Increase your bug knowhow, use information in chapter 2.
Stop repeating the same bugs all the time; pay attention to known bug types.
Pay attention to best practices, common terms, and good methods for software development, and if practical, be consistent with those. Use information in chapter 3.
Do not forget analysis methods like algorithm and limit analysis.
Perform analysis and proving, not only testing.
Use methods intended for analysis, testing, and proving during both development and testing. If practical, use methods intended for testing when analyzing software. Use information in chapters 4-5.
Use methods for eliminating state space explosion outside distributive systems, too.
Execute testing during and after each phase of software life cycle, not only as the last phase before and during implementation.
Consider fault tolerance, use information in chapter 6. Keep in mind lightweight methods like partial replication, repetition of data structures, repeated computations, and safety margin. Take backup copies of your software.
For research scientists
Pay attention to best practices, common terms, and good methods for software development, and if practical, be consistent with them. Use information in chapter 3.
Use terms consistently, so that studies are found easier and understood better.
Consider using keywords related to bug elimination.
Figure out factors about making studies more comparable.
Keep in mind uncertainty factors in studies.
Pay attention to whether your research could also have applications in other areas and application domains, and/or be associated with other fault elimination methods or other environments, and if it is related to other phases of life cycle.
Consider fault elimination methods, fault characteristics, problems, and the like from other research areas and from other application domains or environments or life cycle phases. They might affect your research area, too. If your study involves fault elimination methods, consider fault characteristics and problems from studies of other methods.
Consider cross-field research and common features of your method/problem with other ones.

Continued on the next page

Temporal development of bug types, characteristiscs of each bug type, and bug type correlation are candidate topics for research.
Consider doing research about the effectiveness of the forgotten static analysis methods like time, size, and precision analysis, algorithmic analysis, code reading, or code walkthrough.
Keep in mind that state space explosion is a global problem, not only related to distributive systems. Consider research about lightweight methods to fight state space explosion.
Pay attention to doing analysis and proving earlier than during coding or testing phase.
Continue developing simple proving methods.
Consider research about lightweignt methods for fault tolerance like partial software replication, safety margins, and assessing self-checks.
This survey may help in choosing research issues. For example, several lists and tables in this thesis are candidates for surveys, and more extensive surveys could be done on topics that have been surveyed in this thesis.
For teachers
Include bug knowhow in your courses.
Encourage students to stop repeating the same bugs all the time.
Encourage students to use best practices, common terms, and good methods for software development. Also, encourage them to use analysis, testing, proving, and using reliability means like partial replication in everyday life.
Encourage students to early elimination of bugs.
This survey may help in planning courses.

7.4 Problems in Doing this Thesis and Recommendations for Further Work

This thesis consists of a wide survey about the field of software fault elimination. Because the review is wide, it had been *impossible to do deep reviews* about any subtopics. Also, because the survey has a broad content, a high abstraction level follows.

General surveys about fault elimination have been performed several decades ago when the field did not have many subfields, and those surveys are relatively short. Since then, the field has extended significantly. Lots of surveys are being performed all the time about many topics in the field, see subchapter 7.1.3. Some of those surveys are very narrow and some involve wide area like testing methods, testing tools, or fault tolerance. There are also topics with several small reviews, like some areas of foundations of software engineering, or choosing test cases. There are *so many surveys that it is impossible to give a collection of them in this thesis*.

One problem has been that because the topic is wide, there is plenty of research about it. The time to do this thesis has been limited, so only a small part of existing research could have been investigated. I have limited my research to some most well known journals and online searches for some well known fault related terms, and on what I know based on my previous experience. There is a *risk that some common research topics have been left outside* this review since almost all research of those topics has been published in some other documents. In addition, there may be studies that use other keywords than what I have used, particularly when the usage of terms is inconsistent. Also, there can be subfields that are so *narrow that they easily get unnoticed*. In addition, the publications that I have used usually do *not contain studies with qualitative methods*, so it would generally have been impossible to include organizational studies in the thesis. Another risk is that some topics have connection to fault avoidance and that connection is *hard to observe* and has not been noticed.

There may be opportunities for completing this survey by adding areas where a significant amount of research has been done but *that have been omitted from this thesis*. Also in the more detailed level, there are some *areas where more surveys* could be done. For example, static checks and software analysis could be surveyed in detail. Even though these fields are not being studied a lot, they are broad and surveys could be done about them. Also, this thesis contains short surveys and tables about several topics; these surveys and tables could be extended with more thorough surveys. For example, this thesis surveys formal software engineering methods and temporal development of software fault types; these surveys could be extended. Generally speaking, there are surveys about areas where plenty of research has been done. In subchapter 7.1.4, fields are discussed that do not get very much attention by research people. *More research could be done about those topics*. The scope of this thesis could also be widened: *managerial, organizational, and economical means for failure elimination could also be included in a general survey*, like the connections between process and projects, and managerial views of process improvement.

8 CLOSURE

Plenty of research has been done about different fields of software fault elimination. Chapters 2-6 of this thesis survey research about defect prevention and prediction, bug knowledge, checks, testing, and fault tolerance, respectively. Each of those chapters has a summary in the end. Subchapter 7.1.3 contains a small summary about the main research areas.

However, there are some areas that lack research almost entirely even though they can be efficient means to avoid bugs. The most eyestriking ones are the field of other analytical reviews than software inspection (e.g. algorithm analysis, analyzing limits, and code review); characteristics of different bug types; correlations between different bug types and its avoidance; temporary development of bug types; lightweight methods for software to increase reliability like partial replication, safety margins, and self-check assessment; and some lightweight methods to reduce state space explosion, e.g. exclusion of unnecessary states. More research could be done about those domains.

Some of those domains could be surveyed more, too, like those in the field of checks and analytical reviews. This thesis contains some analysis about temporal development of bug types, but a more thorough analysis could be done about the topic, using a larger sample and possibly several sources. This survey contains small surveys and tables that could be extended by performing thorough surveys. One example is formal software engineering methods. Also, due to limited resources for this thesis, there can be research areas that exist but have not been found. So it is possible that the survey could be completed by adding missing research areas. Also, new research is being done all the time, so this survey could be completed. The survey could also be broadened: for example, short surveys about organizational, managerial, and economical means of failure improvement could be included.

The field also lacks structural framework. The table of contents of the thesis could be regarded as a basis for structural framework. Particularly, the field of analysis and checking could get its position in the framework. Structural framework helps in increasing consistency, e.g. in building glossaries of terms; consistency improves comparability and understandability of documents. Structural framework could also help people find research results. For example, if studies have quality assurance related keywords in addition to keywords related to their application domains, the results can be found and used when working with other application domains. Framework also helps in referring to those research domains outside quality assurance where studies typically have hints about improving software quality.

Also, there should be more definitions of terms, and they should be used more consistently. However, one should keep in mind that sometimes there is a need to use a term in a different way as before. More generally, it should be considered if recommendations could be established to improve comparability of research results, since different studies have not always been comparable. If such recommendations are established, it should be kept in mind that situations differ and there may be occasional need for diversion from those recommendations.

Some problems and areas of fault elimination are studied only in relation to a specific method, although the same problems occur when one does not use those methods. Some problems, means, and methods are studied only in connection with a specific application domain, even though the problems, means, and methods are more general. One should broaden the scope of research of those problems. Also, there should be more cross-field research since methods in different fields are often studied separately even when they have common features.

Very often, one associates quality assurance only with testing. One should keep in mind other efficient means to eliminate failures: one should know about bugs to stop repeating them, the organization often needs a development process and framework, defect prediction should be performed, software should be analyzed and often even proven in addition to testing, and failures could be prevented by partial or total replication of systems and software. Unfortunately, many checks and analysis methods that could already be used in development phases are used only in testing phase when all code has been written. One should do analysis and testing in as early a phase of software development as possible.

So quality assurance needs to be kept in mind all the time during software development, and the above mentioned methods should be used when possible. When doing quality assurance related research and development, one should keep in mind that quality assurance should be part of the development all the time. As stated above, one should also keep in mind that quality assurance methods and research results could often be used with many more methods and application domains than what they are used now. Where a specific method or methodology could be used can also be a topic for research.

This thesis is intended to help everyone take into account all areas of software failure elimination. Subchapter 7.3 contains recommendations for researchers, developers, and teachers. I hope this thesis helps researchers in choosing their research topics, using appropriate keywords, and keeping in mind that results of studies could be used in other contexts, too, and research of other topics may contain elements of failure elimination. Methods in different areas may have common features. In addition, problems in other fields could be present in your field, and research of other fields of fault elimination could be used in your research. Developers could use this thesis as a guide for different means for eliminating bugs. This thesis is also intended to be a framework for teachers in choosing course topics. Much of the information in this thesis will be used in my draft book about software failures.

REFERENCES

Abbott, R.J. 1990. Resourceful Systems for Fault Tolerance, Reliability, and Safety. ACM Computing Surveys, 22(1):35-68.

Abdurazik, A. & Ammann, P. & Ding, W. & Offutt, J. 2000. Evaluation of Three Specification-Based Testing Criteria. Proceedings of the Sixth International Conference on Engineering of Complex Computer Systems, 11-14 Sep, 2000. Pages 179-187. DOI 10.1109/ICECCS.2000.873943.

Adler, P. 1998. Apollo 11 Program Alarms. In: Jones, E.M. & Glover, K. (eds.): Apollo 11 Lunar Surface Journal [e-journal]. NASA.
www.hq.nasa.gov/office/pao/History/alsj/a11/a11.1201-pa.html
Web page by Gordon Roxburgh
Retrieved: 3 Nov, 2007.

Adrion, W.R. & Branstad, M.A. & Cherniavsky, J.C. 1982. Validation, Verification, and Testing of Computer Software. ACM Computing Surveys, 14(2): 159-192.

Aho, A.V. & Hopcrpft, J.E. & Ullman, J.D. 1983. Data Structures and Algorithms. Addison-Wesley Series in Computer Science and Information Processing. 427 pages. ISBN 0-201-00023-7.

Akama, S. 1995. Three-Valued Constructive Logic and Logic Programs. Proceedings of the 25th International Symposium on Multiple-Valued Logic, 23-25 May, 1995. IEEE. Pages 276-281. DOI 10.1109/ISMVL.1995.513543.

Al-Saqabi, K. & Saleh, K. & Ahmad, I. 1996. Recovery from Concurrent Failures in Communication Protocols. Journal of Systems and Software, 35(1):55-65.

Alagar, S. & Venkatesan, S. 2001. Techniques to Tackle State Explosion in Global Predicate Detection. IEEE Transactions on Software Engineering, 27(8):704-714.

Alkadi, I.S. & Alkadi, G.S. 2001. Algorithms that Compute Test Drivers in Object Oriented Testing. IEEE Proceedings of Aerospace Conference, 10-17 Mar, 2001. Volume 1. Pages 115-119. DOI 10.1109/AERO.2001.931700.

Allen, F.E. & Cocke, J. 1976. A Program Data Flow Analysis Procedure. Communications of the ACM 19(3): 137-147.

Alves-Foss, J. & Taylor, C. & Oman, P. 2004. A Multi-Layered Approach to Security in High Assurance Systems. Proceedings of the 37th Annual Hawaii International Conference on System Sciences, Big Island, Hawaii, USA, 5-8 Jan, 2004. 10 pp. DOI 10.1109/HICSS.2004.1265709.

Alur, R. & Benedikt, M. & Etessami, K. & Godefroid, P. & Reps, T. & Yannakakis, M. 2005. Analysis of Recursive State Machines. ACM Transactions on Programming Languages and Systems, 27(4):786-818.

Alur, R. & Etessami, K. & La Torre, S. & Peled, D. 2001. Parametric Temporal Logic for Model Measuring. ACM Transactions on Computational Logic 2(3):388-407.

Alur, R. & McMillan, K. & Peled, D. 2005. Deciding Global Partial-Order Properties. Formal Methods in System Design, 26(1):7-25. Springer.

Amland, S. 2000. Risk-Based Testing: Risk Analysis Fundamentals and Metrics for Software Testing Including a Financial Application Case Study. Journal of Systems and Software 53(3):287-295.

Ammann, P.E. & Knight, J.C. 1988. Data Diversity: An Approach to Software Fault Tolerance. IEEE Transactions on Computers, 37(4):418-425.

Amme, W. & Zehender, E. 1997. Effective Calculation of Data Dependences in Programs with Pointers and Structures. Proceedings of the 23rd EUROMICRO Conference 'New Frontiers of Information Technology', 1-4 Sep, 1997. IEEE. Pages 55-62. DOI 10.1109/EURMIC.1997.617216.

Andrews, J.H. & Yingjun Zhang, Y. 2003. General Test Result Checking with Log File Analysis. IEEE Transactions on Software Engineering, 29(7):634-648.

Appel, A.W. 2001. Foundational Proof-Carrying Code. Proceedings of the 16th Annual IEEE Symposium of Logic in Computer Science, Boston, MA, USA, 6-19 Jun, 2001. Pages 247-256. DOI 10.1109/LICS.2001.932501.

Arida, M. 1999. HINTS & POINTERS for PROS users. Subject: ROSAT Status Report #178. HRI Aspect Time Bug in SASS Processing Prior to SASS7_B. [e-message on message board]. 19 Feb, 1999. HEASARC (High Energy Astrophysics Science Archive Research Center, a partnership between NASA Gobbard Centerin Astrophysics Divisionin and High Energy Astrophysics Division of the Smithsonian Astrophysical Observatory (Cambridge, MA)) Mailing List Archives.
http://heasarc.gsfc.nasa.gov/mail_archive/rosnews/msg00123.html
Retrieved: 3 Nov, 2007.

ARM. 2003. RealView Compilation Tools. Version 2.0. Compiler and Libraries Guide. ARM DUI 0205B.
http://rtds.cs.tamu.edu/web_462/techdocs/ARM/swdev/DUI0205B_rvct_2_0_compiler_libraries.pdf
Retrieved: 29 Sep, 2008.

Armstrong, J.M. & Paynter, S.E. 2006. The Deconstruction of Safety Arguments through Adversarial Counter-Argument. University of Newcastle upon Tyne, UK, School of Computing Science, CS-TR-832. Mar, 2004.

Arora, A. & Kulkarni, S.S. 1998a. Component Based Design of Multitolerant Systems. IEEE Transactions on Software Engineering, 24(1):63-78.

Arora, A. & Kulkarni, S.S. 1998b. Designing Masking Fault-Tolerance via Nonmasking Fault-Tolerance. IEEE Transactions on Software Engineering, 24(6):435-450.

Atlee, J.M. & Gannon, J. 1993. State-Based Model Checking of Event-Driven System Requirements. IEEE Transactions on Software Engineering, 19(1):24-40.

Austin, T.M. & Breach, S.E. & Sohi, G.S. 1994. Efficient detection of all pointer and array access errors. ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, Orlando, FL, USA. ACM Press, New York, NY, USA. 29(6):290-301. ISBN: 0-89791-662-X.

Avižienis, A. 1995. Dependable computing depends on structured fault tolerance. IEEE Proceedings of Sixth International Symposium on Software Reliability Engineering, Toulouse, France, 24-27 Oct. 1995. IEEE Conference Proceedings. Pages:158 - 168. ISBN 0-8186-7131-9.

Avižienis, A. & Laprie, J.-C. & Randell, B. & Landwehr, C. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEEE Transactions on Dependable and Secure Computing, 1(1):11-33.

Avritzer, A. & Kondek, J. & Liu, D. & Weyuker, E.J. 2002. Software Performance Testing based on Workload Characterization. ACM Proceedings of the 3rd International Workshop on Software and Performance, Rome, Italy, 2002. SESSION: Performance Analysis in the Software Lifecycle. ACM Press, New York USA. Pages 17-24. ISBN 1-58113-563-7.

Avrunin, G.S. & Gorbett, J.C. & Dillon, L.K. 1998. Analyzing Partially-Implemented Real-Time Systems. IEEE Transactions on Software Engineering, 24(8):602-614.

Babich, A.F. 1979. Proving Total Correctness of Parallel Programs. IEEE Transactions on Software Engineering, 5(6): 558-574.

Bachelder, E. & Leveson, N. 2001. Describing and Probing Complex System Behavior: A Graphical Approach. SAE Transactions, 110(1):263-273. American Technical Publishers LTD. ISSN 0096-736X.

Badendregt, H.P. 1992. Lambda Calculi with Types. In: Abramsky, S. & Gabbay, D.M. & Maibaum, T.S.E. (eds.): Handbook of logic in computer science. Vol. 2. Pages 117-309. ISBN 0198537611.

Badri, L. & Badri, M. & St-Yves, D. 2005. Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique. 12th Asia-Pacific Software Engineering Conference, 15-17 Dec, 2005. IEEE Conference Proceedings. Pages 9-. DOI 10.1109/APSEC.2005.100.

Bai, C. & Cai, K.-Y. & Chen, T.Y. 2003. An Efficient Defect Estimation Method for Software Defect Curves. Proceedings of the 27th Annual International Computer Software and Applications Conference, 3-6 Nov, 2003. IEEE. Pages 534-539.

Baker, H.G. 1991. Pragmatic Parsing in Common Lisp: or, Putting defmacro in Steroids. ACM SIGPLAN Lisp Pointers, IV(2): 3-15.

Ball, T. & Naik, M. & Rajamani, S.K. 2003. From Symptoms to Cause: Localizing Errors in Counterexample Traces. ACM Proceedings of the 30th ACM AIGPLAN-SIGACT Symposium on Principles of Programming Languages, 15-17 Jan, 2003. ACM SIGPLAN Notices, 38(1):97-105. ISSN 036-1340.

Bao, Y. & Sun, X. & Trivedi, K.S. 2005. A Workload-Based Analysis of Software Aging, and Rejuvenation. IEEE Transactions on Reliability, 54(3):541-548.

Barber, K.S. & Graser, T. & Holt, J. 2003. Evaluating Dynamic Correctness Properties of Domain Reference Architectures. Journal of Systems and Software, 68(3):217-231.

Baresi, L. & Pezzè. M. 1998. Towards Formalizing Structured Analysis. ACM Transactions on Software Engineering and Methodology 7(1):80-107.

Baroni, P. & Guida, G. & Zanella, M. 2001. GART: A Tool for Experimenting with Approximate Reasoming Models. Expert Systems with Applications, 21(1):15-30.

Barton, P.I. 2000. Modeling, Simulation, and Sensitivity Analysis of Hybrid Systems. IEEE International Symposium on Computer-Aided Control System Design, 25-27 Sep, 2000. Pages 117-122. DOI 10.1109/CACSD.2000.900197.

Basili, V.R. & Abd-El-Hafiz, S.K. 1996. A Method for Documenting Code Components. Journal of Systems and Software, 34(2):89-104.

Bastani, F.B. 1985. On the Uncertainty in the Correctness of Computer Programs. IEEE Transactions on Software Engineering, SE-11(9):857-864.

Bastani, F.B. & Yen, I.-L. & Chen, I.-R. 1988. A Class of Inherently Fault Tolerant Distributed Programs. IEEE Transactions on Software Engineering 14(10):1432-1442.

Beauvais, J.-R. & Rutten, E. & Gautier, T. & Houdebine, R. & Guernic, P.L. & Tang, Y.-M. 2001. Modeling Statecharts and Activitycharts as Signal Equations. ACM Transactions on Software Engineering and Methodology, 10(4):397-451.

Beckman, N.E. 2006. A Survey of Methods for Detecting Race Conditions.
May 10, 2006.
http://www.cs.cmu.edu/~nbeckman/papers/race_detection_survey.pdf
Retrieved: 13 Oct, 2008.

Behr, A. & Camarinopoulos, L. 1997. Two Formulas for Computing the Reliability of Incomplete k-out-of-n:G Systems. IEEE Transactions on Reliability, 46(3):421-429.

Belkhouche, B. & Geraci, B.J. 1996. A Formally Specified Prototyping System. Journal of Systems and Software, 34(1): 67-81.

Bellini, P. & Mattolini, R. & Nesi, P. 2000. Temporal Logics for Real-Time System Specification. ACM Computing Surveys, 32(1): 12-42.

Bellman, K.L. & Landauer, C. 1995. Designing Testable, Heterogeneous Software Environments. Journal of Systems and Software, 29(3):199-217.

Benander, A.C. & Benander, B.A. & Sang, J. 2000. An Empirical Analysis of Debugging Performance - Differences Between Iterative and Recursive Constructs. Journal of Systems and Software, 54(1):17-28.

Bennani, M.N. & Menascé, D.A. 2004. Assessing the Robustness of Self-Managing Computer Systems under Highly Variable Workloads. Proceedings of the International Conference on Autonomic Computing, 17-18 May, 2004. IEEE. Pages 62-69. DOI 10.1109/ICAC.2004.1301348.

Bergadano, F. & Gunetti, D. 1996. Testing by Means of Inductive Program Learning. ACM Transactions on Software Engineering and Methodology, 5(2):119-145.

Bergeron, J. & Debbabi, M. & Desharnais, J. & Erhioui, M.M. & Lavoie, Y. & Tawbi, N. 2001. Static Detection of Malicious Code in Executable Programs. International Journal of Requirements Engineering (2001):184-189.

Berglund, E. 2005. Communicating Bugs: Global Bug Knowledge Distribution. Information and Software Technology 47(11):709-719.

Berman, O. & Kumar, U.D. 1998. Reliability Analysis of Communicating Recovery Blocks. IEEE Transactions on Reliability 47(3): 245-254.

Bernardeschi, C. & Bondavalli, A. & Csertán, G. & Majzik, I. & Simoncini, L. 1998. Temporal Analysis of Data Flow Control Systems. Automatica 34(2): 169-182.

Berry, G. & Boudol, G. 1989. The Chemical Abstract Machine. ACM Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, Dec 1989. ACM Press, New York, USA. Pages 81-94. ISBN: 0-89781-343-4.

Bertolino, A. & Strigini, L. 1996. On the Use of Testability Measures for Dependability Assessment. IEEE Transactions on Software Engineering, 22(2):97-108.

Bessiere, C. 2006. Constraint Propagation. University of Montpelllier, France. National Center of Scientific Research. Department of Information and Engineering Sciences and Technologies. Montpellier Laboratory of Computer Science, Robotics, and Microelectronics. Technical Report LIRMM 06020 CRNS. Mar 2006.
Link to Citeseer web page, where the pdf file can be loaded:
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.1676
Retrieved: 18 Jan, 2009.
In: Rossi, F. van Beek, P. & Walsh, T. (eds.). 2006. Handbook of Constraint Programming . Chapter 3. Elsevier. Amsterdam, Netherlands. 978 Pages. ISBN 0-444-52726-5. ISBN 978-0-444-52726-4.

Beyer, D. & Noack, A. & Leverentz, C. 2005. Efficient Relational Calculation for Software Analysis. IEEE Transactions on Software Engineering, 31(2):137-149.

Bhargava, B. & Lian, S.-R. & Leu, P.-J. 1990. Experimental Evaluation of Concurrent Checkpointing and Rollback-Recovery Algorithms. Proceedings of the sixth International Conference on Data Engineering. 5-9 Feb, 1999. IEEE. Pages 182-189. DOI 10.1109/ICDE.1990.113468.

Bidoit, M. & Biebow, B. & Gaudel, M.-C. & Gresse, C. & Guiho, G.D. 1985. Exception Handling: Formal Specification and Systematic Program Construction. IEEE Transactions on Software Engineering, 11(3): 242-252.

Bieman, J.M. & Kang, B.-K. 1998. Measuring Design-Level Cohesion. IEEE Transactions on Software Engineering, 24(2): 111-124.

Bieman, J.M. & Straw, G. & Wang, H. & Munger, P.W. & Alexander, R.T. 2003. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. Proceedings of the Ninth International Software Metrics Symposium, 3-5 Sep, 2003. IEEE Computer Society, Washington DC, USA. Page 40. ISBN 0-7695-1987-3.

Binkley, D. & Harman, M. 2004. Analysis and Visualization of Predicate Dependence on Formal Parameters and Global Variables. IEEE Transactions on Software Engineering, 30(11):715-735.

Birman, A. & Joyner, W.H., Jr. 1976. A Problem-Reduction Approach to Proving Simulation Between Programs. IEEE Transactions on Software Engineering, SE-2(2):87-96

Bisant, D.B. & Lyle, J.R. 1989. A Two-Person Inspection Method to Improve Programming Productivity. IEEE Transactions on Software Engineering, 15(10): 1294-1304.

Bishop, P.G. 2002. Rescaling Reliability Bounds for a New Operational Profile. ACM Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, Rome, Italy, 22-24 Jul. SESSION: Theory of Testing and Reliability. ACM Press, New York, USA. 27(4), pages 180-190. ISBN ISSN 0163-5948, 1-58113-562-9.

Bishop, P.G. 2006. Review of Software Design Diversity. 1. Aug, 2006. Adelard LLP, London, England.
In: Lyu, M. (ed.): Software Fault Tolerance. Wiley Press, New York, NY, USA. 354 pages. ISBN 0471950688.
http://www.adelard.co.uk/resources/papers/pdf/divchap.pdf
Retrieved: 9 May, 2008.

Bishop, P.G. & Bloomfield, R.E. 2003. Using a Log-Normal Failure Rate Distribution for Worst Case Boundary Reliability Prediction. ISSRE 2003, Denver, Colorado, USA, 17-20 Nov, 2003. Pages 237-245.

Bistarelli, S. & Montanari, U. & Rossi, F. 1997. Semiring-Based Constraint Satisfaction and Optimization. Journal of the ACM, 44(2): 201-236.

Biswas, S. & Rajaraman, V. 1987. An Algorithm to Decide Feasibility of Linear Integer Constraints Occurring in Decision Tables. IEEE Transactions on Software Engineering, SE-13(12):1340-1347.

Blanchet, B. & Cousot, P. & Cousot, R. & Feret, J. & Mauborgne, L. & Miné, A. & Monniaux, D. & Rival, X. 2003. A Static Analyzer for Large Safety-Critical Software. ACM SIGPLAN Notices, 38(5): 196–207.

Blass, A. & Gurevich, Y. 2001. Inadequacy of Computable Loop Invariants. ACM Transactions on Computational Logic, 2(1):1-11.

Blum, M. & Kannan, S. 1995. Designing Programs that Check Their Work. Journal of the ACM, 42(1):269-291.

Blute, R. & Scott, P. 2003. Category Theory for Linear Logicians. University of Ottawa, Department of Mathematics. Ottawa, Ontario, Canada.
http://www.site.uottawa.ca/~phil/papers/catsurv.web.pdf
Retrieved: 25 Oct, 2008.
Final version in: Linear Logic in Computer Science, LMS Lecture Note Series 316, 2004, Cambridge University Press.

Bobbio, A. & Franceschinis, G. & Gaeta, R. & Portinale, L. 2003. Parametric Fault Tree for the Dependability Analysis of Redundant Systems and its High-Level Petri Net Semantics. IEEE Transactions on Software Engineering 29(3):270-287.

Boehm, B. 1981. Software Engineering Economics. Advances in Computer Science and Technology. Advances in Computing Science and Technology Series. Prentice-Hall, New Jersey. 767 pages. ISBN-10 0138221227. ISBN-13 9780138221225.

Boehm, B.W. 1988. A Spiral Model of Software Development and Enhancement. IEEE Computer, 21(5):61-72.

Boland, P.J. & El-Neweihi, E. 1995. Component Redundancy *vs* System Redundancy in the Hazard Rate Ordering. IEEE Transactions on Reliability, 44(4): 614-619.

Bonsangue, M. 2001. Introduction. Revision of the introduction of author's PhD thesis. Electronic Notes in Theoretical Computer Science, 8: 4-13.

Boujarwah, A.S. & Saleh, K. & Al-Dallal, J. 2000. Dynamic Data Flow Analysis for Java Programs. Information and Software Technology, 42(11): 765-775.

Bourne, S. 2004. A Conversation with Bruce Lindsay: Designing for Failure May be the Key to Success. Interview. ACM Queue 2(2):22-23.

Boute, R. 2000. Supertotal Function Definition in Mathematics and Software Engineering. IEEE Transactions on Software Engineering, 26(7):662-672.

Bouziane, Z. 1998. A Primitive Recursive Algorithm for the General PETRI Net Reachability Problem. Proceedings of the 39th Annual Symposium on Foundations of Computer Science, 8-11 Nov, 1998. Pages 130-136. DOI 10.1109/SFCS.1998.743436.

Bowring, J.F. & Regh, J.M. & Harrold, M.J. 2004. Active Learning for Automatic Classification of Software Behaviour. ACM SIGSOFT Software Engineering Notes. Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, Massachusetts, USA, 11-14 Jul, 2004. Session: Program Analysis II. ACM Press, New York, USA. 29(4):195-205. ISSN 0163-5948.

Brader, M. 1987. Mariner 1. In: P.G. Neumann (Moderator). Risk Digest. Forum on Rules to Public in Computers and Related Systems. ACM Committee on Computers and Public Policy. 5(63).
http://catless.ncl.ac.uk/Risks
Retrieved: 24 Apr, 2007.

Bravetti. M. 2003. An Integrated Approach for the Specification and Analysis of Stochastic Real-Time Systems. Electronic Notes in Theoretical Computer Science, 68(5): 34-64.

Brestel, J. & Reghizzi, S.C. & Roussel, G. & Pietro, P.S. 2005. A Scalable Formal Method for Design and Automatic Checking of User Interfaces. ACM Transactions on Software Engineering and Methodology, 14(2): 124-167.

Briand, L.C. & Basili, V.R. & Hetmanski, C.J. 1993. Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software Components. IEEE Transactions on Software Engineering, 19(11):1028-1044.

Briand, L.C. & Basili, V.R. & Thomas, W.M. 1992. A Pattern Recognition Approach for Software Engineering Data Analysis. IEEE Transactions on Software Engineering 18(11):931-942.

Briand, L.C. & Di Penta, M. & Labiche, Y. 2004. Assessing and Improving State-Based Class Testing: A Series of Experiments. IEEE Transactions on Software Engineering, 30(11): 770-793.

Briand, L.C. & El Eman, K. & Freimut, B.G. & Laitenberger, O. 2000. Capture-Recapture Models for Estimating Software Defect Content. IEEE Transactions on Software Engineering 26(6):518-540.

Briand, L.C. & Labiche, Y. & Wang, Y. 2003. A Comprehensive and Systematic Methodology for Client-Server Class Integration Testing. Proceedings of the 14th International Symposium on Software Reliability Engineering, 17-20 Nov, 2003. IEEE. Pages 14-25. DOI 10.1109/ISSRE.2003.1251027.

Briand, L.C. & Melo, W.L. & Wust, J. 2002. Assessing the Applicability of Fault-Proness Models Across Object-Oriented Software Projects. IEEE Transactions on Software Engineering, 28(7):706-720.

Briand, L.C. & Wüst, J. & Daly, J.W. & Porter, D.V. 2000. Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. Journal of Systems and Software 51(3): 245-273.

Brilliant, S.S. & Knight, J.C. & Leveson, N.G. 1989. The Consistent Comparison Problem in N-Version Software. IEEE Transactions on Software Engineering, SE-15(11):1481-1485.

Brilliant, S.S. & Knight, J.C. & Leveson, N.G. 1990. Analysis of Faults in an N-Version Software Experiment. IEEE Transactions on Software Engineering, SE-16(2):238-247.

Brockmeyer, M. & Jahanian, F. & Heitmeyer, C. & Labaw, B. 1996. An Approach to Monitoring and Assertion-Checkin of Real-Time Specifications. Proceedings of the 4th International Workshop on Parallel and Distributed Systems, 15-16 Apr, 1996. Pages 236-243. DOI 10.1109/WPDRTS.1996.557687.

Bryant, R.E. 1992. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys 24(3):293-318.

Bucci, G. & Fedeli, A. & Sassoli, L. & Vicario, E. 2004. Timed State Space Analysis of Real-Time Preemptive Systems. IEEE Transactions on Software Engineering, 30(2):97-111.

Budd, T.A. & DeMillo, R.A. & Lipton, R.J. & Sayward, F.G. 1980. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. Proceedings of the 7th ACM SIGPLAN -SIGACT Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, 28-30 Jan, 1980. ACM Press, New York, USA. Pages 220-233. ISBN 0-89791-011-7.

Bukowski, J.V. & Goble, W.M. 1995. Using Markov Models for Safety Analysis of Programmable Electronic Systems. ISA Transactions 34(2): 193-198. Elsevier.

Bultan, T. & Gerber, R. & League, C. 2000. Composite Model Checking: Verification with Type-Specific Symbolic Representations. ACM Transactions on Software Engineering and Methodology, 9(1):3-50.

Bultan, T. & Gerber, R. & Pugh, W. 1999. Model-Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations, and Experimental Results. ACM Transactions on Programming Languages and Systems, 21(4):747-789.

Bunea, C. & Charitos, T. & Cooke, R.M. & Becker, G. 2005. Two-Stage Bayesian Models - Application to ZEDB Project. Reliability Engineering and System Safety, 90(2-3):123-130.

Burke, M.G. & Ryder, B.G. 1990. A Critical Analysis of Incremental Iterative Data Flow Analysis Algorithms. IEEE Transactions on Software Engineering, 16(7):723-728.

Butler, R.W. & Finelli, G.B. 1993. The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. IEEE Transactions on Software Engineering, 19(1):3-12.

Cai, X. & Lyu, M.R. & Vouk, M.A. 2005. An Experimental Evaluation on Reliability Features of N-Version Programming. IEEE Proceedings on the 16th IEEE International Symposium on Software Reliability Engineering, 8-11 Nov, 2005. IEEE Computer Society, Washington DC, USA. Pages 161-170. ISBN ISSN 1071-9458, 0-7695-2482-6.

Cain, T. & Park, E.K. 1996. Algebraic Software Testing in Vector Spaces of Functions. Proceedings of the 20th Conference on Computer Software and Applications, COMPSAC 1996, 19-23 Aug, 1996. IEEE Society, Washington DC, USA. Page 234. ISSN 0730-3157.

Canfora, G. & Cimitile, A. & De Carlini, U. & De Lucia, A. 1998. An Extensible System for Source Code Analysis. IEEE Transactions on Software Engineering, 24(9):721-740.

Cangussu, J.W. & DeCarlo, R.A. & Mathur, A.P. 2003. Using Sensibility Analysis to Validate a State Variable Model of the Software Test Process. IEEE Transactions on Software Engineering, 29(5):430-443.

Cannon, L.W. & Elliott, R.A. & Kirchoff, L.W. & Miller, J.H. & Milner, J.M. & Mitze, R.W. & Schan, E.P. & Whittington, N.O. & Spencer, H. & Keppel, D. & Brader, M. 1990. Recommended C Style and Coding Standards. Revision 6.0. 25 Jun, 1990. Converted to HTML: Selkirk, P. 10 Feb, 1995.
http://www.psgd.org/paul/docs/cstyle/cstyle.htm
Retrieved: 29 Sep, 2008.

Cant, S.N. & Jeffery, D.R. & Henderson-Sellers, B. 1995. A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. Information and Software Technology 37(7):351-362.

Card, D.N. 1998. Learning from Our Mistakes with Defect Causal Analysis. IEEE Software 15(1):56-63.

Card, D.N. & Church, V.E. & Agresti, W.W. 1986. An Empirical Study of Software Design Practices. IEEE Transactions on Software Engineering, SE-12(2):264-271.

Carrington, L.C. & Laurenzano, M. & Snavely, A. & Campbell, R.L. & Davis, L.P. 2005. How Well Can Simple Metrics Represent the Performance of HPC Applications? Proceedings of the ACM/IEEE SC 2005 Conference on Supercomputing, 12-18 Nov, 2005. DOI 10.1109/SC.2005.33.

Cartwright, R. & Felleisen, M. 1996. Program Verification through Soft Typing. ACM Computing Surveys 28(2):349-351.

Carver, R.H. 1996. Testing Abstract Distributed Programs and Their Implementations: A Constraint-Based Approach. Journal of Systems and Software, 33(3):223-237.

Casati, F. & Castano, S. & Fugini, M. & Mirbel, I. & Pernici, B. 2000. Using Patterns to Design Rules in Workflows. IEEE Transactions on Software Engineering, 26(8):760-785.

Cavalcanti, A. & Sampaio, A. & Woodcock, J. 1999. An Inconsistency in Procedures, Parameters, and Substitution in Refinement Calculus. Science of Computer Programming, 33(1): 87-96.

CEA LIST 2004. The Caveat Tool. Software Reliability Laboratory. CEA. France. 1 Sep, 2004.
http://www-list.cea.fr/labos/gb/LSL/caveat/index.html
Retrieved: 22 Oct, 2008.

Ceri, S. & Crespi-Reghizzi, S. & Di Maio, A. & Lavazza, L.A. 1988. Software Prototyping by Relational Techniques: Experiences with Program Construction Systems. IEEE Transactions on Software Engineering, 14(11):1597-1609.

Chan, W.K. & Chen, T.Y. & Tse, T.H. 2002. An Overview of Integration Testing Techniques for Object-Oriented Programs. Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science, International Association of Computer and Information Science, Mt. Pleasant, MI, USA. Pages 696-701.

Chang, C.-P. & Chu, C.-P. & Yeh, Y.-F. 2009. Integrating in-Process Software Defect Prediction with Association Mining to Discover Defect Pattern. Information and Software Technology, 51(2): 375-384.

Chang, K.-H. & Carlisle, W.H. & Cross, II, J.H. & Brown, D. 1991. A Heuristic Approach for Test Case Generation, Proceedings of the 19th Annual Conference on Computer Science, San Antonio, TX, USA. ACM. Pages 174-180. ISBN 0-89791-382-5.

Chang, K.H. & Liao, S.-S. & Seidman, S.B. & Chapman, R. 1998. Testing Object-Oriented Programs: From Formal Specification to Test Scenario Generation. Journal of Systems and Software, 42(2): 141-151.

Chang, R.-Y. & Podgurski, A. & Yang, J. 2008. Discovering Neglected Conditions in Software by Mining Dependence Graphs. IEEE Transactions on Software Engineering, 34(5): 579-596.

Chang, S.-K. & You, W.-T. & Hsu, P.-L. 1994. General-Structured Unknown Input Observers. IEEE Proceedings of the American Control Conference, Baltimore, MD, USA, 29 Jun - 1 Jul, 1994. IEEE. 1(29):666-670.

Chang, W.K. & Jeng, S.-L. 2005. Impartial Evaluation in Software Reliability Practice. Journal of Systems and Software, 76(2):99-110.

Charlesworth, A. 2002. The Undecidability of Associativity and Commutativity Analysis. ACM Transactions on Programming Languages and Systems, 24(5): 554-565.

Chen, I.-R. & Bastani, F.B. 1992. Reliability of Fully and Partially Replicated Systems. IEEE Transactions on Reliability, 41(2):175-182.

Chen, P.M. & Aycock, C.M. & Ng, W.T. & Rajamani, G. & Sivaramakrishnan, R. 1995. Rio: Storing Files Reliably in Memory. University of Michigan, USA, College of Engineering, Department of Electrical Engineering and Computer Science. CSE-TR 250-95.

Chen, T.Y. & Pak-Lok Poon, P.-L. & Tse, T.H. 2003. A Choice Relation Framework for Supporting Category-Partition Test Case Generation. IEEE Transactions on Software Engineering, 29(7):557-593.

Chen, T.Y. & Poon, P.L. & Tse, T.H. 1999. A New Restructuring Algorithm for Classification Tree Method. IEEE Proceedings of the Software Technology and Engineering Practice, 30 Aug – 2 Sep, 1999. Pages 105-114.

Chen, T.Y. & Yu, Y.T. 2001. On the Maximum Algorithms for Test Allocations in Partition Testing. Information and Software Technology 43(2):97-107.

Chen, Y. 1998. Modelling Software Operational Reliability via Input Domain-Based Reliability Growth Model. 28th Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, 23-25 Jun, 1998. IEEE. Pages 314-323. ISBN 0-8186-8470-4.

Chen, Y. & Probert, R.L. & Robeson, K. 2004. Effective Test Metrics for Test Strategy Evolution. Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research, Markham, Ontario, Canada, 4-7 Oct, 2004. IBM Press. Pages 111-123. ISSN 1705-7345.

Cheng, J. 2006. Legal Information Systems. Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, 23-27 Apr, 2006. Pages 319-320.

Cheng, Y.-P. & Young, M. & Huang, C.-L. & Pan, C.-Y. 2003. Towards Scalable Compositional Analysis by Refactoring Design Models. Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Helsinki, Finland, 1-5 Sep, 2003. ACM. Pages 247-256.

Cherniavsky, J.C. & Smith, C.H. 1987. A Recursion Theoretic Approach to Program Testing. IEEE Transactions on Software Engineering, SE-13(7):777-784.

Chesñevar, C.I. & Maguitman, A.G. & Loui, R.P. 2000. Logical Models of Argument. ACM Computing Surveys, 32(4): 337-383.

Cheung, S.C. & Kramer, J. 1994. An Integrated Method for Effective Behaviour Analysis of Distributed Systems. Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, 16-21 May, 1994. IEEE Computer Society Press, Los Alamitos, CA, USA. Pages 309-320. ISBN 0-8186-5855-X.

Cheung, S.C. & Kramer, J. 1996. Checking Subsystem Safety properties in Compositional Reachability Analysis. Proceedings of the 18th International Conference on Software Engineering, 25-30 Mar, 1996. Pages 144-154. DOI 10.1109/ICSE.1996.493410.

Chillarege, R. 1994. Self-Testing Software Probe System For Failure Detection and Diagnosis. Proceedings of the 1994 Conference of the IBM Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, 31Oct – 3 Nov, 1994. IBM Press. Page 10.

Chillarege, R. & Bhandari, I.S. & Chaar, J.K. & Halliday, M.J.& Moebus, D.S. & Ray, B.K. & Wong, M.-Y. 1992. Orthogonal Defect Classification - a Concept for In-Process Measurements. IEEE Transactions on Software Engineering, 18(11):943-956.

Chillarege, R. & Iyer, R.K. 1985. The Effect of System Workload on Error Latency. ACM SIGMETRICS Performance Measurement Review, Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Austin, Texas, USA, Aug 1985. 13(2):69-77. ISSN 0163-5999.

Chillarege, R. & Kao, W.-L. & Condit, R.G. 1991. Defect Type and its Impact on the Growth Curve. Proceedings of the 13th International Conference on Software Engineering, Austin, Texas, USA, 13-17 May, 1991. IEEE Computer Society Press, Los Alamitos, CA, USA. Pages 246-255. ISBN 0-89791-391-4.

Chou, A. & Yang, J. & Chelf, B. & Hallem, S. & Engler, D. 2001. An Empirical Study of Operating System Errors. Symposium on Operating System Principles, Banff, Alberta, Canada, 21-24 Oct, 2001. SESSION: Deconstructing the OS. ACM Press, New York, USA. Pages 73-88. ISBN 1-58113-389-8.

Chu, H.D. 1997. An Evaluation Scheme of Software Testing Techniques. IFIP TC5 WP5.4 3rd International Conference on Reliability, Quality and Safety of Software-Intensive Systems, Athens, Greece, 1997. Chapman & Hall, Ltd, London, UK. Pages 259-262. ISBN 0-412-80280-5.

Chung, C.-G. & Lee, J.-G. 1997. An Enhanced Zero-One Optimal Path Set Selection Method. Journal of Systems and Software, 39(2):145-164.

Chung, I.J. 1989. Improved Control Strategy for Parallel Logic Programming. IEEE International Workshop on Tools for Artificial Intelligence, 1989. Architectures, Languages, and Algorithms. 23-25 Oct, 1989. Pages 702-708. DOI 10.1109/TAI.19889.65384.

Chusho, T. 1987. Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing. IEEE Transactions on Software Engineering, SE-13(5):509-517.

Ciarambino, I. & Contini, S. & Demichela M. & Piccinini, N. 2002. How to Avoid the Generation of Loops in the Construction of Faulttrees. IEEE Proceedings of Annual Reliability and Maintainability Symposium, Seattle, WA, USA, 28-31 Jan, 2002. Pages 178-185. ISBN 0-7803-7348-0.

Clapp, K.C. & Iyer, R.K. & Levendel, Y. 1992. Analysis of Large System Black-Box Test Data. IEEE Proceedings on the third International Symposium of Software Engineering, Research Triangle Park, NC, USA, 7-10 Oct, 1992. Pages 94-103. ISBN 0-8186-2975-4.

Clark, K.L. & van Emden, M.H. 1981. Consequence Verification of Flowcharts. IEEE Transactions on Software Engineering, SE-7(1):52-60.

Clarke, E. & Lu, Y. & Grumberg, O. & Veith, H. & Jha, S. 2003. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. Journal of the ACM, 50(5): 752-794.

Clarke, L.A. & Hassell, J. & Richardson, D.J. 1982. A Close Look at Domain Testing. IEEE Transactions on Software Engineering, 8(4):380-390.

Clarke, S.J. & McDermid, J.A. 1993. Software Fault Trees and Weakest Preconditions: A Comparison and Analysis. Software Engineering Journal 8(4):225-236.

Cleland-Huang, J. & Marrero, W. & Berenbach, B. 2008. Goal-Centric Traceability: Using Virtual Plumblines to Maintain Critical Systemic Qualities. IEEE Transactions on Software Engineering, 34(5): 685-699.

Clermont, M. & Parnas, D. 2005. Using Information about Functions in Selecting Test Cases. ACM SIGSOFT Software Engineering Notes. SESSION: Advances in Model-Based Testing A-MOST 05. ACM Press, New York, NY, USA. 30(4):1-7. ISBN 1-59593-115-5.

Cohen, D.M. & Dalal, S.R. & Fredman, M.L. & Patton, G.C. 1997.: The AETG System: An Approach to Testing Based on Combinatorial Design. IEEE Transactions on Software Engineering, 23(7):437-444.

Cohen, D.M. & Dalal, S.R. & Kajla, A. & Patton, G.C. 1994. The Automatic Efficient Test Generator (AETG) System. IEEE Proceedings of the fifth International Symposium on Software Reliability Engineering, Monterey, CA, USA, 6-9 Nov, 1994. Pages 303-309. ISBN 0-8186-6665-X.

Collofello, J. & Vehathiri, K. 2005. An Environment for Training Computer Science Students on Software Testing. Proceedings of the 35th Annual Conference on Frontiers in Education, Indianapolis, IN, USA, 19-22 Oct, 2005. ASEE/IEEE. Pages T3E - 6-10. DOI 10.1109/FIE.2005.1611937.

Colman, A. & Han, J. 2007. Using Role-Based Coordination to Achieve Software Adaptability. Science of Computer Programming 64(2):223-245.

Conte de Leon, D. & Alves-Foss, J. 2006. Hidden Implementation Dependencies in High Assurance and Critical Computing Systems. IEEE Transactions on Software Engineering, 32(10):790-811.

Cook, J.E. & Wolf, A.L. 1999. Software Process Validation: Quantitavely Measuring the Correspondence of a Process to a Model. ACM Transactions on Software Engineering and Methodology 8(12): 147-176.

Coppit, D. & Jinlin Yang & Khurshid, S. & Wei Le & Sullivan, K. 2005. Software Assurance by Bounded Exhaustive Testing. IEEE Transactions on Software Engineering, 31(4):328-339.

Corbett, J.C. 1993. Identical Tasks and Counter Variables in an Integer Programming-Based Approach to Verification. Proceedings of the 7th International Workshop on Software Specification and Design, Redondo Beach, CA, USA, 6-7 Dec, 1993. SESSION: Real-time Systems. IEEE Computer Society Press, Los Alamitos, CA, USA. Pages 100-109. ISBN ISSN 1063-6765, 0-8186-4360-9.

Cousot, P. 1997. Types as Abstract Interpretations. Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. Pages 316-331.

Cousot, P. & Cousot, R. 1979. Systematic Design of Program Analysis Frameworks. Proceedings on the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Antonio, TX, USA, 29-31 Jan, 1979. ACM Press, New York, NY, USA. Pages 269-282.

Csenki, A. 1993. Reliability Analysis of Recovery Blocks with Nested Clusters of Failure Points. IEEE Transactions on Reliability, 42(1):34-43.

Cui, Q. & Gannon, J. 1992. Data-Oriented Exception Handling. IEEE Transactions on Software Engineering, 18(5): 393-401.

Cukic, B. 1997. Combining Testing and Correctness Verification in Software Reliability Assessment. Proceedings of the 2nd IEEE High-Assurance Systems Engineering Workshop, Washington DC, USA, 11-12 Aug, 1997. Pages 182-187. ISBN 0-8186-7971-9.

d'Amorim, M. & Lauterburg, S. & Marinov, D. 2008. Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs. IEEE Transactions on Software Engineering, 34(5): 597-613.

Dai, Y.S. & Xie. M. & Poh, K.L. & Ng, S.H. 2004. A Model for Correlated Failures in N-version Program. IIE Transactions on Quality and Reliability Engineering, 36(12):1183-1192.

Dalal, S.R. & Jain, A. & Karunanithi, N. & Leaton, J.M. & Lott, C.M. 1998. Model Based Testing of a Highly Programmable System, Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE), Paderborn, Germany, 4-7 Nov, 1998. IEEE Computer Society, Washington DC, USA. Pages 174-178. ISBN 0-8186-8991-9.

Dalal, S.R. & McIntosh, A.A. 1994. When to Stop Testing for Large Software Systems with Changing Code. IEEE Transactions on Software Engineering, 20(4):318-323.

Danicic, S. & Daoudi, M. & Fox, C. & Harman, M. & Hierons, R.M. & Howroyd, J.R. & Ourabya, L. & Ward, M. 2005. ConSUS: a light-weight program conditioner. Journal of Systems and Software, 77(3):241-262.

Dannenberg, R.B. & Ernst, G.W. 1982. Formal Program Verification Using Symbolic Execution. IEEE Transaction on Software Engineering, SE-8(1):43-52.

Dantsin, E. & Eiter, T. & Gottlob, G. & Voronkov, A. 1997. Complexity and Expressive Power of Logic Programming. Proceedings of the 12th Annual Conference on Computational Complexity, Ulm, Germany, 24-27 Jun 1997. Pages 82-101.

Darcy, J.D. 2006. What Every Computer Programmer Should Know About Floating-Point Arithmetic. [e-document]. 23. Jun, 2006. Darcy’s webblog at SUN Microsystems, INC.
http://blogs.sun.com/darcy/resource/Wecpskafpa-StanfordIcme500.pdf
Retrieved: 24 Mar, 2007.

Das, S.K. & Datta, A.K. & Tixeuil, S. 1996. Self-Stabilizing Algorithms in DAG Structured Networks. Proceedings of the Fourth International Symposium on Parallel Architectures, Algorithms, and Networks, 23-25 Jun, 1999. IEEE. Pages 190-195. DOI 10.1109/ISPAN.1999.778938.

Dawson, J.E. 2004. Formalising General Correctness. Electronic Notes in Theoretical Computer Science, 91:21-42.

De Florio, V. & Blondia, C. 2008. A Survey of Linguistic Structures for Application-Level Fault Tolerance. ACM Computing Surveys, 40(2), article 6, 37 pages.

de Lemos, R. 2004. Analyzing Failure Behaviours in Component Interaction. Journal of Systems and Software, 71(1-2):97-115.

de los Angeles Martín, M. & Olsina, L. 2003. Towards an Ontology for Software Metrics and Indicators as the Foundation for a Cataloging Web System. Proceedings of the First Latin American Web Congress, 10-12 Nov, 2003. IEEE. Pages 103-113. DOI 10.1109/LAWEB.2003.1250288.

de Oliveira, K.M. & Zlot, F. & Rocha, A.R. & Travassos, G.H. & Galotta, C. & de Menezes, C.S. 2004.: Domain-Oriented Software Development Environment. Journal of Systems and Software, 72(2):145-161.

Decorte, S. & De Schreye, D. & Vandecasteele, H. 1999. Constraint-Based Termination Analysis of Logic Programs. ACM Transactions on Programming Languages and Systems, 21(6):1137-1195.

Deeprasertkul, P. & Bhattarakosol, P. & O'Brien, F. 2005. Automatic Detection and Correction of Programming Faults for Sofwtare Applications. Journal of Systems and Software, 78(2):101-110.

Delamaro, M.E. & Maldonado, J.C. & Mathur, A.P. 2001. Inference Mutation: An Approach to Integration Testing. IEEE Transactions on Software Engineering, 27(3):228-247.

Delgado, N. & Gates, A.Q. & Roach, S. 2004. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. IEEE Transactions on Software Engineering, 30(12):859-872.

DeLoach, S.A. & Hartrum, T.C. 2000. A Theory-Based Representation for Object-Oriented Domain Models. IEEE Transactions on Software Engineering, 26(6):500-517.

DeMillo, R.A. & Offutt, A.J. 1993. Experimental Results from an Automatic Test Case Generator. ACM Transactions on Software Engineering and Methodology, 2(2):109-127.

Demsky, B. & Rinard, M.C. 2006. Goal-Directed Reasoning for Specification-Based Data Structure Repair. IEEE Transactions on Software Engineering, 32(12):931-951.

Denaro, G. & Morasca, S. & Pezzè, M. 2002. Deriving Models of Software Fault-Proness. Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, 15-19 Jul, 2002. SESSION: Validation and Verification. ACM International Conference Proceeding Series, 27. ACM Press, New York, NY, USA. Pages 361-368. ISBN 1-58113-556-4.

Denning, P.J. 1976. Fault Tolerant Operating Systems. ACM Computing Surveys, 8(4):359-389.

Dershowitz, N. 2007. Software Horror Stories. [in Researcher Dershowitz's webpage]. The Tel Aviv University, Israel.
www.cs.tau.dc.il/~nachumd/horror.html
Retrieved: 24 Apr, 2007.

Desharnais, J. & Möller, B. & Tchier, F. 2000. Kleene Under a Demonic Star. The University of Augsburg, Germany, the Institute of Computer Science, Report 2000-3.

Dhurjati, D. & Kowshik, S. & Adve, V. & Lattner, C. 2005. Memory Safety Without Garbage Collection for Embedded Applications. ACM Transactions on Embedded Computer Systems, 4(1): 73-111.

Dijkstra, R.M. 2000. Computation Calculus Bridging a Formalization Gap. Science of Computer Programming, 37(1-3):3-36.

Dillon, L.K. & Stirewalt, R.E.K. 2003. Inference Graphs: A Computational Structure Supporting Generalization of Customizable and Correct Analysis Components. IEEE Transactions on Software Engineering, 29(2):133-150.

Donzelli, P. & Iazeolla, G. 2001. A Dynamic Simulator of Software Processors to test Process Assumptions. Journal of Systems and Software, 56(1): 81-90.

Doong, R.-K. & Frankl, P.G. 1994. The ASTOOT Approach to Testing Object-Oriented Programs. ACM Transactions on Software Engineering and Methodology, 3(2):101-130.

Dor, N. & Adams, S. & Das. M. & Yang, Z. 2004. Software Validation via Scalable Path-Sensitive Value Flow Analysis. ACM SIGSOFT Software Engineering Notes, 29(4):12-22.

Dorofeeva, R. & El-Fakih, K. & Maag, S. & Cavalli, A.R. & Yevtushenko, N. 2005. Experimental Evaluation of FSM-Based Testing Methods. Proceedings of the Third International Conference on Software Engineering and Formal Methods, 7-9 Sep, 2005. Pages 23-32. DOI 10.1109/SEFM.2005.17.

Dovier, A. & Piazza, C. & Pontelli, E. & Rossi, G. 2000. Sets and Constraint Logic Programming. ACM Transactions on Programming Languages and Systems, 22(5):861-931.

Doyle, J. 1979. A Truth Maintenance System. Massachusetts Institute of Technology, Artificial Intelligence Laboratory. Cambridge, MA, USA.

Duck. 2005. IMP Session Attached Delimiter Bug. Horde development mailing list.
http://lists.horde.org/archives/dev/Week-of-Mon-20050801/018349.html
Aug 6, 2005.
Retrieved: 30 Sep, 2008.

Dunham, C.B. 1986. Test for (IN)equality, Subtraction, Proof of Correctness. ACM SIGNUM Newsletter 21(3):27-30. ISSN 0163-5778.

Dunham, J.R. & Finelli, G.B. 1990. Real-Time Software Failure Characterization. Aerospace and Electronic Systems Magazine, IEEE, 5(11):38-49.

Dunlop, D.D. & Basili, V.R. 1982. A Comparative Analysis of Functional Correctness. ACM Computing Surveys 14(2):229-244.

Dupuy, A. & Leveson, N.G. 2000. An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software. IEEE Proceedings of the 19th Digital Aviation System Conference, Philadelphia, PA, USA, 7-13 Oct, 2000. ISBN 0-7803-6395-7.

Duran, J.W. & Ntafos, S.C. 1984. An Evaluation of Random Testing. IEEE Transactions on Software Engineering, 10(4): 438-444.

Eaddy, M. & Zimmermann, T. & Sherwood, K.D. & Garg, V. & Murphy, G.C. & Nagappan, N. & Aho, A.V. 2008. Do Crosscutting Concerns Cause Defects? IEEE Transactions on Software Engineering, 43(4): 497-515.

Easterbrook. S. & Callahan, J. 1998. Formal Methods for Verification and Validation of Partial Specifications: A Case Study. Journal of Systems and Software, 40(3):199-210.

Eckhardt, D.E. & Caglayan, A.K. & Knight, J.C. & Lee, L.D. & McAllister, D.F. & Vouk, A.M. & Kelly, J.P.J. 1991. An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability. IEEE Transactions on Software Engineering, 17(7): 692-702.

Egyed, A. 2003. A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Transactions on Software Engineering, 29(2):116-132.

Eick, S.G. & Graves, T.L. & Karr, A.F. & Marron, J.S. & Mockus, A. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. IEEE Transactions on Software Engineering, 27(1):1-12.

Eisenstadt, M. 1997. My Hairiest Bug War Stories. Communications of the ACM 40(4):30-37.

El Eman, K. & Wieczorek, I. 1998. The Repeatability of Code Defect Classifications. Proceedings on the 9th International Symposium of Software Reliability Engineering, 4-7 Nov, 1998. Pages 322-333. ISBN 0-8186-8991-9.

Elbaum, S. & Chin,H.N. & Dwyer, M.B. & Jorde, M. 2009. Carving and Replaying Differential Unit Test Cases from System Test Cases. IEEE Transactions on Software Engineering, 35(1): 29-45.

Elbaum, S. & Diep, M. 2005. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. IEEE Transactions on Software Engineering, 31(4):312-327.

(Mootaz) Elnozahy, E.N. & Alvisi, L. & Wang, Y.-M. & Johnson, D.B. 2002. A Survey of Rollback-Recovery Protocols in Message Passing Systems. ACM Computing Surveys, 34(3): 375-408 and 10 pages attached.

Endres, A. 1975. An Analysis of Errors and Their Causes in System Programs. ACM SIGPLAN Notices, 10(6):327-336.

Ernst, M.D. & Cockrell, J. & Griswold, W.G. & Notkin, D. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. IEEE Transactions on Software Engineering, 27(2):99-123.

Fagan, M.E. 1986. Advances in Software Inspection. IEEE Transactions on Software Engineering, SE-12(7):744-751.

Falk, H. 2004. Prolog to Formal Verification of Timed Systems: A Survey and Perspective. An Introduction to the Paper by Wang. Proceedings of the IEEE, 92(8): 1281-1282.

Farlex. 2009. The Free Dictionary by Farlex. 2009.
http://encyclopedia2.thefreedictionary.com/desk+checking
Retrieved: 11 Feb, 2009.

Fateman, R.J. 1990. Advances and Trends in the Design and Construction of Algebraic Manipulation Systems. Proceedings of the International Symposium on Symbolic and Algebraic Computation, Tokyo, Japan, 20-24 Aug, 1990. ACM Press, New York, NY, USA. Pages 60-67. ISBN 0-201-54892-5.

Feather, M.S. 1989. Constructing Specifications by Combining Parallel Elaborations. IEEE Transactions on Software Engineering, 15(2): 198-208.

Feather, M.S. 1998. Rapid Application of Ligheweight Formal Methods for Consistency Analyses. IEEE Transactions on Software Engineering, 24(11):949-959.

Feng, Q. & Lutz, R.R. 2005. Bi-Directional Safety Analysis of Product Lines. Journal of Systems and Software, 78(2):111-127.

Fenton, N.E. & Neil, M. 1999. A Critique of Software Defect Prediction Models. IEEE Transactions on Software Engineering, 25(5):675-689.

Fenton, N.E. & Ohlsson, N. 2000. Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Transactions on Software Engineering, 26(8):797-814.

Ferdinand, C. & Heckmann, R. & Wolff, H.-J. & Renz, C. & Gupta, M. & Parshin, O. 2006. Towards an Integration of Low-Level Timing Analysis and Model-Based Code Generation. Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, 15-19 Nov, 2006. Pages 220-226. DOI 10.1109/ISoLA.2006.31.

Fernandes, T. & Desharnais, J. 2007. Describing Data Flow Analysis Techniques with Kleene Algebra. Science of Computer Programming, 65:173-194.

Ferrari, M. & Fiorentini, C. & Fiorino, G. 2005. On the Complexity of the Disjunction Property in Intuitionistic and Modal Logics. ACM Transaction son Computational Logic, 6(3): 519-538.

Field, J. & Heering, J. & Dinesh, T.B. 1998. Equations as a Uniform Framework for Partial Evaluation and Abstract Intepretation. ACM Computing Surveys, 30(3es), article 2.

Fields, R. & Paternò, F. & Santoro, C. & Tahmassebi, S. 1999. Comparing Design Options for Allocating Communication Media in Cooperative Safety-Critical Contexts: A Method and a Case Study. ACM Transactions on Computer-Human Interaction, 6(4):370-398.

Filé, G. & Giacobazzi, R. & Ranzato, F. 1996. An Unifying View of Abstract Domain Design. ACM Computing Surveys 28(2):333-336.

Filliâtre, J.-C. 2007. Formal Proof of a Program: Find. Science of Computer Programming, 64(3):332-340.

Finkelstein, A. & Dowell, J. 1996. A Comedy of Errors: The London Ambulance Service Case Study. Proceedings of the 8th International Workshop of Software Specification and Design, 22-23, Mar, 1996. IEEE Computer Society, Washington DC, USA. From page 2. ISBN 0-8186-7361-3.

Fiore, M.P. 1995. Axiomatic Domain Theory. In: Fiore, M.P. & Jung, A. & Moggi, E. & O'Hearn, P. & Riecke, J. & Rosolini, G. & Stark, I. Domains and Denotational Semantics: History, Accomplishments, and Open Problems. The University of Birmingham. School of Computer Science. Technical report CSR-96-02.

Fitzpatrick, R. 2006. Computational Physics: an Introductory Course. Lecture Notes. The University of Texas, Austin. 29 Mar, 2006.
http://farside.ph.utexas.edu/teaching/329/lectures/node19.html
Retrieved: 30 Sep, 2008.

Flanagan, C. & Freund, S.N. 2004. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs (Summary). Proceedings of the 18th International Parallel and Distributed Processing Symposium, 26-30 Apr, 2004. Page 269. IEEE. DOI 10.1109/IPDPS.2004.1303345.

Flanagan, C. & Godefroid, P. 2005. Dynamic Partial-Order Reduction for Model Checking Software. ACM SIGPLAN Notices, 40(1):110-121. ACM Press, New York, NY, USA. ISSN 0362-1340.

Flanagan, C. & Qadeer, S. 2002. Predicate Abstraction for Software Verification. ACM SIGPLAN Notices, 37(1): 191-202.

Forgács I. 1994. Double Iterative Framework for Flow-Sensitive Interprocedural Data Flow Analysis. ACM Transactions on Software Engineering and Methodology, 3(1): 29-55.

Foster, K.A. 1980. Error Sensitive Test Cases Analysis. IEEE Transactions on Software Engineering, SE-6(3):258-264.

Francis, P. & Leon, D. & Minch, M. & Podgurski, A. 2004. Tree-based Methods for Classifying Software Failures. 2004. Proceedings on the 15th International Symposium on Software Reliability Engineering, 2-5 Nov, 2004. IEEE Computer Society, Washington DC, USA. Pages 451-462. ISBN ISSN 1071-9458, 0-7695-2215-7.

Frankl, P.G. & Hamlet, R.G. & Littlewood, B. & Strigini, L. 1998. Evaluating Testing Methods by Delivered Reliability. IEEE Transactions on Software Engineering, 24(8): 586-601.

Frankl, P.G. & Weyuker, E.J. 1988. An Applicable Family of Data Flow Testing Criteria. IEEE Transactions on Software Engineering, 14(10):1483-1498.

Frankl, P.G. & Weyuker, E.J. 1993a. An Analytical Comparison of the Fault-detecting Ability of Data Flow Testing Techniques. Proceedings of the 15th International Conference on Software Engineering, 17-21 May, 1993. IEEE. Pages 415-424. DOI 10.1109/ICSE.1993.346024.

Frankl, P.G. & Weyuker, E.J. 1993b. A Formal Analysis of the Fault-Detecting Ability of Testing Methods. IEEE Transactions on Software Engineering 19(3):202-213.

Frankl, P.G. & Weyuker, E.J. 1993c. Provable Improvements on Branch Testing. IEEE Transactions on Software Engineering, 19(10):962-975.

Frankl, P.G. & Weyuker, E.J. 2000. Testing Software to Detect and Reduce Risk. Journal of Systems and Software, 53(3):275-286.

Fraser, M.D. & Kumar, K. & Vaishnavi, V.K. 1991. Informal and Formal Requirements Specification Languages: Bridging the Gap. IEEE Transactions on Software Engineering, 17(5):454-466.

Fredericks, M. & Basili, V. 1998. Using Defect Tracking and Analysis to Improve Software Quality. A DACS State-of-the-Art Report, Contract Number SP0700-98-4000. 14 Nov, 1998. DACS (Data & Analysis Center for Software), Rome, NY, USA.

Freedman, R.S. 1991. Testability of Software Components. IEEE Transactions on Software Engineering, 17(6):553-564.

Frege, G. 1892. Über Sinn und Bedeutung. Zeitschrift für Philosophie und Philosophische Kritik, NF 100: 25-50.
http://www.gavagai.de/HHP31.htm
Retrieved: 24 Oct, 2008.
On Sense and Reference. Translated by Max Black. Extracted to Wikipedia.
http://en.wikisource.org/wiki/On_Sense_and_Reference
Retrieved: 24 Oct, 2008.

Fruth, L.S. & Purtilo, J.M. & White, E.L. 1996. A pattern-based object-linking mechanism for component-based software development environments. Journal of Systems and Software, 32(3):227-235.

Fränzle, M. 2004. Model-Checking Dense-Time Duration Calculus. Formal Aspects of Computing, 16(2):121-139.

FS Networks. 2005. Configuration Guide for Local Traffic Management. Version 9.0. MAN-0122-01.
http://www.scribd.com/doc/2630747/Configuration-Guide-for-Local-Traffic-Management-updated-7252005-LTM-config-guide
Retrieved: 30 Sep, 2008.

Fu, C. & Milanova, A. & Ryder, B.G. & Wonnacott, D.G. 2005. Robustness Testing of Java Server Applications. IEEE Transactions on Software Engineering, 31(4):292-311.

Fujiwara, S. & Bochmann, G.v. & Khendek, F. & Amalou, M. & Ghedamsi, A. 1991. Test Selection Based on Finite State Models. IEEE Transactions on Software Engineering, 17(6):591-603.

Gabow, H.N. & Maheshwari, S.N. & Osterweil, L.J. 1976. On Two Problems in the Generation of Program Test Paths. IEEE Transactions on Software Engineering, SE-2(3):227-231.

Gallian, J.A. 1996. Error Detection Methods. ACM Computing Survey, 28(3): 504-517.

Gannon, J.D. & Hamlet, R.G. & Mills, H.D. 1987. Theory of Modules. IEEE Transactions on Software Engineering, 13(7):820-829.

Ganssle, J.G. 1998.: Disaster! Break Points. [e-document]. Embedded.com. Official Site of Embedded Software Development Community.
http://embedded.com/98/9805br.htm
Retrieved: 23 Apr, 2007.

Garcia, A.F. & Rubira, C.M.F. & Romanovsky, A. & Xu, J. 2001. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. Journal of Systems and Software, 59(2):197-222.

García-Muñoz, L.H. & Armendáriz-Íñigo, J.E. & Decker, H. & Muñoz-Escoí, F.D. 2007. Recovery Protocols for Replicated Databases – a Survey. 21st International Conference on Advanced Information Networking and Application Workshops, 21-23 May, 2007. IEEE. Pages 220-227. DOI 10.1109/AINAW.2007.306.

Garg, P. 1994. Investigating Coverage-Reliability Relationship and Sensitivity of Reliability to Errors in the Operational Profile. Proceedings of the First International Conference on Software Testing, Reliability, and Quality Assurance, 21-22 Dec, 1994. IEEE. Pages 21-35. DOI 10.1109/STRQA.1994.562380.

Gargantini, A. & Morzenti, A. 2001. Automated Deductive Requirements Analysis of Critical Systems. ACM Transactions on Software Engineering and Methodology, 10(3): 255-307.

Gates, A.Q. & Mondragon, O. 2002. FasTLInC: A Constraint-Based Tracing Approach. Journal of Systems and Software, 63(3):241-258.

Gay, D. & Aiken, X. 1998. Memory Management with Explicit Regions. ACM SIGPLAN Notices, 33(5): 313-323.

Gencel, C. & Dermiros, O. 2008. Functional Size Measurement Revisited. ACM Transactions on Software Engineering and Methodology, 17(3), article 15, 36 pages.

Georget, Y. & Codognet, P. 1998. Encoding Global Constraints in Semiring-Based Constraint Solving. 10th International IEEE Conference on Tools with Artificial Intelligence, 10-12 Nov, 1998. Pages 400-407. DOI 10.1109/TAI.1998.744878.

Gerhart, S.L. 1984. Application of Axiomatic Methods to a Specification Analyzer. Proceedings of the 7th International Conference on Software Engineering, Ontario, FL, USA, 26-29 Mar, 1984. IEEE. Pages 441-451.

Gerhart, S.L. & Yelowitz, L. 1976. Observations of Fallibility in Applications of Modern Programming Methodologies. IEEE Transactions on Software Engineering, SE-2(3):195-207.

Gericota, M.G. & Lemos, L.F. & Alves, G.R. & Barbosa, M.M. & Ferreira, J.M. 2006. A Framework for Fault Tolerant Real Time Systems Based on Reconfigurable FPGAs. IEEE Conference on Emerging Technologies and Factory Automation, 20-22 Sep, 2006. Pages 131-138. DOI 10.1109/EFTA.2006.355409.

Germain, E. & Robillard, P.N. 2005. Engineering-Based Processes and Agile Methodologies for Software Development: a Comparative Case Study. Journal of Systems and Software, 75(1-2):17-27.

German, R. & Logothetis, D. & Trivedi, K.S. 1995. Transient Analysis of Markov Regenerative Stochastic Petri Nets: A Comparison of Approaches. Proceedings on the sixth International Workshop on Petri Nets and Performance Models, 3-6 Oct, 1995. Pages 103-112. DOI 10.1109/PNPM.1995.524320.

Gerogiannis, V.C. & Kameas, A.D. & Pintelas, P.E. 1998. Comparative Study and Categorization of High-Level PETRI Nets. Journal of Systems and Software, 43(2):133-160.

Gerrard, C.P. & Coleman, D. & Gallimore, R.M. 1990. Formal Specification and Design Time Testing. IEEE Transactions on Software Engineering, 16(1): 1-12.

Giguette, R. & Hassell, J. 1999. Toward a Resourceful Method on Software Fault Tolerance. Proceedings of the 37 annual South-East Regional Conference (CD-ROM, article 1) ACM-SE 37. ACM Press, New York, NY, USA. ISBN 1-58113-128-3

Giguette, R. & Hassell, J. 2000. A Relational Database Model of Program Execution and Software Components. Proceedings of the 38 annual South-East Regional Conference, Clemson, SC, USA, Apr 7-8, 2000. SESSION: Software Testing and Fault Tolerance. ACM Press, New York, NY, USA. Pages 146-155. ISBN 1-58113-250-6.

Giguette, R. & Hassell, J. 2002. Designing a Resourceful Fault-Tolerance System. Journal of systems and software, 62(1):47-57.

Glass, R.L. 1981. Persistent Software Errors. IEEE Transactions on Software Errors, 7(2): 162-168.

Glinz, M. 2000. Improving the Quality of Requirements with Scenarios. Proceedings of the Second World Congress for Software Quality, Yokohama, Japan, Sep, 2000. Pages 55-60.

Go, K. & Shiratori, N. 1999. A Decomposition of a Formal Specification: An Improved Constraint-Oriented Method. IEEE Transactions on Software Engineering, 25(2): 258-273.

Godfrey, M.W. & Zou, L.: 2005. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. IEEE Transactions on Software Engineering, 31(2):166-181.

Goel, A.L. & Okumoto, K. 1979. A Time-Dependence Error Detection Rate Model for Software Reliability and Other Performance Measures. IEEE Transactions on Reliability, 28(3):206-211.

Gokhale, S.S. & Marinos, P.N. & Lyu, M.R. & Trivedi, K.S. 1997. Effect of Repair Policies on Software Reliability. Proceedings of the 12th Annual Conference on Computer Assurance, Gatheisburg, MD, USA, 16-19 Jun, 1997. IEEE. Pages 105-116. DOI 10.1109/CMPASS.1997.613262.

Gold, N.E. & Harman, M. & Binkley, D & Hierons, R.M. 2005. Unifying Program Slicing and Concept Assignment for High-Level Executable Source Code Extraction. Software Practice & Experience, 35(10):977-1006.

Goldberg, D. 1991. What Every Computer Scientist Should Know about Floating-Point Arithmetic. ACM Computing Surveys, 23(1):5-48.

Goodenough, J.B. & Gerhart, S.L. 1975. Towards a Theory of Test Data Selection. IEEE Transaction on Software Engineering, SE-1(2):156-173.

Gorla, N. & Pu, H.-C. & Rom, W.O. 1995. Evaluation of Process Tools in Systems Analysis. Information and Software Technology, 37(2):119-126.

Goseva-Popstojanova, K. & Hassan, A. & Guedem, A. & Abdelmoez, W. & Nassar, D.E.M. & Ammar, H. & Mili, A. 2003. Architectural-Level Risk Analysis Using UML. IEEE Transactions on Software Engineering, 29(10):946-960.

Goto, Y. & Cheng, J. 2006. A Quantitative Analysis of Implicational Paradoxes in Classical Mathematical Logic. Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, 23-27 Apr, 2006. Pages 42-43.

Gougen, J.A. & Burstall, R.M. 1992. Institutions: Abstract Model Theory for Specifications and Programming. Journal of the ACM 39(1):95-146.

Graves, T.L. & Karr, A.F. & Marron, J.S. & Siy, H. 2000. Predicting Fault Incidence Using Software Change History. IEEE Transactions on Software Engineering, 26(7): 653-661.

Gray, A.R. & MacDonell, S.G. 1997. A Comparison of Techniques for Developing Predictive Models of Software Metrics. Information and Software Technology 39(6):425-437.

Gray, J. 1985. Why do Computers Stop and What Can be Done about It? Tandem Computers. Cupertino, Canada. Technical Report 85.7.

Green, G.C. & Hevner, A.R. & Collins, R.W. 2005. The Impacts of Quality and Productivity Perceptions on the Use of Software Process Improvement Innovations. Information and Software Technology 47(8):543-553.

Gregoriades, A. & Sutcliffe, A. 2005. Scenario-Based Assessment of Nonfunctional Requirements. IEEE Transactions on Software Engineering, 31(5):392-409.

Gries, D. 1981. The Science of Programming. Springer. New York. 366 pages. ISBN 0387964800.

Grindal, M. & Offutt, J. & Andler, S.F. 2004. Combination Testing Strategies: A Survey. George Manson University, Fairfax, FL, Department of Information and Software Engineering. GMU Technical Report, ISE-TR-04-05. Jul, 2004.

Große-Rhode, M. 2002. Compositional Comparison of Formal Software Specifications using Transformation Systems. Formal Aspects or Computing, 13(2):161-186.

Grottke, M. & Trivedi, K.S. 2007. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. IEEE Computer, 40(2):107-109.

Grundy, J. & Hosking, J. & Mugridge, W.B. 1998. Inconsistency Management for Multiple-View Software Development Environments. IEEE Transactions on Software Engineering, 24(11):960-981.

Gu, W. & Kalbarczyk, Z. & Iyer, R.K. & Yang, Z. 2003. Characterization of Linux Kernel Behavior under Errors. Proceedings of the International Conference on Dependable Systems and Networks, 22-25 Jun, 2003. Page 459-.

Guerrini, S. & Martini, S. & Masini, A. 1997. An Analysis of (Linear) Exponentials Based on Extended Sequents. University of Pennsylvania, Philadelphia, PA, USA. NSF Science and Technology Center for Research in Cognitive Science. Computer Science. Technical Reports. IRCS-97-13. 24 Jul, 1997.

Gunter, E. & Peled, D. 2005. Model Checking, Testing and Verification Working Together. Formal Aspects in Computing, 17(2):201-221.

Guo, J. 2003. Software Reuse through Re-Engineering the Legacy Systems. Information and Software Technology, 45(9):597-609.

Guo, L. & Ma, Y. & Cukic, B. & Singh, H. 2004. Robust Prediction of Fault-Proness by Random Forests. 15th International Symposium on Software Reliability Engineering, 2-5 Nov 2-5, 2004. IEEE Computer Society, Washington DC, USA. Pages 417-428. ISBN ISSN 1071-9458, 0-7695-2215-7.

Guo, L. & Mukhopadhyay, S & Cukic, B. 2004. Does Your Result Checker Really Check? Proceedings of the International Conference on Dependable Systems and Networks, 28 Jun – 1 Jul, 2004. IEEE Computer Society, Washington DC, USA. Pages 399-404. ISBN 0-7695-2052-9.

Gupta, R. & Soffa, M.L. & Howard, J. 1997. Hybrid Slicing: Integrating Dynamic Information with Static Slicing. ACM Transactions on Software Engineering and Methodology, 6(4): 370-397.

Gupta, V. & Jagadeesan, R. Saraswat, V.A. 1998. Computing with Continuous Change. Science of Computer Programming, 30(1-2): 3-49.

Güne§ Koru, A. & Tian, J. 2003. An Empirical Comparison and Characterization of High Defect and High Complexity Modules. Journal of Systems and Software 67(3):153-163.

Hac, A. & Chu, X. 1998. A New Cell Loss Recovery Method Using Forward Error Correction in ATM Networks. International Journal of Network Management, 8(2):87-103.

Hakuta, M. & Ohminami, M. 1997. A Study of Software Portability Evaluation. Journal of Systems and Software, 38(2):145-154.

Halder, A.K. 1982. Karnaugh Map Extended to Six or More Variables. IEEE. Electronic Letters 18(20): 868-870.

Hamlet, D. 1994. Foundations of Software Testing. ACM SIGSOFT Software Engineering Notes, 19(5): 128-139.

Hamlet, D. & Taylor, R. 1990. Partition Testing Does Not Inspire Confidence (Program Testing). IEEE Transactions on Software Engineering, 16(12):1402-1411.

Hamlet, R. 1989. Theoretical Comparison of Testing Methods. ACM SIGSOFT Software Engineering Notes, 14(8):28-37. ACM Press, New York, NY, USA. ISSN 0163-5948.

Hangal, S. & Lam, M.S. 2002. Tracking Down Software Bugs Using Automatic Anomaly Detection. Proceedings of the 24th International Conference on Software Engineering, Orlando, FL, USA, 19-25 May, 2002. Pages 291-301. ISBN 1-58113-472-X.

Hansen, K.M. & Ravn, A.P. & Stavridou, V. 1998. From Safety Analysis to Software Requirements. IEEE Transactions on Software Engineering, 24(7):573-584.

Hansen, M.D. 1989. Survey of Available Software-Safety Analysis Techniques. Proceedings of Annual Reliability and Maintainability Symposium. Atlanta, GA, USA, 24-26 Jan, 1989.

Haraszti, Z. & Townsend, J.K. 1999. The Theory of Direct Probability Redistribution and its Application to Rare Event Simulation. ACM Transactions on Modeling and Computer Simulation, 9(2):105-140.

Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8: 231-274.

Harman, M. & Binkley, D. & Danicic, S. 2003. Amorphous Program Slicing. Journal of Systems and Software, 68(1):45-64.

Harrold, M.J. & Offutt, A.J. & Tewary, K. 1997. An Approach to Fault Modeling and Fault Seeding Using the Program Dependence Graph. Journal of Systems and Software, 36(3):273-295.

Hartel, P.H. & Moreau, L. 2001. Formalizing the Safety of Java, the Java Virtual Machine, and Java Card. ACM Computing Surveys, 33(4):517-558.

Hatton, L. 1996. Is Modularization Always a Good Idea? Information and Software Technology 38(11):719-721.

Hatton, L. 1999. Repetitive Failure, Feedback and the Lost Art of Diagnosis. Journal of Systems and Software, 47(2-3):183-188.

Hatton, L. 2004. Safer Language Subsets: An Overview and a Case History, MISRA C. Information and Software Technology, 46(7):465-472.

Hatton, L. 2005. EC - A Measurement-Based Safer Subset of ISO C Suitable for Embedded System Development. Information and Software Technology 47(3):181-187.

Hatton, L. & Roberts, A. 1994. How Accurate is Scientific Software? IEEE Transactions on Software Engineering, 20(10):785-797.

Haugen, ø. 2005. Comparing UML 2.0 Interactions and MSC-2000. Lecture Notes in Computer Science, 3319/2005: 65-79. ISBN 978-3-540-24561-2.

Havelund, K. & Lowry, M. & Penix, J. 2001. Formal Analysis of a Space-Craft Controller Using SPIN. IEEE Transactions on Software Engineering, 27(8):749-765.

Hayes, I. 2002. Reasoning about Real-Time Repetitions: Terminating and Non-Terminating. Science of Computer Programming 43(2-3):161-192.

Hayes, J.H. & Dekhtyar, A. & Sundaram, S.K. 2006. IEEE Transactions on Software Engineering, 32(1): 4-19.

He, J.-Z. & Zhou, Z.-H. & Yin, X.-R. & Chen, S.-F. 2000. Using Neural Networks for Fault Diagnosis. Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, 24-27 Jul, 2000. Vol 5. Pages 217-220. DOI 10.1109/IJCNN.2000.861460.

He Jifeng & Liu Zhiming & Li Xiaoshan. 2002. Towards a Refinement Calculus for Object Systems. Proceedings of the first International Conference on Cognitive Automatics, 19-20 Aug, 2002. Pages 69-76. DOI 10.1109/COGINF.2002.1039284.

He, X. & Yu, H. & Shi, T. & Ding, J. & Deng, Y. 2004. Formally Analyzing Software Architectural Specifications Using SAM. Journal of Systems and Software. 71(1-2): 11-29.

Hecht, M.S. & Ullman, J.D. 1973. Analysis of a Simple Algorithm for Global Data Flow Problems. Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. P. 207-217.

Heimdahl, M.P.E. & Czerny, B.J. 2000. On the Analysis Needs When Verifying State-Based Software Requirements: an Experience Report. Science of Computer Programming 36(1):65-96.

Heintze, N. & Jaffar, J. 1990. A Decision Procedure for a Class of Set Constraints. Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science, Philadelphia, PA, USA, 4-7 Jun, 1990. ISBN 0-8186-2073-0.

Heintze, N. & McAllester, D. 1997. On the Complexity of Set-Based Analysis. ACM SIGPLAN Notices, 32(8):150-163. ACM Press, New York, NY, USA. ISBN 0-89791-918-1.

Hentenryck, P.V. & Michel, L. & Benhamou, F. 1998. Constraint Programming over Nonlinear Constraints. Science of Computer Programming, 30(1-2): 83-118.

Hepner, M. & Gamble, R. & Kelkar, R. & Davis, L. & Flagg, D. 2006. Patterns of Conflict among Software Components. Journal of Systems and Software, 79(4): 537-551.

Herrmann, D.S. & Peercy, D.E. 1999. Software Reliability Cases. A Bridge Between Hardware, Software and System Safety and Reliability. Reliability and Maintainability Symposium, 1999. Proceedings. Annual. Washington DC, USA, 18-21 Jan, 1999. Pages 396-402. ISBN 0-7803-5143-6.

Hierons, R.M. 2002. Comparing Test Sets and Criteria in the Presence of Test Hypotheses and Fault Domains. ACM Transactions on Software Engineering and Methodology, 11(4):427-448.

Hierons, R.M. 2006. Avoiding Coincidental Correctness in Boundary Value Analysis. ACM Transactions on Software Engineering and Methodology, 15(3):227-241.

Hierons, R. & Harman, M. & Fox, C. & Ouarbya, L. & Daoudi, M. 2003. Conditioned Slicing Supports Partition Testing. Software Testing, Verification, and Reliability 2002, 12(1): 23-28.

Hiller, M. & Jhumka, A. & Suri, N. 2002. PROPANE: An Environment for Examining the Propagation of Errors in Software. Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, 27(4): 81-85.

Hocenski, Z. & Martinovic, G. 1999. Influence of Software on Fault-Tolerant Microprocessor Control System Dependability. Proceedings of the IEEE International Symposium on Industrial Electronics, volume 3. 12-16 Jul, 1999. Pages 1193-1197. DOI 10.1109/ISIE.1999.796866.

Hochstein, L. & Lindvall, M. 2005. Combating Architectural Degeneration: a Survey. Information and Software Technology 47(10):643-656.

Hoffman, D. & Snodgrass, R. 1988. Trace Specifications: Methodology and Models. IEEE Transactions on Software Engineering, 14(9):1243-1252.

Holmberg, A. & Eriksson, P.-E.. 2006. Decision Support System for Fault Isolation of JAS 39 Gripen. Development and Implementation. Master Thesis. Department of Electrical Engineering. The University of Linköping. LiTH-ISY-EX--06/3839--SE.

Holzmann, G.J. 1997. The Model Checker SPIN. IEEE Transactions on Software Engineering, 23(5): 279-295.

Hong, H.S. & Cha, S.D. & Lee, I. & Sokolsky, O & Ural, H. 2003. Data Flow Testing as Model Checking. Proceedings of the 25th International Conference on Software Engineering, Portland, OR, USA, 3-10 May, 2003. SESSION: Technical Papers: Testing II: IEEE Computer Society, Washington DC, USA. Pages 232-242. ISBN ISSN 0270-5257, 0-7695-1877-X.

Horwath, T, & Green , J. Lawler, T. 2000. SilkTest and WinRunner Feature Descriptions.
www.infotest.by/documents/SilkWrFeatures.pdf
Retrieved: 20 Oct, 2008.

Hou, R.-H. & Kuo, S.-Y. & Chang, Y.-P. 1994. Applying Various Learning Curves to Hyper-Geometric Distribution Software Reliability Growth Model. Proceedings of the 5th International Symposium on Software Reliability Engineering, 6-9 Nov, 1994. IEEE. Pages 8-17. DOI 10.1109/ISSRE.1994.341342

Hovemeyer, D. & Pugh, W. 2004. Onward!: Finding Bugs is Easy. Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Vancouver, BC, Canada, 24-28 Oct, 2004. ACM Press, New York, NY, USA. Pages 132-136. ISBN 1-58113-833-4.

Howden, W.E. 1976. Reliability of the Path Analysis Testing Strategy. IEEE Transactions on Software Engineering, SE-2(3):208-215.

Howden, W.E. 1980. Functional Program Testing. IEEE Transactions on Software Engineering SE-6(2):162-169.

Howden, W.E. 1982. Weak Mutation Testing and Completeness of Test Sets. IEEE Transactions on Software Engineering, SE-8(4):371-379.

Howden, W.E. 1986. A Functional Approach to Program Testing and Analysis. IEEE Transactions on Software Engineering, SE-12(10):997-1005.

Howden, W.E. 1990. Comment Analysis and Programming Errors. IEEE Transactions on Software Engineering, 16(1):72-81.

Huang, J.C. 1980. A New Verification Rule and Its Applications. IEEE Transactions on Software Engineering, SE-6(5):480-484.

Huang, J.C. 1990. State Constraints and Pathwise Decomposition of Programs. IEEE Transactions on Software Engineering, 16(8):880-896.

Huckle, T. 2005. Collection of Software Bugs. [Professor Huckle's webpage]. The University of Munich, Germany.
http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html
Retrieved: 28 Apr, 2007.

Huebscher, M.C. & McCann, J.A. 2008. A Survey of Autonomic Computing – Degrees, Models, and Applications. ACM Computing Surveys, 40(3), article 7, 28 pages.

Hulgaard, H. & Burns, S.M. & Amon, T. & Borriello, G. 1995. An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems. IEEE Transactions on Computers, 44(11): 1306-1317.

Hungerford, B.C. & Hevner, A.R. & Collins, R.W. 2004. Reviewing Software Diagrams: A Cognitive Study. IEEE Transactions on Software Engineering, 30(2):82-96.

IEEE: 1990. IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology. IEEE (The Institute of Electrical and Electronics Engineers) Standards Association, 1990.

IEEE1540: 2001. IEEE Std 1540-2001: IEEE Standard for Software Life Cycle Processes - Risk Management. IEEE Software Engineering Standards Committee, 2001.

Ip, C.N. & Dill, D.L. 1996. State Reduction Using Reversible Rules. Proceedings of the 33rd Annual Conference on Design Automation, Las Vegas, NV, USA, 3-7 Jun, 1996. ACM Press, New York, NY, USA. Pages 564-567. ISBN 0-89791-779-0.

IPL Information Processing Ltd. 1999. Advanced Coverage Metrics for Object-Oriented Software. Last Update 28 Oct, 1999. Bath, UK.
http://www.ipl.com/pdf/p0833.pdf
Retrieved: 14 Mar, 2007.

Isoda, S. 1998. A Critisism on the Capture-and-Recapture Method for Software Reliability Assurance. Journal of Systems and Software 43(1):3-10.

Israel, S. & Morris, D.1989. A Non-Intrusive Checkpointing Strategy. The Eight Annual International Phoenix Conference on Computers and Communications, 22-24 Mar, 1989. IEEE. Pages 413-421. DOI 10.1109/PCCC.1989.37424.

Itzfeldt, W.D. 1990. Quality Metrics for Software Management and Engineering. Proceedings on the Conference on Managing Complexity in Software Engineering, 13 Apr, 1990. IET Conference Proceedings. IEE Computing Series 17. Series editors: Carré, B. & Jacobs, D.A.H. & Sommerville, I. Pages 127-152.

Iwata, N. & Hanazawa, M. 1993. Porting Classes and Coding Rules for C Programs on CTRON. The 10th TRON Project International Symposium. IEEE. 1-2 Dec, 1993. Pages 144-155. DOI 10.1109/TRON.1993.589181.

Iyer, R.K. & Hsueh, M.-C. & Lee, I. 1996. Fault/Failure Analysis of the Tandem NonStop-UX Operating System. 15th AIAA/IEEE Digital Avionics Systems Conference, Atlanta, GA, USA, 27-31 Oct, 1996. Pages 491-497. ISBN 0-7803-3385-3.

Jackson, D. 1995. Detecting Bugs with Abstract Dependencies. ACM Transactions on Software Engineering and Methodology, 4(2): 109-145.

Jackson, D. 2002. Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology, 11(2): 256-290.

Jacky, J. 1987. Mariner I and Computer Folklore. In: P.G. Neumann (Moderator). Risk Digest. Forum on Rules to Public in Computers and Related Systems. ACM Committee on Computers and Public Policy. 5(65).
http://catless.ncl.ac.uk/Risks
Retrieved: 24 Apr, 2007.

Jacky, J. 1995. Specifying a Safety-Critical Control System in Z. IEEE Transactions on Software Engineering, 21(2): 99-106.

Jacobs, J. & Moll, J.v. & Krause, P. & Kusters, R. & Trienekens, J. & Brombacher, A. 2005. Exploring Defect Causes in Products Developed by Virtual Teams. Information and Software Technology 47(6):399-410.

Jacobs, J. & Moll, J.v. & Kusters, R. & Trienekens, J. & Brombacher, A. 2007. Identification of Factors that Influence Defect Injection and Detection in Development of Software Intensive Products. Information and Software Technology 49(7):774-789.

Jahanian, F. & Mok, A K. L. 1986. Safety Analysis of Timing in Real-Time Systems. IEEE Transactions on Software Engineering, 12(9): 890-904.

Jalote, P. 1989. Testing the Completeness of Specifications. IEEE Transactions on Software Engineering, 15(5):526-531.

Jalote, P. & Munshi, R. & Probsting, T. 2007. The When – Who – How Analysis of Defects for Improving the Quality of Control Process. Journal of Systems and Software, 80(4): 584-589.

Janicki, R. & Khedri, R. 2001. On Formal Semantics of Tabular Expressions. Science of Computer Programming 39(2-3):189-213.

Jansen, D.N. & Hermanns, H. 2005. QoS Modelling and Analysis with UML Statecharts: The StoCharts Approach. ACM SIGMETRICS Performance Evaluation Review, 32(4): 28-33

Jeffrey, D. & Gupta, N. 2007. Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. IEEE Transactions on Software Engineering, 33(2): 108-123.

Jeng, B. 1999. Toward an Integration of Data Flow and Domain Testing. Journal of Systems and Software, 45(1):19-30.

Jeng, B. & Forgács, I. 1999. An Automatic Approach of Domain Test Data Generation. Journal of Systems and Software, 49(1):97-112.

Jeng, B. & Weyuker, E.J. 1994. A Simplified Domain-Testing Strategy. ACM Transactions on Software Engineering and Methodology, 3(3):254-270.

Jeon, G. & Cho, Y. 2002. A Partitioning Method for Efficient System-Level Diagnosis. Journal of Systems and Software, 63(1):1-16.

Jiang, J.J. & Klein, G. & Balloun, J.L. & Crampton, S.M. 1999. System Analysts' Orientations and Perceptions of System Failure. Information and Software Technology 41(2):101-106.

Hongxia Jin & Sullivan, G.F. & Masson, G.M. 1999. Approximate Correctness-Checking of Computational Results. IEEE Transactions on Reliability, 48(4):338-350.

Jo, J.-W. & Chang, B.-M. & Yi, K. & Choe, K.-M. 2003. An Uncaught Exception Analysis for Java. Journal of Systems and Software 72(1):59-69.

Johnson, A.M. & Malek, M. 1988. Survey of Software Tools for Evaluating Reliability, Availability, and Serviceability. ACM Computing Surveys, 20(4): 227-269.

Johnson, C.W. 2003. Failure in Safety-Critical Systems: A Handbook of Accident and Incident Reporting. University of Glasgow Press, Glasgow, Scotland, Oct 2003. ISBN 0-85261-784-4.
Available in electronic form: http://www.dcs.gla.ac.uk/~johnson/book

Johnson, D.W. 2007. Software Quality Tips. Software Testing. How to Evaluate Testing Software and Tools. 13 Mar, 2007. Searchsoftwarequality. Techtarget.
http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1247269,00.html
Retrieved: 21 Oct, 2008.

Jones, R.B. 1996. Formal Specification Languages. Essays in Sceptical Philosophy. Factasia. [Jones' web pages]. 9. Sep, 1996.
http://www.rbjones.com/rbjpub/cs/csfm02.htm
Retrieved: 28 Apr, 2007.

Jones, R.B. 2007. Judgement Forms. Essays in Sceptical Philosophy. Factasia. [Jones' web pages].
http://www.rbjones.com/rbjpub/logic/log004.htm
Retrieved: 28 Apr, 2007.

Jones, S. & Till, D. & Wrightson, A.M. 1998. Formal Methods and Requirements Engineering: Challenges and Synergies. Journal of Systems and Software, 40(3):263-273.

JPL. 2000. Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions. JPL Special Review Board Report JPL-D-18709. NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA. E.g. pages 13 and 111-123.

Jorgensen, P.C. & Erickson, C. 1994. Object-Oriented Integration Testing. Communications of the ACM, 37(9): 30-38.

Kandara, O. 2003. Level of Essentialness of a Node in Flowchart and its Application to Program Testing. Dissertation. Dec 2003. Louisiana State University, Agricultural and Mechanical College.

Kaner, C. 2004. Teaching Domain Testing: A Status Report. Proceedings of the 17th Conference on Software Engineering Education and Training. IEEE, 1-3 Mar, 2004.

Kansomkeat, S. & Rivepiboon, W. 2003. Automated-Generating Test Case Using UML Statechart Diagrams. Proceedings of the 2003 Annual Research Conference on the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology, 17-19 Sep, 2003. ACM International Conference Proceeding Series, 47: 296-300.

Kantz, H. & Koza, C. 1995. The ELECTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity. 25th International Symposium on Fault-Tolerant Computing, Pasadena, CA, USA, 27-30 Jun, 1995. FTCS-25. Pages 453-458. ISBN 0-8186-7079-7.

Katara, M. & Katz, S. 2003. Architectural Views of Aspects. Proceedings of the 2nd International Conference on Aspect Oriented Software Development, Boston, MA, USA, 17-21 Mar, 2003. ISBN 1-58113-660-9.

Katz, S. 1993. A Superimposition Control Construct for Distributed Systems. ACM Transactions on Programming Languages and Systems, 15(2):337-356.

Keck, D.O. & Kuehn, P.J. 1998. The Feature and Service Interaction Problem in Telecommunications Systems: A Survey. IEEE Transactions on Software Engineering, 24(10):779-796.

Kelly, D. & Shepard, T. 2001. A case study in the use of defect classification in inspections. Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative Research, Toronto, ON, CA, 5-7 Nov, 2001. IBM Press. Page 7.

Kelly, D. & Shepard, T. 2004. Task-Related Software Inspection. Journal of Systems and Software, 73(2):361-368.

Khajenoori, S. & Prem, L. & Stevens, K. & Keng, B.S. & Kameli, N. 2004. Knowledge Centered Assessment Pattern: An Effective Tool for Assessing Safety Concerns in Software Architecture. Journal of Systems and Software, 73(2):313-322.

Khoshgoftaar, T.M. & Allen, E.B. & Kalaichelvan, K.S. & Goel, N. 1996. Early Quality Prediction: A Case Study in Telecommunications. IEEE Software, 13(1):65-71.

Khoshgoftaar, T.M. & Ganesan, K. & Allen, E.B. & Ross, F.D. & Munikoti, R. & Goel, N. & Nadhi, A. 1997. Predicting Fault-Prone Modules with Case-Based Reasoning. Proceedings of the Eight International Symposium on Software Reliability Engineering, 2-5 Nov, 1997. IEEE. From page 27. ISBN 0-8186-8120-9.

Khoshgoftaar, T.M. & Seliya, N. & Herzberg, A. 2005. Resource-Oriented Software Quality Classification Models. Journal of Systems and Software, 76(2):111-126.

Kifer, M. & Lozinskii, E.L. 1989. RI: A Logic for Reasoning with Inconsistency. Proceedings of the Fourth Annual Symposium on Logic in Computer Science, 5-8 Jun, 1989. Pages 253-262. DOI 10.1109/LICS.1989.39180.

Sun Kim & Bastani, F.-B. & I-Ling Yen & Chen, I.-R. 2004. Systematic Reliability Analysis of a Class of Application-Specific Embedded Software Frameworks. IEEE Transactions on Software Engineering, 30(4):218-230.

Kim, S. & Pan, K. & Whitehead, Jr, E.E.J. 2006. Memories of Bug Fixes. Proceedings of the 14th ACM SIGSOFT International Symposium of Foundations of Foundations of Software Engineering, Portland, OR, USA, 5-11 Nov, 2006. SESSION: Mining Failures and Bugs. ACM Press, New York, NY, USA. Pages 35-45. ISBN 1-59593-468-5.

Kiper, J.D. 1992. Structural Testing of Rule-Based Expert Systems. ACM Transactions on Software Engineering and Methodology, 1(2): 168-187.

Kiran, N.R. & Ravi, V. 2008. Software Reliability Prediction by Soft Computing Techniques. Journal of Systems and Software, 81(4): 576-583.

Kirk, S.R. & Jenkins, S. 2004. Information Theory-Based Software Metrics and Obfuscation. Journal of Systems and Software, 72(2):179-186.

Kistler J.J. & Satyanarayanan, M. 1992. Disconnected Operation in the CODA File System. ACM Transactions on Computer Systems 10(1):3-25.

Kitchenham, B.A. & Linkman, S.G. & Law, D.T. 1994. Critical Review of Quantitative Assessment. IEEE Software Engineering Journal 9(2):43-53.

Klop, J.W. .1992. Term Rewriting Systems. In: Abramsky, S. & Gabbay, D.M. & Maibaum, T.S.E. (eds.): Handbook of logic in computer science. Vol. 2: Background : Computational Structures. Oxford: Clarendon Press. 582 pages. Pages 1-116. ISBN 0198537611.

Knight, K. 1989. Unification: A Multidisciplinary Survey. ACM Computing Surveys 21(1):93-124.

Kohlas, J. & Stärk, R.F. 2007. Information Algebras and Consequence Operators. Logica Universalis, 1(1):139-165.

Koono, Z. & Soga, M. 1990. Structural Way of Thinking as Applied to Quality Assurance Management. IEEE Journal on Selected Areas in Communications, 8(2): 291-300.

Kopetz, H. 1975. On the Connections Between Range of Variables and Control Structure Testing. ACM SIGPLAN Notices, 10(6):511-517. ACM Press, New York, NY, USA.

Kopetz, H. 2000. Software Engineering for Real Time: A Roadmap. Proceedings of the Conference on the Future Software Engineering. Limerick, Ireland, 4-11 Jun, 2000. ACM Press, New York, NY, USA. Pages 201-211. ISBN 1-58113-253-0.

Korel, L. 1988. PELAS - Program Error-Locating Assistance System. IEEE Trans. on Software Engineering, 14(9):1253-1260.

Kramer, J. & Cunningham, R.J. 1979. Invariants for Specifications. Proceedings of fourth International Conference on Software Engineering. Munich, Germany, 17-19 Sep, 1979. IEEE Press, Piscataway, NJ, USA. Pages 183-193.

Krishnamurthy, D. & Rolia, J.A. & Majumdar, S. 2006. A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems IEEE Transactions on Software Engineering 32(11):868-882.

Krishnan, M.S. & Kellner, M.I. 1999. Measuring Process Consistency: Implications for Reducing Software Defects. IEEE Transactions on Software Engineering, 25(6):800-815.

K.S. How Tai Wah. 2003. An Analysis of Coupling Effect: I: Single Test Data. Science of Computer Programming 48(2-3):119-161.

Kuhn, D.R. 1999. Fault Classes and Error Detecting Capacity of Specification-Based Testing. ATM Transactions on Software Engineering and Methodology 8(4):411-424.

Kuhn, D.R. & Reilly, M.J. 2002. An Investigation of the Applicability of Design of Experiments to Software Testing. Proceedings of 27th Annual NASA Gobbard Software Engineering Workshop. 5-6 Dec, 2002. IEEE Computer Society, Washington DC, USA. From page 91. ISBN 0-7695-1855-9.

Kuhn, D.R. & Wallace, D.R. & Gallo, A.M., Jr. 2004. Software Fault Interactions and Implications for Software Testing. IEEE Transactions on Software Engineering, 30(6):418-421.

Kulisch, U.W. & Miranker, W.L. 1981. Computer Arithmetic in Theory and Practice. Computer Science and Applied Methods. Academic Press. New York 1981. 249 pages. ISBN 012428650X.

Kumar, N. & Vemuri, R. 1992. Finite State Machine Verification on MIMD Machines. European Design Automation Conference EURO-VHDL '92 EURO-DAC '92, 7-10 Sep, 1992. IEEE. Pages 514-520. DOI 10.1109/EURDAC.1992.246312.

Kundu, S. 1978. Note on a Constrained-Path Problem in Program Testing. IEEE Transactions on Software Engineering, SE-4(1):75-76.

Kupferman, O. & Vardi, M.Y. 2001. Weak Alternating Automata Are Not That Weak. ACM Transactions on Computational Logic, 2(3): 408-429.

Kurshan, R.P. & Levin, V. & Minea, M. & Peled, D. & Yenigün, H. 2002. Combining Software and Hardware Verification Techniques. Formal Methods in System Design, 21(3):251-280.

Lacroix, P. 2006. RTL-CHECK: A Practical Static Analysis Framework to Verify Memory Safety and More. Master Thesis. Laval University, Quebec.

Ladkin, P. (prepared for the WWW). 1994. Main Commission Aircraft Accident Investigation Warsaw: Report on the Accident to Airbus A320-211 Aircraft in Warsaw, on 14 September 1993. University of Bielefeld, research group of Prof. P. Ladkin, Mar, 1994.
http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html
Retrieved: 6 May, 2008.

Ladkin, P. (prepared for the WWW). 1996. AA965 Cali Accident Report. Near Buga, Colombia, Dec 20, 1995. Original report title: Aircraft accident report. Controlled flight into terrain, American Airlines flight 965, Boeing 757-223, n651aa, near Cali, Colombia, December 20, 1995. Aeronautica Civil of the Republic of Columbia, Bogota, Columbia. University of Bielefeld, research group of Prof. P. Ladkin, Nov, 1996.
sunnyday.mit.edu/accidents/calirep.html
Retrieved: 6 May, 2008.

Ladkin, P. (prepared for the WWW). 1999. UK Air Accident Investigation Board. AAIB Bulletin No 3:95. Ref:EW/C94/9/2. Category: 1.1. 1999. University of Bielefeld, research group of Prof. P. Ladkin.
http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/Institution/AAIB/AAIB-3-95.html
Retrieved: 6 May, 2008.

Lai, C.D. & Xie, M. & Poh, K.L. & Dai, Y.S. & Yang, P. 2002. A Model for Availability Analysis of Distributed Software/Hardware Systems. Information and Software Technology, 44(6): 343-350.

Laitenberger, O. & Atkinson, C. & Schlich, M. & El Eman, K. 2000. An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents. Journal of Systems and Software, 53(2):183-204.

Laitenberger, O. & DeBaud, J.-M. 1997. Perspective-Based Reading of Code Documents at Robert Bosch GMBH. Information and Software Technology 39(11):781-791.

Laitenberger, O. & DeBaud, J.-M. 2000. An Encompassing Life Cycle Centric Survey of Software Inspection. Journal of Systems and Software 50(1):5-31.

Lander, P. & Berbari, E.J. 1989. Optimizing Signal Averaging Methods. 11th Annual International Conference on IEEE Engineering on Medicine and Biological Society. 9-12 Nov, 1989. Pages 19-20. DOI 10.1109/EMBS.1989.95550.

Landi, W. 1992. Undecidability of Static Analysis. ACM Letters on Programming Languages and Systems, 1(4): 323-337.

Lano, J.R. 1990. The N2 Chart. In: Thayer, R.H. & Dorfman, M. (eds.): System and Software Requirements Engineering. Los Alamitos: IEEE Computer Society Press. Pages 244-271. 719 pages. ISBN 0-8186-8921-8. Original: The N2 Chart. TRW Inc. 1977.

Laski, J.W. & Korel, B. 1983. A Data Flow Oriented Program Testing Strategy. IEEE Transactions on Software Engineering, SE-9(3):347-354.

Latif-Shabgahi, G. & Bass, J.M. & Bennett, S. 2004. A Taxonomy for Software Voting Algorithms Used in Safety-Critical Systems. IEEE Transactions on Reliability, 35(3):319-328.

Latronico, E. & Koopman, P. 2001. Representing Embedded System Sequence Diagrams as a Formal Language. Lecture Notes in Computer Science, 2185:302-316. ISBN 3-540-42667-1. Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools, Toronto, Ontario, Canada, 3-5 Oct, 2001.

Lau, M.F. & Yu, Y.T. 2005. An Extended Fault Class Hierarchy for Specification-Based Testing. ACM Transactions on Software Engineering and Methodology, 14(3):247-276.

Lawrance, J. & Abraham, R. & Burnett, M. & Erwig, M. 2006. Sharing Reasoning about Faults in Spreadsheets: An Empirical Study. IEEE Symposium on Visual Languages and Human-Centric Computing, 4-8 Sep, 2006. Pages 35-42. DOI 10.1109/VLHCC.2006.43.

Laycock, G. T. 1993. The Theory and Practice of Specification Based Software Testing. University of Sheffield. Departure of Computer Science. PhD Thesis. Apr, 1993.
Retrieved: 1 Oct, 2008.

Lazzerini, B. & Lopriore, L. 1989. Abstraction Mechanisms for Event-Control in Program Debugging. IEEE Transactions on Software Engineering, 15(7): 890-901.

Le Traon, Y. & Ouabdesselam, F. & Robach, C. & Baudry, B. 2003. From Diagnosis to Diagnosability: axiomatization, measurement and application. Journal of Systems and Software, 65(1):31-50.

L'Ecuyer, P. & Demers, V. & Tuffin, B. 2007. Rare Events, Splitting, and Quasi-Monte Carlo. ACM Transactions on Modeling and Computer Simulation, 17(2): article 9.

Lee, D. & Yannakakis, M. 1996. Principles and Methods of Testing Finite State Machines - A Survey. AT&T Bell Laboratories. Proceedings of the IEEE, 84(8):1090-1123.

Lee, H.-J. & Ma, Y.-S. & Know, Y.-R. 2004. Empirical Evaluation of Orthogonality of Class Mutation Operators. 11th Asian-Pacific Software Engineering Conference, 30 Nov – 3 Dec, 2004. IEEE computer Society, Washington DC, USA. Pages 512-518. ISBN ISSN 1530-1362, 0-7695-2245-9.

Lee, I. & Iyer, R.K. 1993. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. 23th International Symposium on Fault-Tolerant Computing, FTCS-23, Toulouse, France, 22-24 Jun, 1993. ISBN 0-8186-3680-7.

Inhwan Lee & Iyer, R.K. 1995. Software Dependability in the Tandem GUARDIAN System. IEEE Transactions on Software Engineering, 21(5):455-467.

Lee, I. & Iyer, R.K. 2000. Diagnosing Rediscovered Software Problems Using SYMPTOMS. IEEE Transactions on Software Engineering, 26(2):113-127.

Lee, K. & Fishwick, P.A. 1999. OOPM/RT: A Multimodeling Methodology for Real-Time Simulation. ACM Transactions on Modeling and Computer Simulation, 9(2):141-170.

Lerner, S. & Millstein, T. & Rice, E. & Chambers, C. 2005. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. ACM SIGPLAN Notices, 40(1): 364-377.

Lessmann, S. & Baesens, B. & Mues, C. & Pietsch, S. 2008. Benchmarking Classification Models for Software Defect Prediction. IEEE Transactions on Software Engineering, 34(4):485-496.

Leszak, M. & Perry, D.E. & Stoll, D. 2002. Classification and Evaluation of Defects in a Project Retrospective. Journal of Systems and Software, 61(3):173-187.

Leung, H.K.N. 1995. Selective Regression Testing – Assumptions and Fault Detecting Ability. Information and Software Technology, 37(10): 531-537.

Leung, H. & Tse, T.H. & Chan, F.T. & Chen, T.Y. 2000. Test Case Selection with and without Replacement. Information Science, 129(1-4): 81-103.

Leung, K. R. P. H. & Wong, W. & Kee-Yin NG, J. 2003. Generating Test Cases from Class Vectors. Journal of Systems and Software, 66(1):35-46.

Leveson, N.G. 1991. An Empirical Comparison of Software Fault Tolerance and Fault Elimination. IEEE Transactions on Software Engineering, 17(2):173-182.

Levenson, N.G. 1995. Medical Devices: The Therac-25. University of Washington.
From: Leveson, N.: Safeware: System Safety and Computers. Addison Wesley, 1995. 704 Pages. ISBN 9780201119725.
http://sunnyday.mit.edu/papers/therac.pdf

Leveson, N.G. 2000. Intent Specifications: An Approach to Building Human-Centered Specifications. IEEE Transactions on Software Engineering, 26(1):15-35.

Leveson, N. 2001. Evaluating Accident Models Using Recent Aerospace Accidents. Part I: Event-Based Models. [e-document]. 28 Jun, 2001. Software Engineering Research Laboratory, MIT, Aeronautics and Astrophysics Department, Massachusetts.
http://sunnyday.mit.edu/accidents/nasa-report.pdf
Retrieved: 27 Mar, 2007.

Leveson, N.G. 2004. A New Accident Model for Engineering Safety Systems. Safety Science, 42(2):237-270.

Leveson, N.G. & Cha, S.S. & Knight, J.G. & Shimeall, T.J. 1990. The Use of Self Checks and Voting Software Error Detection: An Empirical Study. IEEE Transactions on Software Engineering, 16(4):432-443.

Leveson, N.G. & Pinnel, L.D. & Sandys, S.D. & Koga, S. & Reese, J.D. 1997. Analyzing Software Specifications for Mode Confusion Potential. Workshop on Human Errors and Systems Development, Glasgow, Mar, 1997.

Leveson, N.G. & Stolzy, J.L. 1987. Safety Analysis Using PETRI Nets. IEEE Transactions on Software Engineering, SE-13(3):386-397.

Levitt, S.P. 2004. C++: An Evolving Language. Seventh AFRICON Conference in Africa, Sep 15-17, 2004. 2:1197-1202. ISBN 0-7803-8605-1.

Lew, A. 1982. On the Emulation of Flowcharts by Decision Tables. Communications of the ACM, 52(12): 895-905.

Lew, K.S. & Dillon, T.S. & Forward, K.E. 1988. Software Complexity and Its Impact on Software Reliability. IEEE Transactions on Software Engineering, 14(11):1645-1655.

Li, H. & Jin. M. & Liu, C. & Gao, Z. 2004. Test Criteria for Context-Free Grammars. Proceedings of the 28th Annual International Computer Software and Applications Conference, 28-30 Sep. 2004. Volume 1. IEEE. Pages 300-305. DOI 10.1109/CMPSAC.2004. 1342847.

Li, Y. & Wahl, N.J. 1999. An Overview of Regression Testing. ACM SIGSOFT Software Engineering Notes, 24(1): 69-73.

Li, Z. & Lu, S. & Myagmar, S. & Zhou, Y. 2006. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering, 32(3):176-192.

Li, Z & Zhou, Y. 2005. Bug Localization: PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. ACM SIGSOFT Software Engineering Notes, 30(5):306-315. SESSION: ESEC/FSE 2005. ISBN 1-59593-014-0.

Liang, D. & Chung, P.E. & Huang, Y. & Kintala, C. & Lee, W.-J. & Tsai, T. K. & Wang, C.-Y. 2004. NT_SwiFT: Software Implemented Fault Tolerance on Windows NT. Journal of Systems and Software, 71(1-2):127-141.

Liblit, B. & Aiken, A. & Zheng, A.X. & Jordan, M.I. 2003. Sampled User Executions for Bug Isolation. ICSE First International Workshop on Remote Analysis and Measurement of Software Systems, 26 Sep, 2003.

Lim, J.B. & Hurson, A.R. 2002. Transaction Processing in Mobile, Heterogeneous Database Systems. IEEE Transactions on Knowledge and Data Engineering, 14(6):1330-1346.

Lipton, R.J. 1975. Reduction: A Method of Proving Properties of Parallel Programs. Communications of the ACM, 18(12):717-721.

Littlewood, B. & Miller, D.R. 1989. Conceptual Modeling of Coincident Failures in Multiversion Software. IEEE Transactions on Software Engineering, 15(12):1596-1614.

Littlewood, B. & Popov, P. & Strigini, L. 2001. Modeling Software Design Diversity - a Review. ACM Computing Surveys, 33(2):177-208.

Littlewood, B. & Strigini, L. 1993. Validation of Ultrahigh Dependability for Software-Based Systems. Communications of the ACM, 36(11):69-80.

Littlewood, B. & Wright, D. 1997. Some conservative stopping rules for the operational testing of safety-critical software. IEEE Transactions on Software Engineering, 23(11):673-683.

Liu, C. & Zhang, X. & Han, J. 2008. A Systematic Study of Failure Proximity. IEEE Transactions on Software Engineering, 43(6): 826-843.

Liu, H. & Tan, H.B.K. 2009. Covering Code Behavior on Input Validation in Functional Testing. Information and Software Technology, 51(2): 546-553.

Liu, S. & McDermid, J.A. 1996. A Model-Oriented Approach to Safety Analysis using Fault Trees and a Support System. Journal of Systems and Software, 35(2):151-164.

Liu, S. & Stavridou, V. & Dutertre, B. 1995. The Practice of Formal Methods in Safety-Critical Systems. Journal of Systems and Software, 28(1):77-87.

Liu, W. & Easterbrook, S. & Mylopoulos, J. 2002. Fifth International Conference on the Unified Modeling Language, Dresden, Germany, 1 Oct, 2002. Rule-Based Detection of Inconsistency in UML Models. Workshop on Consistency Problems in UML-Based Software Development.
http://www.cs.toronto.edu/~sme/papers/2002/uml02wl.18.pdf
Retrieved: 13 Oct, 2008.

Livshits, B. & Zimmermann, T. 2005. Bug Localization: DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. ACM SIGSOFT Software Engineering Notes, 30(5):296-305. SESSION: ESEC/FSE 2005. ISSN 0163-5948.

Lo, J.-H. & Huang, C.-Y. & Chen, I.-Y. & Kuo, S.-Y. & Lyu, M.R. 2005. Reliability Assessment and Sensitivity Analysis of Software Reliability Growth Modeling Based on Software Module Structure. Journal of Systems and Software, 76(1):3-13

Lomet, D. & Salzberg, B. 1991. Versioned Backups and Index Concurrency Results of Work-in-Progress. International Workshop on High Performance Transaction Systems, Pacific Grove, CA, USA, Sep, 1991.

Lopes, A. & Wermelinger, M. & Fiadeiro, J.L. 2003. High-Order Architectural Connectors. ACM Transactions on Software Engineering and Methodology, 12(1):64-104.

Lubarsky, R.S. 2006. CZF and Second Order Arithmetic. Annals of Pure and Applied Logic, 141(1-2): 29-34.

Lukasiewicz, T. 2001. Probabilistic Logic Programming with Conditional Constraints. ACM Transactions on Computational Logic, 2(3): 289-339.

Lung, C.-H. & Zaman, M. & Nandi, A. 2004. Applications of Clustering Techniques to Software Partitioning, Recovery and Restructuring. Journal of Systems and Software, 73(2):227-244.

Luqi & Zhang, L. & Berzins, V. & Qiao, Y. 2004. Documentation Driven Development for Complex Real-Time Systems. IEEE Transactions on Software Engineering, 30(12):936-952.

Lutz, R.R. 1993. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. Proceedings of the IEEE International Symposium on Requirements Engineering, Jan 4-6, 1993.

Lutz, R.R. 1996. Targeting Safety-Related Errors during Software Requirements Analysis. Journal of Systems and Software, 34(3), 1996:223-320.

Lutz, R.R. & Mikulski, I.C. 2003. Operational Anomalies as a Cause of Safety-Critical Requirements Evolution. Journal of Systems and Software, 65(2):155-161.

Lutz, R.R. & Mikulski, I.C. 2004. Empirical Analysis of Safety-Critical Anomalies during Operations. IEEE Transactions on Software Engineering, 30(3):172-180.

Lutz, R.R. & Woodhouse, R.M. 1996. Contributions of SFMEA to Requirements Analysis. Proceedings of the Second International Conference on Requirements Engineering, Colorado Springs, CO, USA, 15-18 Apr, 1996. Pages 44-51. ISBN 0-8186-7252-8.

Lyu, M.R. & Nikora, A. 1992. Applying Reliability Models More Effectively. IEEE Software, 9(4): 43-52.

Macqueen, D.B. & Sannella, D.T. 1985. Completeness of Proof Systems for Equational Specifications. IEEE Transactions on Software Engineering, SE-11(5):454-461.

Madria, S.K. & Maheshwari, S.H. & Chandra, B. & Bhargava, B. 2000. An Open and Safe Nested Transaction Model: Concurrency and recovery. Journal of Systems and Software 55:151-165.

Maghrabi, T. & Golshani, F. 1992. Automatic Program Generation Using Sequent Calculus. Proceedings of the 1992 ACM Annual Conference on Communications, Kansas City, MO, USA, 3-5 Mar, 1992. Pages 73-82. ISBN 0-89791-472-4.

Makarov, Y. 2006. Practical Program Extraction from Classical Proofs. Electronic Notes in Theoretic Computer Science, 155:521-542.

Makowsky, J.A. 1992. Model Theory and Computer Science: An Appetizer. In: Abramsky, S. & Gabbay, D.M. & Maibaum, T.S.E. (eds.): Handbook of logic in computer science. Vol. 1 : Background : Mathematical Structures. Oxford: Clarendon Press. 840 pages. Pages 763-814. ISBN 0198537352.

Malaiya, Y.K. 1995. Antirandon Testing: Getting the Most out of Black-Box Testing. IEEE Proceedings on the 6th International Symposium on Software Reliability Engineering, Toulouse, France, 24-27 Oct, 1995. Pages 86-95. ISBN 0-8186-7131-9.

Malevris, N. 1995. A Path Generation Method for Testing LCSAJs That Restrains Infeasible Paths. Information and Software Technology 37(8):435-441.

Malhart, B.E. 1995. Software Fault Tree Analysis for Requirements System Model. Proceedings of the 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, Tuscon, AZ, USA, 6-9 Mar, 1995. Pages 133-140. ISBN 0-7803-2531-1.

Maples, W. (last modified). 2004. Admin Tips >> Windows Server 2008/2003/2000/XP/NT Administration Knowledge Base >> Windows NT >> Admin Tips >> Utilities >> Windows Freeware exe, dll, ocx, sys Dependency Walker. Windowsnetworking.com. Copyright TechGenix Ltd. Last modified: 20 Apr, 2004. http://www.windowsnetworking.com/kbase/WindowsTips/WindowsNT/AdminTips/Utilities/WindowsFreewareexedllocxsysDependencyWalker.html
Retrieved: 7 Oct, 2008.

Marick, B. 1997. Classic testing Mistakes. [e-document]. Testing Foundations.
http://www.testing.com/writings/classic/mistakes.pdf
Retrieved: 22 Mar, 2007.

Marin, M. & Deursen, A.V. & Moonen, L. 2007. Identifying Crosscutting Concerns Using Fan-In Analysis. ACM Transactions on Software Engineering and Methodology, 17(1), article 3, 17 pages.

Marinov, D. & Andoni, A. & Daniliuc, D. & Khurshid, S. & Rinard, M. 2003. An Evaluation of Exhaustive Testing for Data Structures. MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA. Technical Report, MIT-LCS-TR-92. Sep, 2003.

Marinov, D. & Khurshid, S. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. Proceedings of the 16th Annual International Conference on Automated Software Engineering, 26-29 Nov, 2001. IEEE. Pages 22-31.

Markowitz, V. & Shoshani, A. 1989. On the Correctness of Representing Extended Entity-Relationship Structures in the Relational Model. ACM SIGMOD Record, 18(2): 430-439.

Marlowe, T.J. & Ryder B.G. 1989. An Efficient Hybrid Algorithm for Incremental Data Flow Analysis. Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Pages 184-196. ISBN 0-89791-343-4.

Marré, M. & Bertolino, A. 2003. Using Spanning Sets for Coverage Testing. IEEE Transactions on Software Engineering 29(11): 974-984.

Martin, M. & Livshits, B. & Lam, M.S. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. ACM SIGPLAN Notices. Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 40(10):365-383. SESSION: Tracing traces. ISSN 0362-1340.

Mauborgne, L. 2003. Infinity Relations and Their Representation. Science of Computer Programming, 47(2-3):121-144.

Maxion, R.A. & Olszewski, R.T. 2000. Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study. IEEE Transactions on Software Engineering, 26(9):888-906.

McGeoch, C. 1992. Analyzing Algorithms by Simulation: Variance Reduction Techniques and Simulation Speedways. ACM Computing Surveys, 24(2):195-212.

McKeeman, W.M. 1975. On Preventing Programming Language from Interfering with Programming. IEEE Transactions on Software Engineering, 1(1):19-26.

MCOMIB. 1999. Mars Climate Orbiter Mishap Investigation Board: Phase 1 report. 10 Nov, 1999. NASA MCO Mission Failure Mishap Investigation Board.
ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf
www.space.com/media/mco_report.pdf
Retrieved: 9 May, 2008.

Medvidovic, N. & Grünbacher, P. & Egyed, A. & Boehm, B.W. 2003. Bridging Models Across the Software Lifecycle. Journal of Systems and Software, 68(3):199-215.

Menzies, T. & Greenwald, J. & Frank, A. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. IEEE Transactions on Software Engineering, 33(1): 2-13.

Meunier, J.G, & Biskri, I. & Forest, D. 2005. A Model for Computer Analysis and Reading of Text (CARAT): The SATIM Approach. TEXT Technology. The Journal of Computer TEXT Processing, 14(2). The Society of Digital Humanities, Canada. Pages 123-152.
http://texttechnology.mcmaster.ca/pdf/vol14_2/meunier14-2.pdf
Retrieved: 23 Oct, 2008.

Meyer, B. 2003. A Framework for Proving Contract-Equipped Classes. Abstract State Machines. Lecture Notes in Computer Science. Springer-Verlag. Pages 108-125. ISBN 978-3-540-00624-4.

Michael, C.C. & Jones, R.C. 1996. On the Uniformity of Error Propagation in Software. RST Corporation, Sterling, VA, USA. Technical Report RSTR-96-003-4. 5. Nov, 1996.

MIL-STD 882B. 1984. System Safety Program Requirements. Military Standard, 30 Mar, 1984. AMSC number F3329 FSC SAFT. Department of Defence, Washington DC, USA.

Mill, A. 1985. Towards a Theory of Forward Error Recovery. IEEE Transactions on Software Engineering, SE-11(8):735-748.

Miller, B.P. & Fredriksen, L. & So, B. 1990. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 33(12):32-44.

Miller, B.P. & Koski, D. & Lee, C.P. & Maganty, V. & Murthy, R. & Matarajan, A. & Steidl, J. 1995. Fuzz Revisited: A Re-Examination of the Reliability of UNIX Utilities and Services. The University of Wisconsin, USA, Computer Science Department, CS-TR-1995-1268.

Miller, D. 1994. A Multiple-Conclusion Meta Logic. Proceedings of the 1994 Symposium on Logic in Computer Science, Paris, France, 4-7 Jul, 1994. IEEE. Pages 272-281. DOI 10.1109/LICS.1994.316062.

Miller, J. 2000. Applying Meta-Analytical Procedures to Software Engineering Experiments. Journal of Systems and Software, 54(1):29-39.

Miller, J. & Roper, M. & Wood, M. & Brooks, A. 1995. Towards a Benchmark for the Evaluation of Software Testing Techniques. Information and Software Technology 37(1):5-13.

Miller, K.W. & Morell, L.J. & Noonan, R.E. & Park, S.K. & Nicol, D.M. & Murrill, B.W. & Voas, M. 1992. Estimating the Probability of Failure When Testing Reveals No Failures. IEEE Transactions on Software Engineering, 18(1):33-43.

Mitchell, B. & Zeil, S.J. 1996. A Reliability Model Combining Representative and Directed Testing. Proceedings of the 18th International Conference on Software Engineering. 25-30 Mar, 1996. IEEE. DOI 10.1109/ICSE.1996.493445.

Modugno, F. & Leveson, N.G. & Reese, J.D. & Partridge, K. & Sandys, S.D. 1997. Integrated Safety Analysis of Requirements Specifications. Proceedings of the 3rd International Symposium on Requirements Engineering, Annapolis, MD, USA, 6-10 Jan, 1997. Pages 148-159. ISBN 0-8186-7740-6.

Moher, T.G. 1988. PROVIDE: A Process Visualization and Debugging Environment. IEEE Transactions on Software Engineering, 14(6): 849-857.

Mohri, Y. & Kikuno, T. 1991. Fault Analysis Based on Fault Reporting in JSP Software Development. Proceedings of the 15th Annual International Computer Software and Applications Conference COMPSAC 1991, Tokyo, Japan, 11-13 Sep, 1991. Pages 591-596. ISBN 0-8186-2152-4.

Mok, A.K. & Konana, P. & Guangtian Liu & Chan-Gun Lee & Honguk Woo. 2004. Specifying Timing Constraints and Composite Events: An Application in the Design of Electronic Brokerages. IEEE Transactions on Software Engineering, 30(12):841-858.

Moonen, L. 1997. A Generic Architecture for Data Flow Analysis to Support Reverse Engineering. 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, Holland, 25-26 Sep, 1997. Springer. ISBN 3-540-76228-0.

Morasca, S. 2002. A Proposal for Using Continuous Attributes in Classification Trees. Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, Aschia, Italy, Jul 15-19, 2002. SESSION: Measurement and Empirical Software Engineering. ACM International Conference Proceeding Series, 27:417-424. ACM Press, New York, NY, USA. ISBN 1-58113-556-4.

Morasca, S. & Ruhe, G. 2000. A Hybrid Approach to Analyze Empirical Software Engineering Data and its Application to Predict Module Fault-Proness in Maintenance. Journal of Systems and Software, 53(3):225-237.

Morell, L.J. 1990. A Theory of Fault-Based Testing. IEEE Transactions on Software Engineering, 16(8):844-857.

Moret, B.M.E. 1982. Decision Trees and Diagrams. ACM Computing Surveys 14(4):593-623.

Morris Jr., J.H. & Wegbreit, B. 1977. Subgoal Induction. Communications of the ACM 20(4):209-222.

Morris, J.M. & Bunkenburg, A. 2002. A Source of Inconsistency in Theories of Nondeterministic Functions. Science of Computer Programming 43(1):77-89.

Mossé, D. & Melhem, R. & Sunondo Ghosh. 2003. A Nonpreemptive Real-Time Scheduler with Recovery from Transient Faults and Its Implementation. IEEE Transactions on Software Engineering, 29(8): 752-767.

Motteler, H. & Chung, A. & Sidhu, D. 1995. Undetected Faults in Protocol Testing. IEEE Transactions on Communications, 43(8):2289-2297.

Moynihan, T. 2000.: Coping with 'requirements-uncertainty': the theories-of-action of experienced IS/software project managers. Journal of Systems and Software, 53(2):99-109.

Muccini, H. & Bertolino, A. & Inverardi, P. 2004. Using Software Architecture for Code Testing. IEEE Transactions on Software Engineering, 30(3):160-171.

Mullen, R.E. 1998. The Lognormal Distribution of Software Failure Rates: Origin and Evidence. The Ninth International Symposium on Software Reliability Engineering, 4-7 Nov, 1998. IEEE. Pages 124-133.

Mulvihill, R.J. 1988. Design-Safety Enhancement Through the Use of Hazard and Risk Analysis. IEEE Transactions on Reliability, 37(2):149-158.

Munoz, C.U. 1988. An Approach to Software Product Testing. IEEE Transactions on Software Engineering, 14(11):1589-1596.

Munson, J.C. & Khoshgoftaar, T.M. 1992. The Detection of Fault-Prone Programs. IEEE Transactions on Software Engineering, 18(5):423-433.

Munson, S. 1999. Assessment of Accident Investigation Methods for Wildland Firefighting Incidents by Case Study Method. Thesis. University of Montana.
http://www.iprr.org/3PROJ/Munpaper.html
Retrieved: 24 Mar, 2007.

Murata, T. 1989. Petri Nets: Properties, Analysis, and Applications. Proceedings of the IEEE, 77(4): 541-580.

Murphy, G.C. & Notkin, D. & Sullivan, K.J. 2001. Software Reflexion Models: Bridging the Gap Between Design and Implementation. IEEE Transactions on Software Engineering, 27(4):364-380.

Murrill, B.W. 2008. An Empirical Path-oriented Approach to Software Analysis and Testing. Journal of Systems and Software 81(2): 249-261.

Musa, J.D. & Okumoto, K. 1984. A Logarithmic Poisson Execution Time Model for Software Reliability Measurement. IEEE Proceedings of the 7th International Conference on Software Engineering, Orlando, Florida, USA, 26-29 Mar, 1984. IEEE Press, Piscataway, NJ, USA. Pages 230-238. ISBN ISSN 0270-5257, 0-8186-0528-6.

Myrtveit, I. & Stensrud, E. & Shepperd, M. 2005. Reliability and Validity in Comparative Studies of Software Prediction Models. IEEE Transactions on Software Engineering, 31(5):380-391.

Naixin, L. & Malaiya, Y.K. 1994. On Input Profile Selection for Software Testing. Proceedings of the 5th International. Symposium on Software Reliability Engineering, Monterey, CA, USA, 6-9 Nov, 1994. Pages 196-205. ISBN 0-8186-6665-X.

Nakajo, T. & Azuma, I. & Tada, M. 1993. A Case History Development of a Foolproofing Interface Documentation System. IEEE Transactions on Software Engineering, 19(8):765-773.

Nakajo, T. & Kume, H. 1991. A Case History Analysis of Software Error Cause-Effect Relationships. IEEE Transactions on Software Engineering, 17(8):830-838.

Nakashima, T. & Oyama, M. & Hisada, H. & Ishii, N. 1999. Analysis of Software Bug Causes and its Prevention. Information and Software Technology, 41(15):1059-1068.

NASA (National Aeronautics and Space Administration). 1999. MISR Multi-Angle Imaging SpectroRadiometer. Status 27 Dec, 1999. By Diner, D. NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA.
http://www-misr.jpl.nasa.gov/news/1999/news122799.html
Retrieved: 9 May, 2008.

NASA. 2003. Space Shuttle Launch Archive. Space Shuttle Launch Archive last modified 1 Feb, 2003. Web page author: Dumoulin, J. Kennedy Space Center, Cape Canaveral, FL, USA.
http://science.ksc.nasa.gov/shuttle/missions/sts-missions-flown-01.txt
Retrieved: 9 May, 2008.

NASA. 2006. The ROSAT Mission (1990-1999). ROSAT Guest Observer Facility. Last modified: 7 Sep, 2006. Heasarc (High Energy Astrophysics Science Archive Research Center). Astrophysics Science Division at the NASA Gobbard Space Flight Center; and at the High Energy Astrophysics Division of the Smithsonian Astrophysical Observatory, Cambridge, MA.
http://heasarc.gsfc.nasa.gov/docs/rosat/rosgof.html
Retrieved: 9 May, 2008.

NASA. 2007. Almost perfect. System Failure Case Studies 1(3). [case study report on-line]. Jan, 2007. Process Based Mission Assurance Knowledge Management System.
pbma.hq.nasa.gov/pbma_main_cid_582
Retrieved: 5 May, 2008.

Naumov, P. 2006. Logic of Subtyping. Theoretical Computer Science, 357: 167-185.

Naumovich, G. & Avrunim, G.S. & Clarke, L.A. 1999. Data Flow Analysis for Checking Properties of Concurrent Java Programs. Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, USA, 16-22 May, 1999. IEEE Computer Society, Los Alamitos, CA, USA. Pages 399-410. ISBN 1-58113-074-0.

Navarro, I. & Leveson, N.G. & Lundqvist, K. 2001. Reducing the Effects of System Changes through System Design. MIT SERL Technical Report. Massachusetts Institute of Technology, USA.

NEAR Anomaly Rendezvous Board. 1999. The NEAR Rendezvous Burn Anomaly of December 1998. Final Report of the NEAR (Near Earth Asteroid Rendezvous) Anomaly Review Board. The Johns Hopkins University Applied Physics Laboratory. Nov 1999.
http://near.jhuapl.edu/anom/Hoffman.pdf
Retrieved: 9 May, 2008.

Negrini, R.M. & Sami, M. 1983. Some Properties Derived From Structural Analysis of Program Graph Models. IEEE Transactions on Software Engineering, SE-9(2):172-178.

Neumann, D.E. 2002. An Enhanced Neural Network Technique for Software Risk Analysis. IEEE Transactions on Software Engineering, 28(9):904-912.

Neumann, P.G. 1985. Computer-Related Incidents Illustrating Risks to the Public. In: P.G. Neumann (Moderator). Risk Digest. Forum on Rules to Public in Computers and Related Systems. ACM Committee on Computers and Public Policy. 1(1).
http://catless.ncl.ac.uk/Risks
Retrieved: 24 Apr, 2007.

Neumann, P.G. 1986. On Hierarchical Design of Computer Systems for Critical Applications. IEEE Transactions on Software Engineering, SE-12(9):905-920.

Neumann, P.G. 2007. Illustrative Risks to the Public in the Use of Computer Systems and Related Technology. [e-document]. Computer Science Laboratory. SRI International, Menlo Park, CA, USA.
http://www.csl.sri.com/users/neumann/illustrative.html#6
Retrieved: 24 Mar, 2007.

Ng, S.H. & Chick, S.E. 2001. Analysis Methodology: Reducing Input Parameter Uncertainty for Simulations. Proceedings of the 33th International Conference on Winter Simulation. Arlington, Virginia, USA, 9-12 Dec, 2001. SESSION: Analysis Methodology. IEEE Computer Society, Washington DC, USA. Pages 364-371. ISBN 0-7803-7309-X.

Ngo, M.N. & Tan, H.B.K. 2008. Heuristic-Based Infeasible Path Detection for Dynamic Test Data Generation. Information and Software Technology, 50(7-8): 641-655.

Nicola, V.F. & Goyal, A. 1990. Modeling of Correlated Failures and Community Error Recovery in Multiversion Software. IEEE Transactions on Software Engineering, 16(3): 350-359.

Nicola, V.F. & van Spanje, J.M. 1990. Comparative Analysis of Different Models of Checkpointing and Recovery. IEEE Transactions on Software Engineering, 16(8):807-821.

Nicollin, X. & Sifakis, J. & Yovine, S. 1992. Compiling Real-Time Specifications into Extended Automata. IEEE Transactions on Software Engineering, 18(9): 794-804.

Nikolik, B. 2005. Convergence Debugging. ACM Proceedings of the Sixth International Symposium on Automatic Analysis-Driven Debugging, Monterey, CA, USA, 19-21 Sep, 2005. ACM Press, New York, NY, USA. Pages 89-98. ISBN 1-59593-050-7.

Nikolik, B. 2006. Test Diversity. Information and Software Technology, 48(11):1083-1094.

Ntafos, S.C. 1988. A Comparison of Some Structural Testing Strategies. IEEE Transactions on Software Engineering, 14(6):868-874.

Ntafos, S.C. 2001. On Comparisons of Random, Partition, and Proportional Partition. IEEE Transactions on Software Engineering, 27(10):949-960.

Ntafos, S.C. & Hakimi, S.L. 1979. On Path Cover Problems in Digraphs and Applications to Program Testing. IEEE Transactions on Software Engineering, SE-5(5):520-529.

NTSB. 1980. Aircraft Incident Report. AEROMEXICO DC-10-30, XA-DUH Over Luxembourg, Europe, Nov 11, 1979. NTSB-AAR-80-10. Washington DC, USA. Nov, 1980.
http://www.airdisaster.com/reports/ntsb/AAR80-10.pdf
Retrieved: 9 May, 2008.

Nussbacher, H. 1992. SEN "Horror Stories". [e-message on discussion board]. 16. Mar, 1992. Last changed: 31 Jul, 1993. Bar-Ilan University Computing Center, Israel. Contains ACM SEN VMshare material, contributors: Calender, D. & Corp, A. Web page maintained by Lamb, D.A., Queens University, Kingston, Ontario, Canada.
http://www.cs.queensu.ca/Software-Engineering/archive/horror
Retrieved: 24 Mar, 2007.

Oehlert, P. 2005. Violating Assumptions with Fuzzing. IEEE Security and Privacy 3(2): 58-62.

Offutt, A.J. 1992. Investigations of the Software Testing Coupling Effect. ACM Transactions on Software Engineering and Methodology, 1(1):5-20.

Offutt, A.J. & Abdurazik, A. & Alexander, R.T. 2000. An Analysis Tool for Coupling-Based Integration Testing. Proceedings of the sixth IEEE International Conference on Engineering of Complex Computer Systems, 11-14 Sep, 2000. IEEE. Pages172-178. DOI 10.1109/ICECCS.2000.873942.

Offutt, A.J. & Lee, A. & Rothermel, G. & Untch, R.H. & Zapf, C. 1996. An Experimental Determination of Sufficient Mutant Operators. ACM Transactions on Software Engineering and Methodology, 5(2): 99-118.

Ohba, M. 1982. Software Quality = Test Accuracy x Test Coverage. Proceedings of the 6th International Conference on Software Engineering, Tokyo, Japan, 13-16 Sep, 1982. IEEE Computer Society Press, Los Alamitos, CA, USA. P. 287-293. ISSN 0270-5257.

Okun, V.. & Black, P.E. & Yesha, Y. 2004. Comparison of Fault Classes in Specification-Based Testing. Information and Software Technology 46(8):525-533.

Olender, K.M. & Osterweil, L.J. 1992. Interprocedural Static Analysis of Sequencing Constraints. ACM Transactions on Software Engineering and Methodology, 1(1):21-52.

Orso, A. & Sinha, S. & Harrold, M.J. 2004. Classifying Data Dependences in the Presence of Pointers for Program Comprehension, Testing, and Debugging. ACM Transactions on Software Engineering and Methodology, 13(2):199-239.

Ostrand, T.J. & Balcer, M.J. 1988. The Category-Partition Method for Specifying and Generating Functional Tests. Communications of the ACM 31(6):676-686.

Ostrand, T.J. & Weyuker, E.J. 2002. The Distribution of Faults in a Large Industrial Software System. ACM SIGSOFT Software Engineering Notes, Proc 2002 ACM SIGSOFT international symposium on Software testing and analysis, 27(4), Jul Roma, Italy, 22-24 Jul, 2002. SESSION: Faults and Failure Analysis. ACM Press, New York, USA. Pages 55-64. ISBN ISSN 0163-5948, 1-58113-562-9.

Ostrand, T.J. & Weyuker, E.J. & Bell, R.M. 2005. Predicting the Location and Number of Faults in Large Software Systems. IEEE Transactions on Software Engineering, 31(4):340-355.

Ostroff, J.S. 1992. Formal Methods for the Specification and Design of Real-Time Safety Critical Systems. Journal of Systems and Software, 18(1):33-60.

Ou, Y. & Dugan, J.B. 2000. Sensitivity Analysis of Modular Dynamic Fault Trees. Proceedings on IEEE International Computer Performance and Dependability Symposium, IPDS, 27-30 Mar, 2000. IEEE Computer Society, Washington DC, USA. Pages 35-43. ISBN 0-7695-0553-8.

Owre, S. & Rushby, J. & Snahkar, N. & von Henke, F. 1995. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. IEEE Transactions on Software Engineering, 21(2): 107-125.

Oxford English Dictionary IV. 1989. Oxford English Dictionary. Simpson, J.A. & Weiner, E.S.C. (preparers). Second edition, vol. IV. Cladenron Press. Oxford 1989.
ISBN 0-19-861216-2. Set ISBN: 0-19-861186-2.

Oxford English Dictionary VI. 1989. Oxford English Dictionary. Simpson, J.A. & Weiner, E.S.C. (preparers). Second edition, vol. VI. Cladenron Press. Oxford 1989.
ISBN 0-19-861218-4. Set ISBN: 0-19-861186-2.

Ozsoyoglu, G. & Wang, H. 1989. A Relational Calculus with Set Operations, Its Safety, and Equivalent Graphical Languages. IEEE Transactions on Software Engineering, 15(9): 1038-1052.

Padberg, F. & Ragg, T. & Schoknecht R. 2004. Using Machine Learning for Estimating the Defect Content After an Inspection. IEEE Transactions on Software Engineering, 30(1):17-28.

Palsberg, J. 1998. Equality-Based Flow Analysis Versus Recursive Types. ACM Transactions on Programming Languages and Systems, 20(6):1251-1264.

Pan, J. 1999. The Dimensionality of Failures - A Fault Model for Characterizing Software Robustness. Proceedings of the Fault Tolerant Computer Symposium, Madison, WI, USA, 15-18 Jun, 1999.

Paradkar, A. 2005. Case Studies on Fault Detection Effectiveness of Model Based Test Generation Techniques. ACM SIGSOFT Software Engineering Notes. SESSION: Advances in Model-Based Testing (A-MOST 2005). 30(4):1-7. ISBN 1-59593-115-5.

Parker, D.S. 1989. Partial Order Programming (Extended Abstract). Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: 260-266.

Parnas, D.L. 1993. Predicate Logic for Software Engineering. IEEE Transactions on Software Engineering, 19(9):856-862.

Parrish, A. & Zweben, S.H. 1991. Analysis and Refinement of Software Test Data Adequacy Properties. IEEE Transactions on Software Engineering, 17(6):565-581.

Pasquini, A. & De Agostino, E. & Di Marco, G.D. 1996. An Input-Domain Based Method to Estimate Software Reliability. IEEE Transactions on Reliability, 45(1):95-105.

Payne, J. 2005. University of Edinburgh Logic 1 - Tutorial Handout 1. Connectives, Formulae, and Scope.
http://homepages.ed.ac.uk/s0344154/teaching/logic/guides/handout1.pdf
Retrieved: 24 Oct, 2008.

Pedreschi, D. & Ruggieri, S. 2003. On Logic Programs that Always Succeed. Science of Computer Programming, 48(2-3):163-196.

Pedrycz, W. & Succi, G. 2005. Genetic Granular Classifiers in Modeling Software Quality. Journal of Systems and Software, 76(3):277-285.

Peleg, M. & Dori, D. 2000. The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods. IEEE Transactions on Software Engineering, 26(8):742-759.

Peng, W.W. & Wallace, D.R. 1993. Software Error Analysis. U.S. Department of Commerce. Technology Administration. National Institute of Standards and Technology. Computer Laboratory. Gaithersburg, MD, USA. NIST Special Publication 500-209. Mar, 1993.

Perry, D.E. & Wolf, A.L. 1992. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17(4):40-52. ACM Press, New York, NY, USA. ISSN 0163-5948.

Petersson, H. & Thelin, T. & Runeson, P. & Wohlin, C. 2004. Capture-Recapture in Software Inspection after 10 Years Research - Theory, Evaluation and Application. Journal of Systems and Software, 72(2):249-264.

Petrenko, A. & Boroday, S. & Groz, R. 2004. Confirming Configurations in EFSM Testing. IEEE Transactions on Software Engineering, 30(1):29-42.

Pettorossi, A. & Proietti, M. 2004. A Theory of Totally Correct Logic Program Transformations. Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, Verona, Italy, 24-25 Aug, 2004. Pages 159-168.

Pfleeger, S.L. 2000. Risky Business: What we Have Yet to Learn About Risk Management. Journal of Systems and Software, 53(3):265-273.

Phillips, I.C.C. 1992. Recursion Theory. In: Abramsky, S. & Gabbay, D.M. & Maibaum, T.S.E. (eds.): Handbook of logic in computer science. Vol. 1 : Background : Mathematical Structures. Oxford: Clarendon Press. 840 pages. Pages 763-814. ISBN 0198537352.

Phillips, N.C.K. 1984. Safe Data Type Specifications. IEEE Transactions on Software Engineering, 10(3):285-289.

Pighin, M. & Marzona, A. 2003. An Empirical Analysis of Fault Persistence through Software Releases. Proceedings on the 2003 International Symposium on Empirical Software Engineering, 30 Sep – 1 Oct, 2003. IEEE Computer Society, Washington DC, USA. P. 206-212. ISBN 0-7695-2002-2.

Pillai, K. & Nair, V.S.S. 1997. Statistical Analysis of Nonstationary Software Metrics. Information and Software Technology, 39(5): 363-373.

Pitt, D.H. & Shields, M. 2002. Local Invariance. Formal Aspects of Computing, 14(1):35-54.

Podgurski, A. & Clarke, L.A. 1990. A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance. IEEE Transactions on Software Engineering, 16(9):965-979.

Podgurski, A. & Masri,W. & Mccleese, Y. & Wolff, F.G. & Yang, C. 1999. Estimation of Software Reliability by Stratified Sampling. ACM Transactions on Software Engineering and Methodology, 8(3): 263-283.

Pohjolainen, P. 2002. Software Testing Tools. The University of Kuopio.
http://www.cs.uku.fi/research/Teho/SoftwareTestingTools.pdf
Retrieved: 9 May, 2008.

Poigné, A. 1992. Basic Category Theory. In: Abramsky, S. & Gabbay, D.M. & Maibaum, T.S.E. (eds.): Handbook of logic in computer science. Vol 1. Pages 416-640. ISBN 0198537352.

Pop, P.C. 2002. The Generalized Minimum Spanning Tree Problem. PhD Thesis. University of Twente. Twente University Press, Enschede, Netherlands. ISBN 9036517850.
http://www.ub.utwente.nl/webdocs/tw/1/t0000021.pdf
Retrieved: 14 Oct, 2008.

Porter, A. & Siy, H. & Mockus, A. & Votta, L. 1998. Understanding the Sources of Variation in Software Inspections. ACM Transactions on Software Engineering and Methodology 7(1): 41-79.

Porter, A.A. & Siy, H.P. & Toman, C.A. & Votta, L.G. 1997. An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development. IEEE Transactions on Software Engineering, 23(6): 329-346.

Porter, A.A. & Votta, L.G., Jr. & Basili, V.R. 1995. Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment. IEEE Transactions on Software Engineering, 21(6):563-575.

Porwal, R. & Gursaran. 2004. An Experimental Evaluation of Weak-Branch Criterion for Class Testing. Journal of Systems and Software, 70(1-2):209-224.

Potgieter, P.H. 2006. Zeno Machines and Hypercomputation. Theoretical Computer Science, 358(1):23-33.

Powell, D. 1992. Failure Mode Assumptions and Assumption Coverage. 22nd International Symposium on Fault Tolerant Computing, Boston, MA, USA, 8-10 Jul, 1992. IEEE Computer Society. Pages 386-395. ISBN 0-8186-2875-8.

Prasad, D. 2006. SLAM and BLAST. Model Checking Tools. In: Strunk, E.A. & Aiello, A. & Knight, J.C. (eds.): A Survey of Tools for Model Checking and Model-Based Development. Department of Computer Science. University of Virginia. Technical Report CS-2006-17. Pages 5-19.

Prasad D. & McDermid, J. 1999. Dependability Evaluation using a Multi- Criteria Decision Analysis Procedure. Proceedings of the Conference on Dependable Computing for Critical Applications, 6-8 Jan, 1999. IEEE Computer Society, Washington DC, USA. Pages 339-358. ISBN 0-7695-0248-9.

Prather, R.E. & Myers, J.P., Jr. 1987. The Path Prefix Software Testing Strategy. IEEE Transactions on Software Engineering, SE-13(7):761-766.

Preparata, F.P. & Metze, G. & Chien, R.T. 1967. On the Connection Assignment Problem of Diagnosable Systems. IEEE Transactions on Electronic Computers EC-16(6):848-854.

Pretschner, A. & Lötzbeyer, H. & Philipps, J. 2004. Model Based Testing in Incremental System Development. Journal of Systems and Software, 70(3):315-329.

Probert, R.L. 1982. Optimal Insertion of Software Probes in Well-Delimited Programs. IEEE Transactions on Software Engineering, SE-8(1):34-42.

Pucci, G. 1992. A New Approach to the Modeling of Recovery Block Structures. IEEE Transactions on Software Engineering, 18(2):159-167.

QADownLoads
Quality Assurance and Software Testing Downloads.
http://www.qadownloads.com/Tools/

Qin, F. & Tucek, J. & Sundaresan, J. & Zhou, Y. 2005.
Rx: Treating Bugs as Allergies---A Safe Method to Survive Software Failures. ACM SIGOPS Operating Systems Review, Proceedings of the 20th ACM Symposium on operating systems principles, Brighton, UK, 2005. SESSION. bugs. ACM Press, New York, NY, USA. 39(5):235-248. ISSN 0163-5980.

Rai, A. & Song, H. & Troutt, M. 1998. Software Quality Assurance: An Analytical Survey and Research Prioritization. Journal of Systems and Software, 40(1):67-83.

Rainer, A. & Hall, T. 2003. A Quantitative and Qualitative Analysis of Factors Affecting Software Processes. Journal of Systems and Software, 66(1):7-21.

Rallis, N.E. & Lansdowne, Z.F. 2001. Reliability Estimation for a Software System with Sequential Independent Reviews. IEEE Transactions on Software Engineering, 27(12):1057-1061.

Ramamoorthy, C.V. & Bastani, F.B. 1982. Software Reliability - Status and Perspective. IEEE Transcations on Software Engineering, SE-8(4):354-371.

Ramamoorthy, C.V. & Garg, V.K. & Prakash, A. 1986. Programming in the Large. IEEE Transactions on Software Engineering, 12(7):769-783.

Ramesh, B. & Edwards, M. 1993. Issues in the Development of a Requirements Traceability Model. Proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, CA, USA, 4-6 Jan, 1993. Pages 256-259. ISBN 0-8186-3120-1.

Raymond, E.S. 2003. The Jargon File, version 4.4.7. Schroedingbug. 29 Dec, 2003.
http://catb.org/jargon/html/S/schroedinbug.html
Retrieved: 28 Oct, 2008.

Ravn, A.P. & Rischel, H. & Hansen, K.M. 1993. Specifying and Verifying Requirements of Real-Time Systems. IEEE Transactions on Software Engineering, 19(1):41-55.

Redwine, S.T., Jr. 1983. An Engineering Approach to Software Test Data Design. IEEE Transactions on Software Engineering, SE-9(2):191-200.

Reese, J.D. & Leveson, N.G. 1997. Software Deviation Analysis. International Conference on Software Engineering, Boston, MA, 17-13 May, 1997.
http://sunnyday.mit.edu/papers/sda.pdf
Retrieved: 9 May, 2008.

Regnell, B. & Runeson, P. & Wohlin, C. 2000. Towards Integration of Use Case Modelling and Usage-Based Testing. Journal of Systems and Software, 50(2):117-130.

Reid, W.S. 1995. Tales from the Crucible Secrets of a High Tech Expert. [Professor Huckle's webpage]. University of Munich, Germany.
http://www5.in.tum.de/~huckle/trenches.htm
Retrieved: 24 Mar, 2007.

Reps, T. 2000. Undecidability of Context-Sensitive Data-Independence Analysis. ACM Transactions on Programming Languages and Systems, 22(1):162-186.

Richardson, D.J. & Thompson, M.C. 1993. An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection. IEEE Transactions on Software Engineering, 19(6):533-553.

Rine, D. 1996. Sharing Objects as Structural Defects in Object-Oriented Programming without Safe Typing. Information and Software Technology 38(7):451-453.

Robillard, M.P. 2008. Topology Analysis of Software Dependencies. ACM Transactions on Software Engineering and Methodology, 17(4), article 8, 36 pages.

Robillard, M.P. & Coelho, W. & Murphy, G.C. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. IEEE Transactions on Software Engineering, 30(12):889-903.

Robillard, M.P. & Murhpy, G.C. 2007. Representing Concerns in Source Code. ACM Transactions on Software Engineering and Methodology, 16(1), article 3, 38 pages.

Romanovsky, A. & Strigini, L. 1995. Backward Error Recovery via Conversations in Ada. IEEE Software Engineering Journal, 10(6): 219-232.

Romero. M. 2005. Introduction to Predicate Logic. 24 Feb, 2005. The University of Pennsylvania. The Department of Linguistics.
http://babel.ling.upenn.edu/courses/ling255/PredicateLogic.pdf
Retrieved: 24 Oct, 2008.

Roper, M. 1999. Software Testing - Searching for the Missing Link. Information and Software Technology, 41(14):991-994.

Roper, M. & Wood, M. & Miller, J. 1997. An Empirical Evaluation of Defect Detection Techniques. Information and Software Technology 39(11):763-775.

Rosenblum, D.S. 1995. A Practical Approach to Programming with Assertions. IEEE Transactions on Software Engineering, 21(1): 19-31.

Rothermel, G. & Elbaum, S. & Malishevsky, A.G. & Kallakuri, P. & Qiu, X. 2004. On Test Suite Composition and Cost-Effective Regression Testing. ACM Transactions on Software Engineering and Methodology, 13(3): 277-331.

Rowe, N.C. 1988. Absolute Bounds on Set Intersection and Union Sizes from Distribution Information. IEEE Transactions on Software Engineering, 14(7):1033-1048.

Royce, T. & Necaise, R. 2003. A Parallel Algorithm for DNA Alignment. Crossroads. The ACM Student Magazine. Interdisciplinary Computer Science, issue 9.3. Pages 10-15.

RTI. Research Triangle Institute. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3. Health, Social, and Economics Research. RTI Project Number 7007.011. NIST 2002.

Rubey, R.J. 1975. Quantitative Aspects of Software Validation. ACM SIGPLAN Software Engineering Notices, 10(6):246-251.

Rushby, J. 1993. Anomalies in Digital Flight Control Systems. [e-document]. Notes excerpted from: Formal Methods and the Certification of digital Systems, SRI-CSL-93-07. Nov, 1993. SRI International, Menlo Park, CA, USA. Computer Science Laboratory.
www.csl.sri.com/users/rushby/anomalies.html

Sag, I.A. & Wasow, T. 1999. Syntactic Theory: A Formal Introduction. Center for the Study of Language and Information. The University of Malta, Msida, Malta. 20 Jan, 1999.
http://staff.um.edu.mt/mros1/ftp/download/sw99.pdf
Retrieved: 25 Oct, 2008.

Saglietti, F. 1990. Software Diversity Metrics Quantifying Dissimilarity in the Input Partition. IEEE Software Engineering Journal, 5(1): 59-63.

Saleh, K. & Boujarwah, A.A. & Al-Dallal, J. 2001. Anomaly Detection in Concurrent Java Programs Using Dynamic Data Flow Analysis. Information and Software Technology, 43(15):973-981.

Sankaranarayanan, S. & Sipma, H.B. & Manna, Z. 2004. Non-Linear Loop Invariant Generation Using Gröbner Bases. Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy, 14-16 Jan, 2004. Pages 318-329. ISBN-X 1-58113-729-X.

Santone, A. 2003. Heuristic Search + Local Model Checking in Selective mu-Calculus. IEEE Transactions on Software Engineering, 29(6): 510-523.

Santor, G. 2007. [Webpage of Professor Santor]. Fanshawe College, London, Ontario, Canada.
http://infotech.fanshawec.ca/gsantor/Computing/FamousBugs.htm
Retrieved: 24 Apr, 2007.

Santos, R.M. & Santos, J. & Orozco, J.D. 2005. A Least Upper Bound on the Fault Tolerance of Real-Time Systems. Journal of Systems and Software, 78(1):47-55.

Sanyal, S. & Aida, K. & Gaitanos, K. & Wowk, G. & Lahiri, S. 1992. Defect Tracking and Reliability Modeling for a New Product Release. Proceedings of the 1992 Conference on the Centre for Advanced Studies on Collaborative Research 1(1). IBM Canada Limited Laboratory, North York, Ontario, Canada.

Sarkar, D. & De Sarkar, S.C. 1989. A Set of Inference Rules for Quantified Formula Handling and Array Handling in Verification of Programs Over Integers. IEEE Transactions on Software Engineering, 15(11): 1368-1381.

Sarkar, S. & Rama, G.M. & Kak, A.C. 2007. API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization. IEEE Transactions on Software Engineering, 33(1): 14-32.

Schewe, K.-D. & Thalheim, B. 1999. Towards a Theory of Consistency Enforcement. Acta Informatica 36:97-141.

Schmidt, D.A. 1998. Data Flow Analysis is Model Checking of Abstract Interpretations. Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, 19-21 Jan, 1998. ACM Press, New York, NY, USA. Pages 38-48. ISBN 0-89791-979-3.

Schmidt, D.A. 2007. A Calculus of Logical Relations for Over- and Underapproximating Static Analyses. Science of Computer Programming, 64(1):29-53.

Schneidewind, N.F. 2000. Software Quality Control and Prediction Model for Maintenance. Annals of Software Engineering, 9(1-4):79-101. J.C. Baltzer AG, Science Publishers, Red Bank, NJ, USA. ISSN 1022-7091.

Schneidewind, N.F. & Hoffmann, H.-M. 1979. An Experiment in Software Error Data Collection and Analysis. IEEE Transactions on Software Engineering. SE-5(3):276-286.

Schultz, R.D. & Cardenas, A.F. 1987. An Approach and Mechanism for Auditable and Testable Advanced Transaction Processing Systems. IEEE Transactions on Software Engineering 13(6): 666-676.

Schumann, J. 1999. Automated Theorem Proving in High-Quality Software Design.
Applied Logic Series, Vol. 19. Intellectics and Computational Logic (To Wolfgang Bibel on the Occasion of his 60th Birthday),: 295-312. ISBN 9-7923-6261-6.
http://ti.arc.nasa.gov/m/pub/archive/1999-0153.pdf
Retrieved: 14 Oct, 2008.

SDL Software Development Laboratories. 2006. ClockMon. A Free Windows Real Time Clock Monitoring and Synchronizing Utility. Version 2.3.0.291. 19 Oct, 2006.
http://www.softdevlabs.com/ClockMon/ClockMon.html
Retrieved: 10 Oct, 2006.

Seacord, R. 2007. PRE31-C. Never Invoke an Unsafe Macro with Arguments Containing Assignment, Increment, Decrement, Volatile Access, or Function Call. CERT
27 May, 2007. Modified by Swoboda, D., Sep 2008.
https://www.securecoding.cert.org/confluence/display/seccode/PRE31-C.+Never+invoke+an+unsafe+macro+with+arguments+containing+assignment,+increment,+decrement,+volatile+access,+or+function+call
Retrieved: 30 Sep, 2008.

Seebach, P. 2006. Everything You Ever Wanted to Know about C Types, Part 4: Portability and Pitfalls.
http://www.ibm.com/developerworks/power/library/pa-ctypes4/index.html
Retrieved: 29 Sep, 2008.

Sekerinski, E. 2003. Exploring Tabular Verification and Refinement. Formal Aspects in Computing, 15(2-3):215-236.

Selby, R.W. 1990. Empirically Based Analysis of Failures in Software Systems. IEEE Transactions on Reliability, 39(4):444-454.

Selby, R.W. & Basili, V.R. 1991. Analyzing Error-Prone System Structure. IEEE Transactions on Software Engineering, 17(2):141-152.

Shahmehri, N. & Kamkar, M, & Fritzson, P. 1995. Usability Criteria for Automated Debugging Systems. Journal of Systems and Software, 31(1): 55-70.

Shapiro, E.Y. 1983. Algorithmic Program Debugging. The MIT Press Classics Series, The MIT Press, Cambridge, MA, USA. 248 pages. ISBN 0-262-69307-0.

She, J. & Pecht, M.G. 1992. Reliability of a k-out-of-n Warm-Standby System. IEEE Transactions on Reliability, 41(1):72-75.

Sheffield. 2001. Northern General Hospital NHS Trust. Report of the Inquiry Committee into the Computer Software Error in Downs Syndrome Screening. Sheffield. Sep, 2001.
http://www.sheffield.nhs.uk/nhssheffield/resources/downs-report.pdf
Retrieved: 24 Mar, 2007.

Sheil, B.A. 1981. The Psychological Study of Programming. ACM Computing Surveys, 13(1):101-120.

Shepherd, D. 1992. Using HOL to Produce Custom Verification Tools. International Workshop on HOL Theorem Proving System and its Applications, 28-30 Aug, 1991. IEEE. Pages 162-169.

Shepherdson, J.C. & Sturgis, H.E. 1963. Computability of Recursive Functions. Journal of the ACM 10(2):217-255.

Sheth, A. & Hartung, C. & Han, R. 2005. A Decentralized Fault Diagnosis System for Wireless Sensor Networks. IEEE International Conference on, Mobile Adhoc and Sensor Systems Conference, 7-10 Nov, 2005. 3 pages. DOI 10.1109/MAHSS.2005.1542799.

Shima, K. & Takada, S. & Matsumoto, K. & Torii, K. 1997. A Study on the Failure Intensity of Different Software Faults. Proc 19th international conference on Software Engineering, Boston, MA, USA, 17-23 May, 1997. ACM Press, New York, USA. Pages 86-94. ISBN 0-89791-914-9.

Shimeall, T.J. & Leveson, N.G. 1991. An Empirical Comparison of Software Fault Tolerance and Fault Elimination. IEEE Transactions on Software Engineering, 17(2):173-182.

Shiratori, N. & Zhang, Y.-X. & Takahashi, K. & Noguchi, S. 1991. A User Friendly Software Environment for Protocol Synthesis. IEEE Transactions on Computers, 40(4): 477-486.

Shmueli, G. 2003. Computing Consecutive-Type Reliabilities Non-Recursively. IEEE Transactions on Reliability, 52(3): 367-372.

Shull, F. & Cruzes, D & Basili, V & Mendonça, M. 2005. Simulating Families of Studies to Build Confidence in Defect Hypotheses. Information and Software Technology, 47(15):1019-1032.

Siegel, S.F. & Avrunin, G.S. 2000. Improving the Precision of INCA by Preventing Spurious Cycles. ACM SIGSOFT Software Engineering Notes, 25(5):191-200. ISBN 1-58113-266-2.

Sinha, A. & Smidts, C. 2006.: HotTest: a Model-Based Test Design Technique for Enhanced Testing of Domain Specific Applications. ACM Transactions on Software Engineering and Methodology 15(3):242-278.

Sinha, P. & Hanumantharya, A. 2005. A Novel Approach for Component-Based Fault-Tolerant Software Development. Information and Software Technology 47(6):365-382.

Sinha, S. & Harrold, M.J. 2000. Analysis and Testing of Programs with Exception Handling Constructs. IEEE Transactions on Software Engineering, 26(9):849-871.

Sistla, A.P. 2004. Employing Symmetry Reductions in Model Checking. Computer Languages, Systems & Structures, 30(3-4):99-137.

Smidts, C. & Huang, X. & Widmaier, J.C. 2002. Producing Reliable Software: An Experiment. Journal of Systems and Software, 61(3):213-224.

Smidts, C. & Stoddard, R.W. & Stutzke, M. 1996. Software Reliability Models: An Approach to Early Reliability Prediction. Proceedings of the 7th International Symposium on Software Reliability Engineering, 30 Oct – 2 Nov, 1996. IEEE Computer Society, Washington DC, USA. From page 132. ISBN 0-8186-7707-4.

Smith, B. & Feather, M.S. & Muscettola, N. 2000. Challenges and Methods in Testing a Remote Agent Planner. Chien, S. & Kambhampati, S. & Knoblock, C.A. (eds.). Proceedings of the fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, 14-17 Apr, 2000. Pages 254-263. ISBN 1-57735-111-8.

Snelting, G. & Robschink, T. & Krinke, J. 2006. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. ACM Transactions on Software Engineering and Methodology, 15(4): 410-457.

Sogame, H. (prepared for the WWW). 1999. China Airlines Boeing 747-SP Accident Report. NTSB/AAR-86/03. NTSB, Washington DC, USA.
www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/ChinaAir/AAR8603.html
Retrieved: 6 May, 2008.

Sogame, H. & Ladkin, P. (prepared for the WWW). 1996. Aircraft Accident Investigation Report 96-5. China Airlines. Airbus Industrie A300B4-622R. B1816, Nagoya Airport, April 26, 1994. Aircraft Accident Investigation Commission. Ministry of Transport, Japan.
Prepared for Internet 1996.
www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Nagoya/nagoyarep/nagoya-top.html
Retrieved: 6 May, 2008.

Sohn, SD. & Seong, PH. 2004. Quantitative Evaluation of Safety Critical Software Testability Based on Fault Tree Analysis and Entropy. Journal of Systems and Software, 73(2):351-360.

Sommerville, I. & Ransom, J. 2005. An Empirical Study of Industrial Requirements Engineering Process Assessment and Improvement. ACM Transactions on Software Engineering and Methodology, 14(1):85-117.

Qinbao Song & Shepperd, M. & Cartwright, M. & Mair, C. 2006. Software Defect Association Mining and Defect Correction Effort Prediction. IEEE Transactions on Software Engineering, 32(02):69-82.

Spohrer, J.C. & Soloway, E. 1986a. Alternatives to Construct-Based Program Misconceptions. Proceedings of ACM SIGCHI conference on human factors in computing systems, Boston, MA, USA, 13-17 Apr, 1986. ACM Press, New York, NY, USA. Pages 183-191. ISBN 0-89791-180-6.

Spohrer, J.C. & Soloway, E. 1986b. Novice Mistakes: Are the Folk Wisdoms Correct? Communication of the ACM, 29(7):624-632.

Sreemani, T. & Atlee, J.M.. 1996. Feasibility of Model Checking Software Requirements: A Case Study. Proceedings of the 11th Conference on Computer Assurance, Gaithersburg, MD, USA, 17-21 Jun, 1996. Pages 77-88.

Staknis, M.E. 1993. A Formal Investigation of Checking the Input to Critical Systems. IEEE Transactions on Reliability 42(4):588-595.

Stallinger, F. & Grünbacher, P. 2001. System Dynamics Modelling and Simulation of Collaborative Requirements Engineering. Journal of Systems and Software, 59(3):311-321.

Stamelos, I. 2003. Detecting associative shift faults in predicate testing. Journal of Systems and Software, 66(1):57-63.

Stavely, A.M. 1995. Verifying Definite Iteration over Data Structures. IEEE Transactions on Software Engineering, 21(6):506-514.

Stonebraker, M. & Hanson, E.N. & Potamianos, S. 1988. The POSTGRES Rule Manager. IEEE Transactions on Software Engineering, 14(7): 897-907.

Strigini, L. 2004. "Fault Tolerance Against Design Faults", in Diab, H.B. & Zomaya, A.Y. (eds.) "Dependable Computing Systems: Paradigms, Performance Issues, and Applications", J. Wiley & Sons, 2004. 638 pages. ISBN 9780471674221.

Strobl, W. 2000. [Zope] Risks. Was ZopeLDAP 1.0b4 Breaks the Root Directory Security Tab. [e-message on discussion board]. 16 Nov, 2000. Users of Z Object Publishing Environment.
http://mail.zope.org/pipermail/zope/2000-November/120719.html
Retrieved: 24 Mar, 2007.

Strom, R.E. & Yellin, D.M. 1993. Extending Typestate Checking Using Conditional Liveness Analysis. IEEE Transactions on Software Engineering, 19(5):478-485.

Stuart, D.A. & Brockmeyer, M. & Mok, A.K. & Jahanian, F. 2001. Simulation-Verification: Biting at the State Explosion Problem. IEEE Transactions on Software Engineering, 27(7):599-617.

Stumptner, M. & Wotawa, F. 1998. A Survey of Intelligent Debugging. AI Communications, 11(1): 35-51.

Suárez, A. & Lutsko, J.F. 1999. Globally Optimal Fuzzy Decision Trees for Classification and Regression. IEEE Transactions on Pattern Analysis and Machine Intelligence, 21(12):1297-1311.

Subramanyam, R. & Krishnan, M.S. 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering, 29(4):297-310.

Subramanian, S. & Vishnuvajjala, R.V. & Mojdehbakhsh, R. & Tsai, W.T. & Elliott, L. 1995. A framework for designing safe software systems. Proceedings of the Nineteenth Annual International Computer Software and Applications Conference, COMPSAC 95, Dallas, TX, USA, 9-11 Aug, 1995. IEEE Computer Society, Washington DC, USA. Pages 409-414. ISBN 0-8186-7119-X.

Succi, G. & Pedrycz, W. & Stefanovic, M. & Miller, J. 2003. Practical Assessment of the Models for Identification of Defect-Prone Classes in Object-Oriented Commercial Systems Using Design Metrics. Journal of Systems and Software, 65(1):1-12.

Sullivan, K. & Yang, J. & Coppit, D. & Khurshid, S. & Jackson, D. 2004. Software Assurance by Bounded Exhaustive Testing. ACM SIGSOFT Software Engineering Notes, 29(4):133-142. Session II: Testing. ISSN 0163-5948.

Sullivan, M. & Chillarege, R. 1991. Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems. Twenty-First International Symposium on Fault-Tolerant Computing, Digest of Papers, Montreal, Quebec Canada, 25-27 Jun, 1991. Pages 2-9. ISBN 0-8186-2150-8.

Sullivan, M. & Chillarege, R. 1992. A Comparison of Software Defects in Database Management Systems and Operating Systems. Proceedings on the Twenty-Second International Symposium of Fault-Tolerant Computing, Boston, MA, USA, 8-10 Jul, 1992. IEEE Computer Society, Washington DC, USA. Pages 475-484. ISBN 0-8186-2875-8.

Sutcliffe, A. & Maiden, N. 1998. The Domain Theory for Requirements Engineering. IEEE Transactions on Software Engineering, 24(3):174-196.

SWEBOK. 2007. The Guide to Software Engineering Body of Knowledge. Software Engineering Research Laboratory. 12 Jan, 2007.
http://www.swebok.org/
Retrieved: 31 Oct, 2008.

Swobodova, L. 1981. Performance Monitoring in Computer Systems: A Structured Approach. ACM SIGOPS Operating Systems Review, 15(3): 39-50.

Sy, N.T. & Deville, Y. 2001. Automatic Test Data Generation for Programs with Integer and Float Variables. Proceedings of the 16th Annual International Conference on Automated Software Engineering, 26-29 Nov, 2001. IEEE. Pages 13-21.

Taghdiri, M. 2004. Inferring Specifications to Detect Errors in Code. Proceedings of the 19th International Conference on Automated Software Engineering (ASE), Linz, Austria, 20-24 Sep, 2004. Pages 144-153. ISBN ISSN 1068-3062, 0-7695-2131-2.

Tai, A.T. & Tso, K.S. & Alkalai, L. & Chau, S.N. & Sanders, W.H. 2001. Synergistic Coordination between Software and Hardware Fault Tolerance Techniques. International Conference on Dependable Systems and Networks, 1-4 Jul, 2001. IEEE. Pages 369-378. DOI 10.1109/DSN.2001.941421.

Tai, K.C. 1993. Predicate-Based Test Generation for Computer Programs. Proceedings of the 15th International Conference on Software Engineering, 17-21 May, 1993. IEEE. Pages 267-276. DOI 10.1109/ICSE.1993.346037.

Kuo-Chung Tai. 1996. Theory of Fault-Based Predicate Testing for Computer Programs. IEEE Transactions on Software Engineering, 22(8):552-562.

Tai, K.-C. & Carver, R.H. & Obaid, E.E. 1991. Debugging Concurrent ADA Programs by Deterministic Execution. IEEE Transactions on Software Engineering, 17(1): 45-63.

Tai, K.-C. & Paradkar, A. & Su, H.-K. & Vouk, M.A. 1993. Testing and Debugging: Fault-Based Test Generation for Cause-Effect Graphs. Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering, Toronto, Ontario, Canada, 24-28 Oct, 1993. IBM Press(1):495-504.

Taibi, T. & Taibi, F. 2006. Formal Specification of Design Patterns and Their Instances. IEEE International Conference on Computer Systems and Applications, 8 Mar, 2006. Pages 33-36.

Takagi, N. & Nakashima, K. & Mukaidono, M. 1996. A Necessary and Sufficient Condition for Lukasiewicz Logic Functions. Proceedings of the 26th International Symposium on Multiple-Valued Logic, 29-31 May, 1996. Pages 37-42. DOI 10.1109/ISMVL.1996.508333.

Takahashi, K. & Oka, A. & Yamamoto, S. & Isoda, S. 1995. A Comparative Study of Structural and Text-Oriented Analysis and Design Methodologies. Journal of Systems and Software, 28(1):69-75.

Tan, Z. 2007. Estimation of Exponential Component Reliability from Uncertain Life Data in Series and Parallel Systems. Reliability Engineering and System Safety 92(2): 223-230.

Taylor, R.N. & Levine, D.L. & Kelly, C.D. 1992. Structural Testing on Concurrent Programs. IEEE Transactions on Software Engineering, 18(3): 206-215.

Tekinerdogan, B. & Sozer, H. & Aksit, M. 2008. Software Architecture Reliability Analysis using Failure Scenarios. Journal of Systems and Software, 81(4):558-575.

Ter Hofstede, A.H.M. & Proper, H.A. 1998. How to Formalize it? Formalization Principles for Information System Development Methods. Information and Software Technology, 40(10):519-540.

Tervonen, I. & Kerola, P. 1998. Towards Deeper Co-Understanding of Software Quality. Information and Software Technology, 39(14-15):995-1003.

Tewksbury, S. 2002. Information Warfare: Possible Research Themes. Jan 7, 2002.
http://stewks.ece.stevens-tech.edu/Personal/Reports/Security/infowarfare.pdf
Retrieved: 11 Oct, 2008.

Thelin, T. & Petersson, H. & Runeson, P. & Wohlin, C. 2004. Applying Sampling to Improve Software Inspections. Journal of Systems and Software, 73(2): 257-269.

Thévenod-Fosse, P. & Waeselynck, H. 1993. STATEMATE Applied to Statistical Software Testing. ACM SIGSOFT Software Engineering Notes, 18(3): 99-109.

Thomas, W.M. & Delis, A. & Basili, V.R. 1997. An Analysis of Errors in a Reuse-Oriented Development Environment. Journal of Systems and Software, 38(3):211-224.

Tian, J. 1996. An Integrated Approach to Test Tracking and Analysis. Journal of Systems and Software, 35(2):127-140.

Tian, J. 1999. Measurement and continuous improvement of software reliability throughout software life-cycle. Journal of Systems and Software, 47(2-3):189-195.

Tian, J. 2002. Better Reliability Assessment and Prediction through Data Clustering. IEEE Transactions on Software Engineering, 28(10):997-1007.

Tian, J. & Nguyen, A. & Allen, C. & Appan, R. 2001: Experience with Identifying and Characterizing Problem-Prone Modules in Telecommunication Software Systems. Journal of Systems and Software, 57(3):207-215.

Tian, J. & Troster, J. 1998. A Comparison of Measurement and Defect Characteristics of New and Legacy Software Systems. Journal of Systems and Software, 44(2):135-146.

Tian, J. & Troster, J. & Palma, J. 1997. Tool Support for Software Measurement, Analysis and Improvement. Journal of Systems and Software, 39(2):165-178.

Tip, F. & Dinesh, T.B. 2001. A Slicing-Based Approach for Locating Type Errors. ACM Transactions on Software Engineering and Methodology, 10(1):5-55.

Tomek, L.A. & Muppala, J.K. & Trivedi, K.S. 1993. Modeling Correlation in Software Recovery Blocks. IEEE Transactions on Software Engineering, 19(11):1071-1086.

Tonella, P. 2003. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. IEEE Transactions on Software Engineering, 29(6):495-509.

Torres-Pomales, W. 2000. Software Fault Tolerance: A Tutorial. NASA Langley Research Center, Hampton, VA, USA. NASA/TM-2000-210616.

Traore, I. & Aredo, D.B. 2004. Enhancing Structured Review with Model-Based Verification. IEEE Transcations on Software Engineering, 30(11): 736-753.

Trivedi, K.S. 2002. SREPT: A Tool for Software Reliability Estimation and Prediction. Proceedings of the International Conference on Dependable Systems and Networks, 23-26 Jun, 2002. IEEE. Page 546. DOI 10.1109/DSN.2002.1028977.

Trivedi, K.S. & Bobbio, A. & Ciardo, G & German, R. & Puliafito, A. & Telek, M. 1995. Non-Markovian Petri Nets. ACM SIGMETRICS Performance Evaluation Review 23(1): 263-264.

Tropp, J.A. & Gilbert, A.C. & Strauss, M.J. 2006. Algorithms for Simultaneous Sparse Approximation. Part 1: Greedy Pursuit. Signal Processing 86(3): 572-588.

Troscinski, M. 2003. ANSYS/Multiphysics FSI with Applications. 20 Mar, 2003.
www.fe-net.org/downloads/FENet_Meetings/Barcelona_Spain_Feb_2003/FENET_Barcelona_Feb2003_MPA_Ellis.pdf
Retrieved: 9 Oct, 2008.

Tucker, J.V. & Zucker, J.I. 2002. Abstract Computability and Algebraic Specification. ACM Transactions on Computational Logic, 3(2): 279-333.

Uchida, S. & Monden, A & Iida, H. & Matsumoto, K-i & Kudo, H. 2002. A Multiple-View Analysis Method for Debugging Processes. Proceedings of the 2002 International Symposium on Empirical Software Engineering, 3-4 Oct, 2002. IEEE. Pages 139-147. DOI 10.1109/ISESE.2002.1166933.

Ural, H. & Yang, B. 1993. Modeling Software for Accurate Data Flow Representation. Proceedings of the 15th International Conference on Software Engineering, Baltimore, MD, USA, 17-21 May, 1993. IEEE. Pages 277-286. ISBN: 0-89791-588-7.

van Breugel, F. & Mislove, M. & Ouaknine, J. & Worrell, J. 2005. Domain Theory, Testing and Simulation for Labelled Markov Processes. Theoretical Computer Science 333(1-2):171-197.

van den Brand, M.G.J. & Klint, P. & Vinju, J.V. 2003. Term Rewriting with Traversal Functions. ACM Transactions on Software Engineering and Methodology, 12(2): 152-190.

van der Meulen, M.J.P. & Bishop, P.G. & Revilla, M. 2004. An Exploration of Software Faults and Failure Behaviour in a Large Population of Programs. 15th IEEE International Symposium on Software Reliability Engineering, 2-5 Nov, 2004. ISSRE 2004. IEEE Computer Society, Washington DC, USA. Pages: 101-112. ISBN ISSN 1071-9458, 0-7695-2215-7.

van der Schoot, H. & Ural, H. 1998. On Improving Reachability Analysis for Verifying Process Properties of Networks of CFSMs. Proceedings of the 18th International Conference on Distributed Computing Systems, Amsterdam, Netherlands, 26-29 May, 1998. Pages 130-137. ISBN 0-8186-8292-2.

van Lamsweerde, A. 2004. Elaborating Security Requirements by Construction of Intentional Anti-Models. Proceedings of the 26th International Conference on Software Engineering, 23-28 May, 2004. IEEE. Pages 148-157.

van Lamsweerde, A. & Darimot, R. & Leiter, E. 1998. Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Transactions on Software Engineering, 24(11):908-926.

van Lamsweerde, A. & Leiter, E. 2000. Handling Obstacles in Goal-Oriented Requirements Engineering. IEEE Transactions on Software Engineering, 26(10):978-1005.

Van Rompaey, B. & Bois, B.D. & Demeyer, S. & Rieger, M. 2007. On the Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. IEEE Transactions on Software Engineering, 33(12): 800-817.

Verbaeten, S. & Sagonas, K. & De Schreye, D. 2001. Termination Proofs for Logic Programs with Tabling. ACM Transactions on Computational Logic 2(1):57-92.

Vermeulen, S.T.J.A. & Rijanto, H. & van der Duyn Schouten, F.A. 1998. Modelling the Influence of Preventive Maintenance on Protection System Reliability Performance. IEEE Transactions on Power Delivery, 13(4): 1027-1032.

Vickers, S. & Hill, G. 2001. Presheaves as Configured Specifications. Formal Aspects of Computing, 13(1):32-49.

Virkkunen, V.-E. J. 1980. A Unified Approach to Floating-point Rounding with Applications to Multiple-precision Summation. Dissertation. The University of Helsinki, the Department of Computer Science. ISBN 951-45-1948-5.

Visser, P.R.S. & Jones, D.M. & Bench-Capon, T.J.M. & Shave, M.J.R. 1997. An Analysis of Ontology Mismatches; Heterogeneity versus Interoperability. AAA1 1997 Spring Symposium on Ontological Engineering, Stanford, USA. Kraft.
Also appeared as: Farquhar, A. & Grunninger, M, (eds.). AAA1 Technical Report, SS-97-06.

Voas, J.M. 1992. PIE: A Dynamic Failure-Based Technique. IEEE Transactions on Software Engineering, 18(8):717-727.

Voas, J. & Charron, F. & Miller, K. 1996. Investigating Rare-Event Failure Tolerance: Reductions in Future Uncertainty. Proceedings of IEEE High-Assurance System Engineering Workshop HASE 96, In Conjunction with the 15th Symposium on Reliable Distributed Systems, Niagara-on-the-Lake, Canada, 22 Oct, 1996. IEEE Computer Society, Washington DC, USA. From page 78. ISBN 0-8186-7629-9.

Voas, J.M. & Miller, K.W. 1995a. Examining Fault-Tolerance Using Unlikely Inputs - Turning the Test Distribution Up-Side Down. Proceedings of the tenth Annual Conference on System Integrity, Softawre Safety, and Process Security. Computer Assurance, Gaithersburg, MD, USA, 25-29 Jun, 1995. IEEE Computer Society, Washington DC, USA. Pages 3-11.

Voas, J.M. & Miller, K.W. 1995b. Software Testability: The New Verification. IEEE Software 12(3):17-28.

Vokáč, M. 2004. Defect Frequency and Design Patterns: An Empirical Study of Industrial Code. IEEE Transactions on Software Engineering, 30(12):904-917.

von Mohrenschildt, M. 2000. Algebraic Composition of Function Tables. Formal Aspects of Computing 12(1):41-51.

Vouk, M.A. & Padakar, A.M. & McAllister, D.F. 1990. Modeling Execution Time of Muli-Stage N-Version Fault-Tolerant Software. Proveedings of the Fourteenth Annual International Computer Software and Applications Conrefenre, 31 Oct – 2 Nov, 1990. IEEE. Pages 505-511. DOI 10.1109/CMPSAC.1990.139422.

Vouk, M.A. & Tai, K.C. 1993. Some Issues in Multi-Phase Software Reliability Modeling. Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering, Toronto, Ontario, Canada, 24-28 Oct, 1993. SESSION: Testing and Debugging. Volume 1. Pages 513-523. IBM Press.

Wahbe, R. & Lucco, S. & Anderson, T.E. & Graham, S.L. 1993. Efficient Software-Based Fault Isolation. ACM SIGOPS Operation Systems Review, 27(5):203-216. ACM Press, New York, NY, USA. ISBN 0-89791-632-8.

Walicki, M. & Meldal, S. 1997. Algebraic Approaches to Nondeterminism: An Overview. ACM Computing Surveys, 29(1): 30-81.

Walkinshaw, N. & Bogdanov, K. & Holcombe, M. 2006. Identifying State Transitions and their Functions in Source Code. Proceedings of Testing: Academic and Industrial Conference – Practice and Research Techniques. IEEE. Pages 49-58. DOI 10.1109/TAIC-PART.2006.12.

Wallace, D.R. & Ippolito, L.M. & Hecht, H. 1997. Error, Fault, and Failure Data Collection and Analysis. National Institute of Standards and Technology, Gaithersburg, MD, USA.
http://hissa.nist.gov/eff/qweff.html
Retrieved: 27 Oct, 2008.

Wallace, D.R. & Kuhn, D.R. 2001. Failure Modes in Medical Devices Software: an Analysis of 15 Years of Recall Data. International Journal of Reliability, Quality and Safety Engineering, 8(4):351-371.

Walter, C.J. & Lincoln, P. & Suri, N. 1997. Formally Verifying On-Line Diagnosis. IEEE Transactions on Software Engineering, 23(11):684-721.

Wand, Y. & Weber, R. 1990. An Ontological Model of an Information System. IEEE Transactions on Software Engineering, 16(11):1282-1292.

Farn Wang. 2005. Symbolic Parametric Safety Analysis of Linear Hybrid Systems with BDD-Like Data-Structures. IEEE Transactions on Software Engineering, 31(1):38-51.

Wang, S. & Shin, K.G. 2006. Task Construction for Model-Based Design of Embedded Control Software. IEEE Transactions on Software Engineering, 32(4):254-264.

Wang, Y.-M. & Huang, Y. & Fuchs, W.K. 1993. Progressive Retry for Software Error Recovery in Distributed Systems. Digest of Papers, Twenty-Third International Symposium on Fault-Tolerant Computing, FTCS-23, Toulouse, France, 22-24 Jun, 1993. Pages 138-144. ISBN 0-8186-3680-7.

Wasserman, H. & Blum, M. 1997. Software Reliability via Run-Time Result-Checking. Journal of the ACM, 44(6):826-849.

Watson, A.H. & McCable, T.J. 1996. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500-235. U.S. Department of Commerce. Technology Administration. National Institute of Standards and Technology. Computer Laboratory. Gaithersburg, MD, USA.

Wegner, P. & Goldin. D. 2003. Computation Beyond Turing Machines. Communications of the ACM, 46(4):100-102.

Weinberg, G.M. & Freedman, D.P. 1984. Reviews, Walkthroughs, and Inspections. IEEE Transactions on Software Engineering, 10(1):68-72.

Weiser, M. 1984. Program Slicing. IEEE Transactions on Software Engineering, SE-10(4):352-357.

Weiser, M.D. & Gannon, J.D. & McMullin, P.R. 1985. Comparison of Structural Test Coverage Metrics. IEEE Software, 2(2) : 80-85.

Weiss, S.N. & Weyuker, E.J. 1988. An Extended Domain-Based Model of Software Reliability. IEEE Transactions on Software Engineering, 14(10):1512-1524.

Werner, L.L. 1986. A Study of 'Hard to Find' Data Processing Errors. ACM 14th Annual Conference on Computer Science, Cincinnati, OH, USA. Page 410. ISBN 0-89791-177-6.

Westerfield, M. 1992. ORCA/C 2.0^{TM} A C Compiler and Development System for the Apple IIGS. Byte Works Inc, Albuquerque, NM, USA.
http://www.byteworks.org/resume/samples/c.pdf
Retrieved: 22 Oct, 2008.

Westland, J.C. 2002. The Cost of Errors in Software Development: Evidence from Industry. Journal of systems and software, 62(1):1-9.

Weyuker, E.J. 1988. The Evaluation of Program-Based Software Test Data Adequacy Criteria. Communications of the ACM, 31(6) : 668-675.

Weyuker, E.J. & Jeng, B. 1991. Analyzing Partition Testing Strategies. IEEE Transactions on Software Engineering, 17(7):703-711.

Weyuker, E.J. & Ostrand, T.J. 1980. Theories of Program Testing and the Application of Revealing Subdomains. IEEE Transactions on Software Engineering, SE-6(3):236-246.

Weyuker, E.J. & Vokolos, F.I. 2000. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. IEEE Transactions on Software Engineering, 26(12):1147-1156.

Whang, K.-Y. & Malhotra, A. & Sockut, G.H. & Bruns, L. & Choi, K.-S. 1992. Two-Dimensional Specification of Universal Quantification in a Graphical Database Query Language. IEEE Transactions on Software Engineering, 18(3): 216-224.

White, L.J. & Cohen, E.I. 1980. A Domain Strategy for Computer Program Testing. IEEE Transactions on Software Engineering, SE-6(3):247-257.

White, L.J. & Sahay, P.N. 1985. Experiments Determining Best Paths for Testing Computer Program Predicates. Proceedings of the 8th International Conference on Software Engineering, London, UK, 28-30 Aug, 1985. IEEE Computer Society, Los Alamitos, CA, USA. Pages 238-243. ISBN 0-8186-0620-7.

White, L.J. & Wiszniewski, B. 1988. Complexity of Testing Iterated Borders for Structured Programs. IEEE Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, Banff, Alta., Canada, 19-21 Jul, 1988. Pages 231-237. ISBN 0-8186-0868-4.

Whittaker, J.A. 2001. Software's Invisible Users. IEEE Software 18(3):84-88.

Wijnstra, J.G. 2003. From Problem to Solution with Quality Attributes and Design Aspects. Journal of Systems and Software, 66(3):199-211.

Williams, C.C. & Hollingsworth, J.K. 2005. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. IEEE Transactions on Software Engineering, 31(6):466-480.

Williamson, K. & Healy, M. 1999. Industrial Applications of Software Synthesis via Category Theory. Proceedings of the 14th International Conference on Automated Software Engineering, 12-15 Oct, 1999. IEEE. Pages 35-43. DOI 10.1109/ASE.1999.802090.

Winskel, Y. & Nielsen, M. 1995. Models for Concurrency. In: Abramsky, S. & Gabbay, D.M. & Maibaum, T.S.E. (eds.): Handbook of logic in computer science. Vol. 4. Oxford: Clarendon Press. 672 pages. Pages 1-148. ISBN 0198537808.

Wohlin, C. & Körner, U. 1990. Software Faults: Spreading, Detection and Costs. IEEE Software Engineering Journal 5(1):33-42.

Wondergem, B.C.M. & van Bommel, P. & van der Weide, Th.P. 2001. Combining Boolean Logic and Linguistic Structure. Information and Software Technology, 43(1):53-59.

Wong, W.E. & Mathur, A.P. 1995. Reducing the Cost of Mutation Testing: An Empirical Study. Journal of Systems and Software, 31(3):185-196.

Wong, W.E. & Sugeta, T. & Qi, Y. & Maldonado, J. C. 2005. Smart Debugging Software Acrhitectural Design in SDL. Journal of Systems and Software, 76(1):15-28.

Woodbury, M.H. & Shin, K.G. 1990. Measurement and Analysis of Workload Effects on Fault Latency in Real-Time Systems. IEEE Transactions on Software Engineering, 16(2):212-216.

Woodward, M.R. & Halewood, K. 1988. From Weak to Strong, Dead or Alive? An Analysis of some Mutation Testing Issues. Proceedings of the 2nd Workshop on Software Testing, Verification and Analysis, Banff, Alta., Canada, 19-21 Jul, 1988. Pages 152-158. ISBN 0-8186-0868-4.

Wooff, D.A. & Coldstein, M. & Cohen, F.P.A. 2002. Bayesian Graphical Models for Software Testing. IEEE Transactions on Software Engineering, 28(5):510-525.

Wu, J. & Fernandez, E.B. & Zhang, M. 1996. Design and Modeling of Hybrid Fault-Tolerant Software With Cost Constraints. Journal of Systems and Software 35(2):141-149.

Xia, F. 1999. Look Before You Leap: On Some Fundamental Issues in Software Engineering Research. Information and Software Technology 41(10):661-672.

Xia, F. 2000. On the Concept of Coupling, its Modeling and Measurement. Journal of Systems and Software, 50(1):75-84.

Xie, M. & Hong, G.Y. & Wohlin, C. 1999. Software Reliability Prediction Incorporating Information from Similar Project. Journal of Systems and Software 49(1):43-48.

Xie, Y. & Engler, D. 2003. Using Redundancies to Find Errors. IEEE Transactions on Software Engineering, 29(10):915-928.

Xu, H. & Dugan, J.B. 2004. Combining Dynamic Fault Trees and Event Trees for Probabilistic Risk Assessment. Annual Symposium on Reliability and Maintainability, 26-29 Jan, 2004. Pages 214-219. DOI 10.1109/RAMS.2004.1285450.

Xu, J. & Randell, B. 1997. Software fault tolerance: t/(n-1)-variant programming. IEEE Transactions on Reliability, 46(1):60-68.

Xu, W. & DuVarney, D.C. & Sekar, R. 2004. An Efficient and Backwards-Compatible Transform to Ensure Memory Safety in C Programs. ACM SIGSOFT Software Engineering Notes, 29(6):117-126. SESSION: Safety and Security. ACM Press, New York, NY, USA. ISSN 0163-5948.

Yang, C.-C. & Chen, J. J.-Y. & Chau, H.L. 1989. Algorithms for Constructing Minimal Deduction Graphs. IEEE Transactions on Software Engineering, 15(6):760-770.

Yang, J. & Twohey, P. & Engler, D. & Musuvathi, M. 2004. Using Model Checking to Find Serious File System Errors. Proceedings of the 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, USA, 6-8 Dec, 2004. USENIX and ACM SIGOPS.

Yi, K. 1998. An Abstract Interpretation for Estimating Uncaught Exceptions in Standard ML Programs. Scientific Computer Programming 31(1):147-173.

Yilmaz, C. & Cohen, M.B. & Porter, A.A. 2006. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. IEEE Transactions on Software Engineering, 32(1):20-34.

Yoo, C.S. & Seong, P.H. 2002. Experimental analysis of specification language diversity impact on NPP software diversity. Journal of Systems and Software, 62(2):111-122.

Young, W.D. 1997. Comparing Verification Systems: Interactive Consistency in ACL2. IEEE Transactions on Software Engineering, 23(4):214-223.

Yunfeng, W. & Bixin, L. & Jun, P. & Ming, Z. Guoliang, Z. 1999. A Formal Software Development Approach Based on COOZ and Refinement Calculus. Proceedings of the 31th Conference on Technology of Object-Oriented Languages and Systems, 22-25 Sep, 1999. IEEE. Pages 261-266. DOI 10.1109/TOOLS.1999.796492.

Yur, J.-s. & Ryder, B.G. & Landi, W.A. 1999. An Incremental Flow- and Context-sensitive Pointer Aliasing Analysis. Proceedings of the 1999 International Conference on Software Engineering, 16-22 May, 1999. IEEE. Pages 442-451. DOI 10.1109/ECSE.1999.841034.

Yur, J.-S. & Ryder, B.G. & Landi, W.A. & Stocks, P. 1997. Incremental Analysis of Side Effects for C Software Systems. Proceedings of the 1997 International Conference on Software Engineering, 17-23 May, 1997. IEEE. Pages 422-432.

Zave, P. & Jackson, M. 1996. Where do Operations Come from? A Multiparadigm Specification Technique. IEEE Transactions on Software Engineering, 22(7):508-528.

Zeil, S.J. 1983. Testing for Perturbations of Program Statements. IEEE Transactions on Software Engineering, SE-9(3):335-346.

Zeil, S.J. 1999. Verification and Validation. CS 451/551. Old Dominion University, Norfolk, VA, USA.
ftp://ftp.cs.odu.edu/pub/zeil/cs451/Lectures/03vandv/vandv.pdf
Retrieved: 25 Oct, 2008.

Zeil, S.J. & Afifi, F.H. & White, L.J. 1992. Detection of Linear Errors via Domain Testing. ACM Transactions on Software Engineering and Methodology, 1(4):422-451.

Zeil, S.J. & White, L.J. 1981. Sufficient Test Sets for Path Analysis Testing Strategies. Proceedings on the 5th International Conference on Software Engineering, San Diego, CA, USA, 9-12 Mar, 1981. IEEE Press, Piscataway, NJ, USA. Pages 184-191. ISBN 0-89791-146-6.

Zelkowitz, M.V. & Rus, I. 2004. Defect Evolution in a Product Line Environment. Journal of Systems and Software, 70(1-2):143-154.

Zeller, A. & Hildebrandt, R. 2002. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering, 27(2): 183-200.

Zhang, Q. & Harris, I.G. 2000. A Domain Coverage Metric for the Validation of Behavioral VHDL Descriptions. Proceedings of the 2000 International Test Conference, 3-5 Oct, 2000. Pages 302-308. DOI 10.1109/TEST.2000.894218.

Zhang, X. & Pham, H. 2000. An Analysis of Factors Affecting Software Reliability. Journal of Systems and Software, 50(1):43-56.

Zhao, M. & Wohlin, C. & Ohlsson, N. & Xie, M. 1998. A Comparison Between Software Design and Code Metrics for the Prediction of Software Fault Content. Information and Software Technology, 40(14):801-809.

Zheng, A.X. & Jordan, M.I. & Liblit, B. & Naik, M. & Aiken, A. 2006. Statistical Debugging: Simultaneous Identification of Multiple Bugs. Proceedings of the 23rd International Conference on Machine Learning ICML '06, Pittsburgh, PA, USA, 25-29 Jun, 2006. Pages 1105-1112.

Zheng, J. & Williams, L. & Nagappan, N. & Snipes, W. & Hudepohl, J.P. & Vouk, M.A. 2006. On the Value of Static Analysis for Fault Detection in Software. IEEE Transactions on Software Engineering, 32(4):240-253.

Zhu, H. 2003. A Note on Test Oracles and Semantics of Algebraic Specifications. Proceedings of the third International Conference on Software Quality, 6-7 Nov, 2003. Pages 91-98. DOI 10.1109/QSIC.2003.1319090.

Zhu, H. & Hall, P.A.V. & May, J.H.R. 1997. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4):366-427.

Zhu, H. & Jin, L. & Diaper, D. & Bai, G. 2002. Software Requirements Validation via Task Analysis. Journal of Systems and Software, 61(2): 145-169.