Design Imperatives for Improved

Architecture-Based Reliability Prediction

of

Software Systems

Assefa Dagne Semegn

B.Sc. in Electrical Engineering (Addis Ababa University, Ethiopia)

M.Phil in Computer Applications (University of Roorkee, India)

A Thesis Submitted for the Degree of

Doctor of Philosophy

in

Software Engineering

Software Quality Research Laboratory (SQRL) and Centre for Telecommunications Value-Chain Research (CTVR), University of Limerick, Ireland

Supervisors: Prof. Eamonn Murphy and Dr. Donal Heffernan

Submitted to the University of Limerick, October 2009
Table of Contents

Abstract ........................................................................................................................................ vii
Declaration .................................................................................................................................. viii
Acknowledgments ................................................................................................................... ix
Dedication ................................................................................................................................... xi
List of Figures .......................................................................................................................... xii
List of Tables ............................................................................................................................. xiii
List of Tables ............................................................................................................................. xiii
1. Introduction ............................................................................................................................ 1
   1.1 Background ..................................................................................................................... 1
   1.2 Objective of the Research ............................................................................................. 6
      1.2.1 Identify Design Imperatives for Improved Reliability Prediction ...................... 7
      1.2.2 Identifying Mathematical Documentation Techniques to Describe the Behavior of Software Systems .................................................................... 7
      1.2.3 Predicting the Reliability of Software Systems based on their Precise Mathematical Description ................................................................. 8
   1.3 Achievement of the Thesis ......................................................................................... 8
   1.4 Novelty of The Work .................................................................................................. 10
   1.5 Organization of the Thesis ....................................................................................... 11
2. Research Question and Approach .................................................................................... 13
   2.1 Introduction ................................................................................................................ 13
   2.2 Research Question ..................................................................................................... 13
   2.3 Research Approach ................................................................................................... 13
3. Related Work ...................................................................................................................... 16
   3.1 Introduction:............................................................................................................... 16
   3.2 The three Tributaries of Design for Reliability Predictability ..................................... 16
   3.3 Main Principles in Software Design and Construction ................................................. 17
   3.4 Design for Correctness by Proof .............................................................................. 18
   3.5 Design for Change ..................................................................................................... 20
   3.6 Design for Fault-Tolerance ....................................................................................... 24
      3.6.1 Software Redundancy ..................................................................................... 24
      3.6.2 Response to Undesired Events and Exception Handling .................................. 30
   3.7 Assessment of Existing Design Approaches ............................................................. 30
   3.8 Precise Specification and Mathematical Documentation of Computer Systems .......... 35
      3.8.1 Assessment of Existing Documentation Approaches ........................................ 36
   3.9 Software Reliability Modeling .................................................................................. 37
      3.9.1 Reliability Growth Models .............................................................................. 37
      3.9.2 Architecture Based Reliability Modeling ......................................................... 39
      3.9.3 Reliability Modeling for Fault-tolerant Software Systems ................................ 47
   3.10 Assessment of Existing Software Reliability Models .............................................. 50
   3.11 Chapter Summary ..................................................................................................... 51
4. Analysis of the Reliability Prediction Problem .................................................................. 53
   4.1 Introduction ............................................................................................................... 53
   4.2 Main Characteristics of Software Failure ................................................................... 53
7.2 Software Architecture .......................................................................................... 108
  7.2.1 Hierarchical Aggregates ............................................................................... 110
  7.2.2 Type Structures .......................................................................................... 115
  7.2.3 Modular Structure ....................................................................................... 116
  7.2.4 Interface Structure ...................................................................................... 117
  7.2.5 Service-Module Dependency Matrix: .......................................................... 123
7.3 Dynamic Structure Functions ............................................................................... 131
  7.3.1 Connector Structure.................................................................................... 132
  7.3.2 Event and Process Structures ...................................................................... 142
  7.3.3 Connection and Event Structures for Fault Tolerance, Component
      Restorability and Redundancy ......................................................................... 148
  7.3.4 Event-Flow Equations ................................................................................ 153
  7.3.5 Event-Flow-Graphs (EFG) .......................................................................... 159
7.4 Chapter Summary ................................................................................................ 166
8. Document Based Structural Reliability Modeling .................................................... 168
  8.1 Introduction ....................................................................................................... 168
  8.2 Modeling Component Reliability ...................................................................... 169
     8.2.1 Component Failure Rate............................................................................ 170
     8.2.2 Justification for the Use of Exponential Failure Rate for Software
         Components: .................................................................................................. 172
  8.3 Modeling Interaction Failures (Stochastic Approach) ........................................ 174
     8.3.1 Missing Deadline ..................................................................................... 176
     8.3.2 Race Conditions ...................................................................................... 197
     8.3.3 Resource Depletion and Congestion Conditions ....................................... 201
     8.3.4 Deadlocks and Livelocks ......................................................................... 210
     8.3.5 Incompatible Input-Outputs ...................................................................... 213
     8.3.6 Incompleteness of Requirements ............................................................ 215
     8.3.7 Mis-ordered Events .................................................................................. 215
  8.4 Service Reliability Structure ............................................................................ 217
     8.4.1 Component Reliability Roles in Services ................................................ 218
  8.5 The Structure Function ..................................................................................... 223
     8.5.1 Interaction Concerns as Virtual Components .......................................... 224
     8.5.2 Series Connection of Modules in a Path.................................................. 229
     8.5.3 Modular Redundancies in Services .......................................................... 229
     8.5.4 Fault Tolerance and Module Irrelevance ............................................... 230
     8.5.5 Multimode Operation of Modules and Service Degradation .................... 231
     8.5.6 Evaluating the Structure Function ............................................................ 235
     8.5.7 Combining Different Paths: ...................................................................... 237
  8.6 Reliability Function Evaluation ........................................................................ 241
     8.6.1 Statistically Independent Failure Causes /Undesired Events .................... 241
     8.6.2 Dependent Failure Causes /Undesired Events .......................................... 241
  8.7 System Failure Rate Estimation ........................................................................ 243
     8.7.1 System Failure Rate Estimation Using the Service-Module Dependency
         Derivative Matrix .......................................................................................... 243
     8.7.2 Including the Effect of Duty Cycle ............................................................ 248
     8.7.3 Testability Analysis Using the Service-Module Derivative Matrix ......... 248
8.8 Chapter Summary ........................................................................................................ 250
9. Design for Reliability Predictability Applied on a Mobile Streaming System as a Case Study ............................................................................................................................................. 251
  9.1 Introduction ........................................................................................................ 251
  9.2 Introductory Description of MSS ....................................................................... 251
  9.3 Simplified View of Interaction with MSS .............................................................. 253
  9.4 System Constraints and Stochastic Properties in MSS ........................................ 254
  9.5 DRP Imperatives as Applied to MSS ................................................................. 256
    9.5.1 Design for Modular Redundancy ................................................................. 256
    9.5.2 Ensure Non-interfering Behavior of Connections ........................................ 257
    9.5.3 Improve Statistical Independence among Redundant Components and Services ................................................................. 258
    9.5.4 Mathematical Description of Components and System ......................... 258
  9.6 Chapter Summary ............................................................................................... 261
10. Document Driven Structural Reliability Assessment of a Mobile Streaming System (Case Study) ......................................................................................................................... 262
   10.1 Introduction ....................................................................................................... 262
   10.2 Analysis of Undesired Events ......................................................................... 262
      10.2.1 Deadline Risk Analysis: .......................................................................... 266
      10.2.2 Race Conditions Risk Analysis: ............................................................... 293
      10.2.3 Resource Depletions and Congestions .................................................... 295
      10.2.4 Fault Tolerance and Multimode Operation ............................................. 299
      10.2.5 Restorability and Effectiveness: ............................................................... 302
      10.2.6 Overall Service Reliability Assessment: ................................................. 312
   10.3 Chapter Summary ............................................................................................ 316
11. Results, Discussion and Conclusion .................................................................... 317
   11.1 Introduction ...................................................................................................... 317
   11.2 Results and Contribution of the Thesis ........................................................... 317
      11.2.1 Design for Modular Redundancy and Fault Tolerance ......................... 317
      11.2.2 Precise Specification and Description of Modules and Structural Relations among System Elements ................................................................. 319
      11.2.3 Structure Based Reliability Prediction .................................................... 321
      11.2.4 DRP Design Imperatives ........................................................................ 322
   11.3 Discussion and Conclusion ............................................................................. 325
   11.4 The Way Forward ........................................................................................... 327
12. Abbreviations ....................................................................................................... 328
13. Glossary ................................................................................................................ 330
14. Bibliography: ........................................................................................................ 337
Appendix A. Mathematical Documents of MSS (Partial) ........................................... 344
   A.1 TFM Specification of Requirements ............................................................... 344
      A.1.1 BP Service in MSS ................................................................................. 344
   A.2 Module Guide (Summary) .............................................................................. 348
   A.3 TFM Specification of Interfaces ..................................................................... 350
      A.3.1 Short Range Communication Link (SRCL) ............................................ 350
      A.3.2 Long Range Communication Link (LRCL) ........................................... 354
      A.3.3 LocalDevice (LD) ................................................................................. 357
Design Imperatives for Improved Architecture-Based Reliability Prediction of Software Systems

Assefa Dagne Semegn

Abstract

Reliability prediction of a software product is complex due to interdependencies and interactions among components and the difficulty of representing this behavior with tractable models. Models developed by making simplifying assumptions about the software structure may be simple to use but their result may be far from what happens in reality. Making assumptions closer to reality that allows complex interactions and interdependencies among components results in models that are too complex to use and/or their results may be too difficult to interpret.

The reliability prediction problem is aggravated by the absence of precise information on the behavior of components and their interactions, information that is relevant for reliability modeling. Usually, the interactions are not known precisely because of subtle undocumented side effects. Without accurate precise information, even mathematically correct models will not yield accurate reliability predications. Deriving the necessary information from program code is not practical if not impossible because the code contains too much implementation detail to be useful in creating a tractable model and because it is difficult to fully analyze.

This author approached the problem from three tracks:

1. Identifying design imperatives that will make the system behavior easier to predict
2. Identifying mathematical documentation techniques to describe the behavior of software systems
3. Adapting structural reliability modeling techniques to predict the reliability of software systems based on their mathematical description

This thesis documents the resulting novel approach of designing, specifying, and describing the behavior of software systems in a way that helps to predict their reliability from the reliability of the components and their interactions. The design approach, which the author names design for reliability predictability (DRP), integrates design for change, precise behavioral documentation and structure based reliability prediction to achieve improved reliability prediction of software systems. The specification and documentation approach builds upon precise behavioral specification of interfaces using the trace function method (TFM) and introduces a number of connection documents or structure functions. These functions capture both the static and dynamic behaviors of component based software systems and are used as a basis for a novel document driven structure based reliability predication model. System reliability assessment is studied in at least three levels: component reliability, which is assumed to be known, interaction reliability, a novel approach in studying software reliability and service reliability, whose estimation is the primary objective of reliability assessment. The approach is applied successfully as a case study in the construction of an industrial product which is described in this thesis.
Declaration

The work described in this thesis is, except where otherwise stated, entirely that of the author and has not been submitted as an exercise for a degree at this or any other University.

__________________        ________________
Eamonn Murphy       Donal Heffernan

Assefa Dagne Semegn
October 2009

The work described in this thesis is, except where otherwise stated, entirely that of the author.

__________________        ___________________
Eamonn Murphy       Donal Heffernan
Acknowledgments

First and foremost, I praise Lord, the Almighty, Who gave me all what I need to bring this work to a success and to whom I owe my very existence.

This work is supported by the Science Foundation of Ireland (SFI) through funding to the Software Quality Research Laboratory (SQRL) and the Center for Telecommunication Value Chain Research (CTVR) joint project, at the University of Limerick, Ireland.

Numerous people around me: family, supervisors, colleagues, staff, and friends, have contributed directly and indirectly to make the work a success, which I would like to acknowledge.

My deepest gratitude goes to my supervisors, Prof. Eamonn Murphy and Dr. Donal Heffernan for their invaluable feedback, advice, encouragement and support that made the completion of the thesis a reality. Your inputs have been extremely useful in shaping up the thesis. I thank you very much for being there for me when I needed you.

It is an honor for me to pass my acknowledgment to Prof. David Parnas, the father of software engineering, who put me on the path of this exciting and challenging research, and who tirelessly supervised me, providing his invaluable feedbacks, not only when he served as the director of SQRL but also extending well over a year even after his retirement from the University. Dave, your numerous and timeless ideas, your wisdom and unmatched contributions to software engineering, and your quest for fundamental theories and principles have been a source of inspiration for me throughout my research work. Thank you for everything!

I am indebted to many of the former SQRL staff and colleagues for their support at various times. Dr. Sergiy Vilkomer provided me his feedbacks and advice as a mentor in the early stages of my research. Dr. Markus Clermont, Dr. Tom Arbuckle, Ms. Ita Page, Mr. Simon Marr, and all other staff have rendered me their administrative and technical assistance that was instrumental for the smooth running of my research work. Similarly, I would also like to acknowledge Dr. Goyal Suresh and Dr. Jeff Punch, from CTVR, for their cooperation.

Sincere thanks are extended to Dr. Annette McElligot, head of CSIS department, who had made available her support in a number of ways. My thanks also go to Professor Pierre-Jacques Courtois, AVuclear, Authorized Nuclear Safety Institute, and Prof. Xie Min, National University of Singapore, for their valuable comments, based on the brief interactions I had at the early and late stages of my research, respectively.

Many people outside my research center have also contributed by making Ireland a convenient place for my stay. Many thanks to Mr. Kieran O’Farrell and Mr. James Harnedy from Galty Technologies, who provided me the opportunity to work on an exciting project and allowed me to use it as my case study in this thesis, giving me the chance of trying out many of my ideas. I gratefully thank Dr. Patricia O’Flaherty and her
daughters for their kind hospitality for me and my family. Credit also goes to Ms. Jennifer Cleary, Research Mobility Network, for filling a gap in an otherwise difficult system.

My deepest gratitude and respect goes to my mother Mulu Jemberie, and my late father Dagne Semegn, who raised me with care and love, who instilled in me self confidence and determination and who taught me the great lessons of life.

I am blessed with so many friends and family members, whom I have relied on for their advice, support and encouragement that had impacted my work tremendously, and to whom I am extremely grateful. My appreciation and thanks goes to my brothers and sisters, for their support. It is a pleasure to thank Dr. Yohannes Getnet, who filled the void that I left back home, and Dr. Melaku Geboye Desta along with Yilikal Admassu, for the enlightening exchange of ideas on various topics and their consistent encouragement throughout the research period.

Finally, I wish I have special words that can express my gratitude to my darling wife Wubayehu Zewdie, who paid the greatest sacrifice and contributed the most. Wubua, in this journey of research, your unconditional love, extraordinary strength, and unwavering support have created the energy and your insightful ideas and guidance have given me the light, making your role for the success of this work to have no parallel. You are incredible! A million thanks and deepest love to my gorgeous children: Soliyana and Kaletsidk, for giving me unlimited happiness. You have made my life complete!
Dedication

I write this dedication while in a state of deep mourning at the loss of my dear mother, Mulu Jemberie, who passed away on 12 November 2009, just thirteen days before my viva. I did not even say a proper good bye to her when I left her over four years ago to come to Ireland for my PhD; I was sure I would see her again, with my PhD done. It was not to be. I miss her today, just as I miss my father, Dagne Semegn, who died 9 years ago; I will miss her every day as I will miss him. One thing I know is that the two are now together, in heaven. I hope they are happy with what I have achieved, and proud of the job they did in making me who I am. This work is dedicated to the memory of my two loving and caring parents. This is the only gift I have for them. May their souls rest in peace! Amen!
List of Figures

Figure 2-1: Software Reliability Predictability Problem .................................................. 15
Figure 3-1: Research Areas Related to Design for Reliability Predictability (DRP) ....... 16
Figure 5-1: TFM Table for Requirements of Keyboard Checker Software (Parnas, 2009b) ................................................................. 80
Figure 6-1: Reliability structure diagram for three elementary system configurations .... 93
Figure 7-1: Logical relationship between different aggregates of software system ...... 111
Figure 7-2: Hierarchical Aggregate Structure................................................................. 114
Figure 7-3: Flow-graph representation of an interface connection matrix ................. 131
Figure 7-4: State Transition Diagram for an Interfering Protective Component .......... 146
Figure 7-5: Block diagram of a connection enabled component .................................. 151
Figure 7-6: Composite Structure from Connection enabled Components ................. 151
Figure 7-7: Event structure in a simplified version of MSS .......................................... 160
Figure 7-7: Event Flow Graph for a simplified MSS ..................................................... 162
Figure 8-2: Flow Graph of Component State Transition .............................................. 170
Figure 8-3: Event Flow Graph for a producer-consumer process ................................. 188
Figure 8-4: Flow Graph for a producer-consumer process without the effect of missing deadline ............................................................................................................. 189
Figure 8-5: Flow Graph of Birth Death Process ............................................................ 204
Figure 8-6: State Transition Diagram modeling Possible Deadlock Occurrence ...... 211
Figure 8-7: State Transition Diagram for a protective component .............................. 219
Figure 8-8: State transition diagram for a supportive component .............................. 222
Figure 8-9: Flow Graph Representation of Series and Parallel Connections .......... 242
Figure 9-1: MSS Network Structure .............................................................. 252
Figure 9-2: Simplified View of Interaction with MSS ................................................. 253
Figure 10-1: EFG displaying only MSS event that are connected to Undesired Events 266
Figure 10-2: EFG in relation to SRCL-TimeOut ......................................................... 267
Figure 10-3: EFG in relation to LRCL TimeOut ......................................................... 269
Figure 10-4: EFG modeling the SRCL and LRCL Timeout Events .............................. 272
Figure 10-5: Simplified EFG for the SRCL timeout event (MSS sequential process) .. 273
Figure 10-6: EFG for the LRCL Timeout event (MSS sequential process) .................. 276
Figure 10-7: Simplified EFG for the LRCL Timeout event (MSS sequential process) ... 276
Figure 10-8: Flow Graph for combining Deadline Reliability (MSS Sequential Process) ......................................................................................................................... 279
Figure 10-9: Flow Graph for Representing State of Queue for availability of space .... 284
Figure 10-10: Flow Graph for State Transition of Producer-Consumer Processes with Synchronized speeds .................................................. 289
Figure 10-11: Flow Graph for evaluating cumulative deadline (MSS Concurrent Processes) ......................................................................................................................... 291
Figure 10-12: EFG model SRCL Disconnection and Reconnection ........................... 306
Figure 10-13: Signal flow graph modeling SRCL disconnection .............................. 310
Figure A-1: Composite Structure for MSS (Partial) ...................................................... 372
Figure A-2: Connection Graph ................................................................................... 374
Figure A-3: Event Flow Graph of MSS (Partial) ......................................................... 377
List of Tables

Table 3-1: Summarized Debate on N-Version Programming (NVP) ............................... 28
Table 3-2: State based models .......................................................................................... 43
Table 3-3: Path-based models ........................................................................................... 46
Table 3-4: Additive models .............................................................................................. 47
Table 4-1: Comparison between Hardware and Software Failure (Wallace R. Blischke and Murthy, 2000) ............................................................................................................ 55
Table 4-2: Faults introduced during different life-cycle phases(Yu, 1998) ...................... 57
Table 4-3: Percentage of defects per category (Villemeur, 1992a, p 646) ....................... 58
Table 7-1: Hierarchical Aggregation of Software Parts ................................................ 111
Table 8-1: Possible Reliability Roles of Components in a Service ................................. 218
Table 9-1: Source Code Statistics for MSS & its Environment ..................................... 261
Table 10-1: Interaction Failure Components: ................................................................. 263
Table 10-2: Parameter Values in MSS ............................................................................ 281
Table 10-3: Deadline Reliability ..................................................................................... 281
Table 10-4: Process Partitions in MSS .......................................................................... 282
Table 10-5: Waiting times and deadline estimates ......................................................... 286
Table 10-6: Deadline Reliability ..................................................................................... 287
Table 10-7: Deadline reliability between events a and b ................................................ 292
Table 10-8: Overall deadline reliability for deferent workloads .................................... 292
Table 10-9: Deadline reliability for different design options ........................................ 293
Table 10-10: Service output classification ..................................................................... 312
Table 10-11: Classification of components by their reliability role ............................. 312
Table 11-1: Summary of design imperatives in relation to software architecture and design process ................................................................................................................. 322
Table A-1: Module Guide Summary of MSS .................................................................. 348
1. Introduction

1.1 Background
Software is one of the most complex engineering products. While it has been among the key components in changing the society from “industrial age” to “information age”, its complexity has also been a major bottleneck for the success of various projects. For instance, the key question that could not be answered with respect to United State’s strategic defense initiative (SDI) project was dependability of the software (Parnas, 1985).

Recognizing the complexity of software, the interest to consider it as an engineering product that needs to be produced through an engineering process dates back as early as the 1960’s. Although, various technological advances have been made in the past four decades, many issues that were discussed in the 1968’s NATO sponsored conference (Nato, 1968), are still unresolved.

One such example is related to a reliability requirement that was reported in that conference which reads as: “a design requirement for our Electronic Switching System was that it should not have more than two hours system downtime (both software and hardware) in 40 years.” (Nato, 1968, p 323-326). Such requirements are hard to be guaranteed even today, after nearly 40 years in light of the inherent complexity of software systems and the daunting task of software reliability prediction.

The main reason for this is the complexity of software systems, which some argue to be essential (Brooks, 1987) and cannot be avoided. As remarked by Dijkstra, “software presents the only discipline and profession where an individual’s skull has to bridge from a bit to hundreds of megabytes, a gigantic ratio of $10^9$ that baffles our imagination” (Dijkstra, 1988).

As a result, mastering the complexity of software has been the hallmark of software design, besides achieving other qualities. To this end, the principle of information hiding (Parnas, 1972a), that laid the foundation of today’s technologies (IEEE, 2007), has practically proved itself to be the most important concept that not only helps to master
complexity but also provide such qualities as changeability, maintainability, reusability, and evolvability.

The over 30 years-old, yet timeless, principle of software design states that “the design begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. The interface of a module should be chosen to reveal as little as possible about its inner workings.”

This principle has become the foundation of object oriented design (IEEE, 2007) and many other current technologies, such as *design for reusability* (Erich et al., 1995) in object-oriented design patterns, and *design for composition* - (Szyperski, 2002) in component-software where interfaces get self standing position in the system and serve as contracts between components. Component level changeability and structural relations are also at the heart of what some authors call software architecture (Reddy et al., 2006, Guijun and Fung, 2004).

More recent programming efforts coin the term *aspect* which is claimed to address *separation of concerns* (Gregor Kiczales et al., 1997, Atkinson and Kuhne, 2003, Schult and Polze, 2002). The concept in there is basically aimed at addressing some of the limitations of object-oriented languages in supporting changeability, more than problems in designs. Otherwise, its main goal stated as *changeability*, is the same goal that started the *design for change* paradigm three decades ago. A similar work on N-degree *separation of concerns* is also reported in (Tarr et al., 1999).

Unfortunately, these technological developments do not guarantee anyone to meet stated reliability targets nor do they enforce the application of sound principles. Sophisticated tools may come with sophisticated problems resulting in difficult-to-uncover bugs. Programmers can use/abuse tools to construct badly structured programs whose correctness cannot be verified. Too much emphasis in coding and unjustified dependency on testing results in systems that continue to grow in size and complexity with increased uncertainty on their dependability.
As a result, software failure is still common place, causing substantial economic problems as well as life loss. Some examples include:

- A study by the National Institute of Standards and Technology (NIST) indicates that software errors cost the U.S. Economy about $59.5 billion (0.6%GDP) annually (RTI, 2002, Newman, 2002). The same study indicates that “software developers spent 80% of development costs on identifying and correcting defects, and yet few products of any type other than software are shipped with such high levels of errors.”

- The cumulative effect of a chopping error that missed 0.000000095 sec in every 10th second, accumulated over 100hrs to create a discrepancy of 0.34 secs. resulted in missing of a Patriotic missile to intercept a Scud missile with a consequence of loss of life of 28 people (GAO, 1992)

- Software problems in radiation therapy equipment, Therac-25, is believed to have caused at least six known accidents of massive radiation overdoses resulting in deaths and serious injuries (Leveson and Turner, 1993).

- The inquiry commissions report on the failure of flight 501 of ARIAN 5 (Lions, 1996) which is one of the most costly system failures due to software bug, shows that the root cause of the disaster is because of unhandled exception resulted from an operand error.

In general, “software crises”, a term that was coined many decades ago to indicate the challenges of software construction is still with us as a chronic problem. To this date, while hardware is sold with guarantee or warranty, software is sold with a disclaimer.

To address reliability prediction of software systems, various approaches have been proposed over the years. Two classes of reliability modeling approaches have evolved: black box approaches (classified as software reliability growth models) and white box approaches (classified as architecture based reliability models).

The first category does not take the internal structure of the software into consideration and as a result is not very effective for predicting large-scale software systems composed of various components.
The models in the second category have been proposed to take the structure of a software system into consideration and to predict software system reliability from the reliability of its components and their interconnections. Most architecture based reliability models assume components to fail independently and only when executed (Swapna, 2007) (Katerina Goseva-Popstojanova, 2001). Many of them also assume components to be sequentially executed and appear in series connection in terms of reliability.

The main difference between these models is how they account for variations on usage profile of components. In most cases, software architecture is modeled by the ‘transfer of control’ among components, whereby the relative contribution of the components is estimated. Essentially, this is an effort to estimate what classical reliability theory calls \textit{the duty cycle} (Bazovsky, 1961) of components in the system.

Only a few conceptual models (Littlewood et al., 2001) are proposed to address the issue of reliability estimation in the case of redundancy through N-version Programming (NVP) (Avizienis, 1995). These models include those reported in papers (Bev Littlewood Peter et al., 2000), (Wattanapongskorn and Coit, 2007), (Zafiropoulos and Dialynas, 2004), (Ege et al., 2001) and (Ping et al., 2008). However, due to the difficulty of obtaining reliable data to support various models, there are no universally accepted models for estimating the failure rate of redundant software versions.

Besides N-version programming, there are various ways to achieve software fault tolerance including multilevel restoration systems. One model that considers the possibility of multilevel restorations is provided by Vilkomer et. al. (Vilkomir et al., 2005). Like many other models that model software systems at different levels of performance, this model does not use system structural information.

The author’s analysis reveals the following basic gaps between possible software structures and architecture-based reliability models:

- For a network of components that can be executed concurrently, either in a distributed or non-distributed environment, the concept of ‘transfer of control’ is not well defined and may not exist at all. Many components can be active or operational at the same time communicating with each other using some communication signals.
• There are a wide range of reliability improvement components, such as protective components, redundant components, failover switches, backup and restorative components where the ‘series connection’ assumption does not hold.

• Design approaches that allow fault-tolerance and degradation behavior result in components that differ in level of criticality, level of importance, type of output and level of relevance in that duty cycle alone does not fully explain the relative importance of components to the system reliability even for sequentially executing components.

• In addition to component primary failures caused by design faults in each software component, components are also subjected to dependent failures in which they fail due to failure of other components in their environment, and interaction failures, in which no particular component may be singled out as the cause of failure but the joint interaction of various components results in undesired events that lead to an unacceptable system behavior. Such dependencies require precise interconnection information.

The gap between software structure and architecture based reliability prediction approaches may have two causes: poor structure where the reliability as well as functional role of components cannot be known precisely or lack of precise specification or description of system behavior in well-structured systems where the functional as well as the reliability role of components can clearly be identified.

In both cases, the problem is related to that of a software product, where, by product the author means including its specification and description. Generally, the problem of assessing the reliability of software systems is highly dependent on the product characteristics, its components, their interconnections, the operational environment, and the presence of precise specification of the required behavior to compare with a precise description of the actual behavior.

It is not possible to predict the reliability of a product whose requirements are not known since it cannot be determined whether an output or an observed behavior is acceptable or not. Similarly, it is not possible to make structural assessment about the product if its components and their relations are not known precisely. For this reason, this author has
focused on the study of the required behavior or properties of software design that will allow the effective application of relevant structural reliability assessment techniques on software structure to estimate its reliability from the reliability of its components and their interactions.

This is a challenging problem since, even when all information is available, structural reliability assessment of component based software systems is believed to be one of the most difficult problems in software engineering. Kappes et. al. (Kappes et al., 2000) proved the absence of an algorithm (and hence program) that can calculate or even approximate the reliability of component oriented software systems by showing the problem to be equivalent to the well known halting problem. Butler et. al. (Butler and Finelli, 1993) have discussed the infeasibility of quantifying the reliability of life critical real-time software. Littlewood et. al. (Littlewood and Strigini, 2000) commented on the difficulty of pre-assessment of reliability of software systems.

### 1.2 Objective of the Research

The main objective of this research is to identify design properties and documentation techniques that make the reliability of a software product easier to predict based on the analysis of the reliability of its components and information about their interconnections. This author calls the problem of predicting the reliability of software systems, the \textit{reliability predictability problem} and tackles it from three tracks:

1. Identifying design imperatives that will make the system behavior easier to predict
2. Identifying mathematical documentation techniques to describe the behavior of software systems
3. Adapting structural reliability modeling techniques to predict the reliability of software systems based on their mathematical description
1.2.1 Identify Design Imperatives for Improved Reliability Prediction

The first objective of this thesis is set to identify design imperatives that will make the system reliability easier to predict.

One may develop systems whose behavior is hard to understand and describe using any approach. Such systems may have no uniquely identifiable components whose interrelationships can be precisely known. Despite the various principles and methods that have been proposed to improve the structure of software systems, the software industry is still full of badly structured systems. There are still programmers that write as much as fifteen thousand of lines of code in a class (a module in object-oriented programming language) and project managers that accept such systems as product. Such poorly structured systems cannot be analyzed with any realistic tractable model.

Structural reliability assessment can be applied only when the structural relation among the different components in the system can be clearly identified and known. Thus, one part of this thesis is to identify the necessary decomposition and composition criteria that will make a software product easier to analyze and its reliability simpler to predict.

1.2.2 Identifying Mathematical Documentation Techniques to Describe the Behavior of Software Systems

The second objective of the thesis is finding ways of mathematically describing the behavior of software systems.

Other engineering disciplines such as Electrical and Mechanical Engineering depend on mathematics to completely characterize and precisely analyze different aspects of their systems. The mathematical models provided in the form of transfer functions or system of equations gives all the necessary information required to study the behavior of their products.

In software engineering, this is not common. The descriptions about the product are usually informal or semiformal which are often known to be ambiguous, inconsistent, and difficult to keep up to date with the actual product. In the absence of precise relevant documentation, it is not uncommon to resort to reading program code for reliability
assessment or other purpose. However, deriving the necessary information from program code is not practical, if not impossible, because the code contains too much implementation detail to be useful in creating a tractable model. Besides, program codes tell us what the system does, not what it is supposed to do. Therefore, another part of this thesis is aimed at identifying precise mathematical documentation technique of software products that would help to derive its reliability structure.

### 1.2.3 Predicting the Reliability of Software Systems based on their Precise Mathematical Description

Although many reliability models have been proposed over the years, nearly all of them are based on pure theoretical or mathematical basis having very little connection with actual software structure, its behavior and its failure characteristics. Only few, such as the one discussed in (Katerina Goseva-Popstojanova et al., 2005), attempt to connect reliability models with real software products. Even in such cases, the models usually take simplifying assumptions on software structure which makes the reliability prediction questionable. These problems arise in attempting to derive structural information that is necessary for reliability prediction based on program code. Program code can generally be considered as a poor source of information for reliability assessment because

- It contains too much unnecessary details that make the resulting information intractable to model
- The reliability role of components may not be clearly identified from the code
- The textual organization of the program code could be quite different from its runtime organization

Therefore, the third objective of this thesis is to use precise mathematical descriptions of software for making reliability assessment.

### 1.3 Achievement of the Thesis

This thesis documents the result of these investigations. It introduces a novel approach of designing, specifying, and describing the behavior of software systems in a way that helps to predict their reliability from the reliability of the components and their
interactions. It identifies design imperatives and relevant mathematical documentation techniques for improved reliability predictability of software systems.

The design approach, which the author names, *design for reliability predictability* (DRP), integrates *design for change, precise behavioral documentation* and *structure based reliability prediction* to achieve improved reliability predictability of software systems. It enhances changeability to the level of supporting modular redundancy (when necessary), fault tolerance and restorability for improved reliability.

However, since the gain in reliability from redundancy can be compromised due to statistical dependencies and interaction failure causes, the design is supplemented with precise documentation and structure based reliability assessment techniques.

The specification and description approach builds upon precise behavioral specification of interfaces using the trace function method (TFM) (Parnas and Dragomiroiu, 2006), (Parnas, 2009b) and introduces a number of structure functions: module-interface connection matrix, input-output connection matrix and event-flow-graphs. These functions capture both the static and dynamic behavior of software system and are used to derive reliability structure functions.

The author integrates these structure functions with standard reliability and probabilistic models to develop a novel document based multilevel structural reliability assessment technique.

DRP has been successfully applied in the design and development of a mobile streaming system (MSS) as a case study. MSS is a system that consists of a network of software, hardware, and wireless communication components in a distributed and stochastic environment with real-time characteristics. The time sensitivity and unreliability of the wireless channels, the non-fault tolerant behavior of the peripheral device (where missing of a single bit from tens of megabits causes unacceptable output), and the limited computing and storage resource size of the mobile device make the system non-trivial to design, document and perform reliability assessment.

These features make the system reliability sensitive to its various structures, notably its event structures. As a result, various design alternatives have been assessed based on
their reliability estimates. The design with best service reliability from the different alternatives is implemented finally. The main lesson learnt from the case study is a successful marriage among the three components of design for reliability predictability, which are: design for change, precise behavioral specification/description and structure based reliability assessment that allowed quantification of different reliability concerns among different design alternatives.

### 1.4 Novelty of The Work

The main contribution of this work is the integration of design for change, precise specification and description, and structural reliability prediction into one design approach, which is called design for reliability predictability (DRP).

The *design for change* paradigm that has been the main focus of research in the past couple of decades is enhanced with the objective of reliability to support modular redundancy, whenever needed, restorability, and fault-tolerance. The contribution in here is the application of changeability property of designs for creating a reliability structure that is possibly different from mere series connection of components. Having assessed various design approaches proposed over the years, this author is convinced that properties, such as changeability, are design features rather than specific tool or programming language features. Even when one applies identical design techniques and tools, various designs may differ from each other in their qualities in general and reliability in particular.

The *specification and description* is built upon a newly developed black box specification technique, called the trace-function-method (TFM) and introduces a number of novel structure functions, namely ‘module-interface connection matrices’, ‘input-output connection matrices’, and ‘event-flow-graphs’. Of course, connection matrices are well-known mathematical models used in different disciplines such as in the functional description of switching networks (Deo, 1974). The contributions in here relates to their application in specification and/or description of a software structure.

The document based multilevel structural reliability assessment approach reported in this thesis differs from those covered in the literature in the following ways:
• It is based on precise component and system behavioral documentation as opposed to program code, or informal or semi-formal documentation
• It makes use of various structures about software systems: module-interface structures, composite structures, event-structures, type structures, etc. instead of just ‘control-transfer’ concept
• It does not consider all components to be in mere series connection but rather uses the reliability role played by each component, its failure mode and effect. To this end, components may play a protective role, redundant role, restorative role, supportive, etc.
• It addresses reliability assessment at least from three levels – component reliability, interaction reliability and service reliability
• It takes into account failure causes that may arise due to interactions among a network of components and scarcity of resources which has barely been explored in the literature. Various interaction reliability estimation formulas are also derived based on relevant reliability or probability theory.
• Reliability assessment is not limited to the possible presence or absence of incorrect program statements or bugs. It is rather an assessment of the possible occurrence of undesired events that affect required system outputs.

1.5 Organization of the Thesis
The rest of the document is structured as follows: Chapter 2 discusses the research question and approach taken to tackle the problem. Chapter 3 explores related work to this thesis. Three areas of research are covered: software design and construction, specification and description and structure based reliability prediction. Chapter 4 provides analysis of the problem where software failure characteristics and software fault mitigating approaches are discussed. Chapter 5: provides the definitions and basic concepts used in design for reliability predictability. Chapter 6 identifies the design imperatives based on reliability theory. These imperatives are further elaborated and mathematical models are developed in chapter 7. Chapter 8: Gives the document based structural reliability assessment method developed by the author. Novel formulas for component reliability, interaction reliability and service reliability are provided. Chapter
9: Presents DRP as applied to a case study, a mobile streaming system (MSS). Chapter 10 presents reliability assessment of the case study. Chapter 11 covers the results, discussions, conclusions and open questions.
2. Research Question and Approach

2.1 Introduction
This chapter states the main research question and provides the approach taken by the author to find a solution to the research problem.

2.2 Research Question
The main research question of this thesis is stated as follows:

What design imperatives and documentation techniques can help to make the reliability of a software product more predictable based on analysis of its structure and information about the reliability of its components?

Specifically, the thesis seeks to find answers for the following questions:

- Can we improve software reliability predication by applying specific ways of software design?
- What are the design criteria that will help us to better estimate/predict the reliability of a system composed of components whose reliabilities are known?
- What kind of mathematical documentation and model would enable us to understand the reliability structure of software systems?

2.3 Research Approach
Figure 2-1 shows a pictorial description of this author’s approach to the problem. Existing practice in software design can be classified into two: software that does not have well defined structure to make any significant structural reliability analysis or software with some structure, but the structural information is far from what is needed for reliability analysis.

The former type of software systems can only be studied through the application of what is known as black box reliability modeling approaches, also called reliability growth models. However, these approaches do not take advantage of software structure and hence are less effective for predicting complex systems.
The later type of software may have some structures but the description of these structures is usually based on design criteria that may not have any connection to reliability. As a result, architecture based reliability models that have been proposed over the years to predict such software systems suffer from either of two limitations. Either they make simplifying assumptions on the software structure making the prediction far from the reality or when they attempt to make the models closer to the structures, the models become too complex to be understood.

The approach taken in here is thus to study structural reliability assessment methods along with software failure causes in order to identify reasonable assumptions that can be used as design imperatives for improved reliability predictability software systems.
Figure 2-1: Software Reliability Predictability Problem
3. Related Work

3.1 Introduction:
This chapter covers the research areas that are related to the work reported in here. The author identifies three areas of research, each of which with huge amount of literature spanning over many decades that are at least partially if not completely and directly related to this research. These areas, the proposals, achievements and limitations are thoroughly discussed in the sections to follow.

3.2 The three Tributaries of Design for Reliability Predictability

Figure 3-1: Research Areas Related to Design for Reliability Predictability (DRP)
Figure 3-1 shows the three areas of research work that are partially related to this work, which are:

1. Software Design and Construction (SDC)
2. Specification and Documentation (S&D)
3. Structure Based Reliability Predication (SRP)

Many decades of research effort have been spent in each of these areas independently in that there is too much literature to be fully covered as related work in one thesis. However, the integration of these three areas as a unified research work has not been addressed thoroughly or adequately enough. Additionally, the software reliability problem is a critical problem that continues to affect many projects negatively.
Thus, this thesis attempts a new beginning in integrating these three components into one unified design approach in order to improve the quantification of the reliability of the software products we construct.

This, however, does not mean that each of the three areas exist in complete isolation to each other. A substantial part of research in design methods has focused on improving reliability of software systems. Many go to the extent of providing mathematical proof for correctness. Research on specification and description techniques have also aimed at improving system reliability by reducing ambiguity and inconsistencies while structural reliability prediction efforts attempt to quantify reliability of a system from its components. Software reliability engineering (SRE) (Lyu, 2007), which is defined as “the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability”, attempts to span different phases of software development but had mainly focused on prediction of software reliability based on statistical testing and use of software reliability growth models.

In the sections given below, the three areas that are relevant to this work are discussed.

### 3.3 Main Principles in Software Design and Construction

Software design is a process of transforming software system requirements into a set of interacting components, also called modules, where the components are either available as already implemented/reusable components or are implementable as programs, and can later be composed together to provide the functionalities stated in software system requirements.

Analysis and synthesis of any complex system requires the ability to decompose the system into components (*decomposition*) and to combine components to form the desired system (*composition*). The main goal of this principle is to master complexity and the principle has been in use for a number of centuries throughout human’s technological development.

As the task of software development became more than a personal programming activity, but rather a “*multi-person construction of multiversion programs*” (Parnas, 1975a), then followed the need to adapt the same principle to software design. Thus, the idea of
decomposing software systems into components and composing the components to form the desired system is as old as software engineering itself.

Sometimes, the terms modularization and integration are used to refer to decomposition and composition respectively. Similarly, in the literature the terms module and component seem to take similar meanings although there are many more terms along with disagreements on their meanings. Many times, the meanings are mixed with specific design methodologies, techniques and even programming languages that lead to coining of new terms every time the techniques, methods and/or languages change. For instance, Szyperski lists 14 different definitions for a software component and adds his own (Szyperski, 2002).

Parnas makes a subtle distinction between module and component. According to his definition “a module is a collection of programs to be created as a single work assignment by a programmer or group of programmers” and “a component is a collection of programs that are distributed as a unit and used in larger systems without modification.” (Parnas, 2009b)

The distinction is in the objective, where a module is taken as a unit of software development, while a component is a unit of software distribution and deployment. “A component may comprise several independently developed modules; a module may be included in several components” (Parnas, 2009b).

Today, the advantages of modularization are well known. The main difference among different software modularization techniques proposed over the years lies on the criteria chosen for decomposition. Some of the key principles or criteria are discussed in the following subsections

3.4 Design for Correctness by Proof

The development of the structured constructs (concatenation, selection, repetition) and virtual machines concept (Dijkstra, 1970, Dijkstra, 1968) was motivated by the desire to prove program correctness. Dijkstra considers software to be a series connection of components, where the probability of correctness of the whole program consisting of N components is given by something like $P=p^N$, where $p$ is the probability of correctness
each individual component. He notes that for \( P \) to be significantly different from zero for large \( N \), then \( p \) should be very, very close to 1.

Besides introducing the three constructs, Dijkstra also proposed to organize programs as a hierarchy of *virtual machines* for better understandability and ease of ensuring their correctness. He used hierarchical organization in his work on the T.H.E. multiprogramming operating system (Dijkstra, 1968).

With the virtual machines approach, the main program is viewed as if “it is executed by its own, dedicated machine, equipped with the adequate instruction repertoire operating on the adequate variables and sequenced under control of its own instruction counter.” It is hoped that the correctness of a program constructed in this manner can be discussed and established even if the dedicated machine is non-existent in reality. This virtual machine, the ‘upper machine’ will then have to be constructed by deciding upon the data structures to provide for its state space and developing various algorithms, each of them providing an instruction assumed for the order code of the upper machine. The ‘lower’ machine may have a set of private variables, introduced in its own benefit and completely outside the realm and scope of the upper machine. The programs written may also not exist at lower level and hence they have to be simulated by the next-lower level machine, and the process continues until a program that can be executed by the hardware is obtained.

Despite the absence of program proving mechanisms for large industrial software projects, the *structured constructs* laid the basis for the structured programming methods that were popular in 1970’s and 1980’s. The constructs are still at the heart of imperative programming languages that include today’s object-oriented languages.

Although the method provided an improvement over the predecessor non-structured design approaches and became popular for over a decade, some limitations became apparent. These limitations include:

1. Changeability – programs organized using the structured constructs use very little abstraction concepts. The organization of the program texts follows directly from what is thought to be the correct way of sequencing the runtime events. If the assumptions about the sequencing later changes, then the software become useless.
2. Independent Development - designers of structured software systems have to agree on certain control flows, calling conventions, etc about their programs. This requires a lot of communication and understanding among themselves, making less room for independent development. For complex problems that involve hundreds of developers, the design approach could not scale up easily making timely completion of software difficult.

“As a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than to try to ignore them, for the latter vain effort will be punished by failure.” Dijkstra

3.5 Design for Change

A breakthrough in software design was achieved through the introduction of the information hiding principle (Parnas, 1972a) which is now accepted to be the basis of many of the recent technological advances (IEEE, 2007).

According to this principle, a module is characterized by the design decisions it hides from all other modules. It provides an abstract interface through which its services can be accessed. This interface hides the implementation details of the module that remain as the module’s secret.

The main advantages of applying the principle are changeability, independent development and comprehensibility. These and various other advantages have been demonstrated in a number of papers. Using this principle, Parnas was able to produce 192 quite distinct versions of a KWIC index program using 15 modules (each module satisfying one of the 5 module specifications) produced by 15 students independently as early as the fall of 1971 (Parnas, 1975a). The modules were interchangeable and the testing of a sample of 25 of the versions was reported to be successful. In (Parnas, 1979), the information-hiding principle coupled with the virtual machine concept and the uses hierarchy was used to demonstrate how software could be designed for ease of extension and contraction.

Prior to the appearance of the information hiding principle, a module was considered as a major step in processing. It usually results from program decomposition based on a flow-
chart. However, the information-hiding module, as introduced in Parnas’ paper does not correspond to a step in processing. Some modules could be used almost always, while others may or may not be used during processing.

Parnas writes, “the process of decomposition starts from identification of difficult design decisions or design decisions that are likely to change. Each module is then designed to hide such a decision from others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing.”

Parnas notes that at run time, one may not be able to distinguish the criteria that were used to decompose the system into modules. By using appropriate tools for assembly, modules decomposed based on the information hiding principle may be bound together to form an efficient system that may be identical to a system composed of modules that were decomposed based on some other criteria.

Parnas advises “to achieve an efficient implementation, we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collection of code from various modules.”

The principle has helped to tame the inherent complexity of software development and improve many of its qualities such as reusability, interoperability, maintainability, evolvability, etc. As remarked by Weiss, (Weiss, 2001), “a module may be a design-time entity, or a load-time entity, or an entity created at some other time.”

Various concepts that have been coined to communicate specific design approaches such as separation of concerns, encapsulation, abstraction, polymorphism and virtualization, are all different manifestations of the information hiding principle.

The evolution of technologies, mainly programming languages, from structured design (Dijkstra, 1968), to object-oriented (Object Management Group Inc. (OMG), 2002), (Erich et al., 1995), component-oriented (Szyperski, 2002) and aspect-oriented (Gregor Kiczales et al., 1997), also highlight the unique place taken by the information hiding concept in software design.
The introduction of the abstract data type (ADT\textsuperscript{1}) as a programming language concept in 1974 (Liskov and Zilles, 1974) with the initial objective of supporting structured design, seems to have facilitated the application of the information hiding principle at programming language level. This paved the way for the popularization of the object-oriented design methods, which are characterized by the three concepts: \textit{encapsulation}, \textit{polymorphism} and \textit{inheritance}.

From the three features, while the first two are equivalent to \textit{information hiding} and the third, \textit{inheritance} (i.e. implementation inheritance) became contentious mainly because of its potential to violate the information hiding (or encapsulation) principle. A widely cited problem associated by lack of changeability due to inheritance is known as ‘the fragile base class problem’ (Szyperski, 2002). Another limitation of object-oriented programming in its early days was unrestricted dependency on implementation. In many programming languages, classes represent implementations and they normally depend on each directly. As a result, the only changes that are supported are modification (re-writing the class code) of the internals of classes but not their replacement by other compatible classes (components).

These problems were later addressed by separating interfaces from classes, emphasizing polymorphism and object compositions over inheritance, as is evident from the seminal study of design patterns of object-oriented systems. Based on their study of hundreds of patterns, Erich et al. recommend two principles of reusable object-oriented design (Erich et al., 1995):

1. Program to an interface not an implementation
2. Favor object composition over inheritance

The move to the component-oriented-model from the object-oriented approaches, as implied in the book by Szyperski (Szyperski, 2002), can also be viewed as a growing emphasis on the separation of the interface from the implementations as well as increasing preference to composition over inheritance. In these models, interfaces have taken a self-standing position and are considered as contracts between clients and

\textsuperscript{1} An abstract data type defines a class of abstract objects, which is completely characterized by the operations available on those objects
providers. Interfaces are used as a means through which components interact. An interface contains a set of named operations. The interfaces specification plays a dual role as it serves both providers – that implement the interface and those clients that make use of the interface.

Components, which are concrete implementations, can either become providers or clients of interfaces. The providers and clients do not need to know each other, as they need only to comply with the specification of the intermediate interfaces. For this reason, interfaces and their specifications are normally viewed separately from the specification of any component that may be a provider or client of the interface. In the component-oriented models, the ‘specification’ of components may be given by naming all interfaces that components need to adhere to and any additional properties that they may comply to. A component may implement one or more number of interfaces.

The same interface can then be used by a number of different clients and provided by a number of providers. Providers could be substituted for each other if they satisfy the same contract. The substitution can happen at any time – including execution time. A provider can also provide its services to a number of different clients as long as the various clients make use of the same interface implemented by the provider. A system is created by composing components based on their interfaces.

Recent endeavors in aspect-oriented programming(Gregor Kiczales et al., 1997, Atkinson and Kuhne, 2003, Schult and Polze, 2002) are also attempts to separate different system concerns into separate modules and thereby improve the modular structure of the system by overcoming some of the restrictions put in object-oriented languages. The approaches are likely to add some flexibility in programming languages, but as far as the basic principle is concerned, a design decomposition based on the information hiding principle does not normally mix cross-cutting concerns and specific concerns into a module.

Although, the application of any sound principle is not restricted to the use of a particular technology/language and the use of certain technologies does not guarantee the application of any given principle, looking through the above technological evolutions, one can see that the modular structure of software system has been a major focus area of technological research in the past three decades, with the central objective of how to
achieve design for change. This objective was what was clearly spelt out by the information hiding principle.

In summary, while the goal of structured constructs and the step-wise program composition approach may be considered as design for correctness through proof, the goal of information hiding principle can be taken as design for change.

“People learned programming the way we seem to learn natural language; students saw programs and wrote programs with the same patterns as the ones they had seen. Rather than designing in a disciplined, systematic way, they would “think like the computer”, writing instructions in the order they would be executed.”, David Parnas

3.6 Design for Fault-Tolerance

Various software reliability improvement techniques and fault-tolerance approaches have been proposed over the years. These can be studied under two categories:

1. Software redundancy
2. Response to undesired events or exception handling

These are further discussed below.

3.6.1 Software Redundancy

Redundancy is a major approach in producing reliable systems in most engineering disciplines, electrical, mechanical, etc. Two or more components may be said to be working in redundant mode (also called parallel mode) if the components provide similar functions and the system comprising them is to be operational as long as one or some minimal combination of the components provides the function even in cases where others might have failed.

Some architectures use similar concept where replicated components on different computing nodes may be configured to work in redundant mode and the system is considered operational as long as there is an operational path that connects the inputs with outputs. For instance, the Object Management Group (OMG) has included specification for CORBA fault tolerant components in its CORBA specification document (OMG (Object Management Group, 2002)). Design of fault-tolerant CORBA
Components, where components could be deployed in multiple copies and the connectors would be connecting those working components, are reported in (Favarim et al., 2003).

In hardware systems, the components that may be configured to work in redundant mode may be identical (of the same type / same design and production) e.g. two identical light bulbs to illuminate a room, or diverse (of different types or designs etc.). e.g., a light bulb and a florescent. In these systems, there is little difference, if any at all, between using identical components and diverse components as hardware failure is usually considered to be of random nature.

On the other hand, in software systems, the type of redundancy obtained from using identical components (or replicas) is significantly different from those using diverse components. The major limitation of redundancy by using identical components is that the method cannot significantly reduce failures caused by design faults. If software fails to provide correct output for a given input due to an internal bug operating under normal environmental conditions, then all its replicas would fail for the same input conditions.

Therefore, an alternative form of software redundancy has been proposed through the use of diverse components, i.e. components providing the identical service but having different internal design. The proposal for such diversified redundancy is as old as the origins of the computing machine themselves, being over one and half century’s old and tracing back to Charles Babbage’s time. In 1837, Charles Babbage wrote, “when the formula is very complicated, it may be algebraically arranged for computation in two or more distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite sure of the accuracy of them all.” (Avizienis, 1995).

According to the same reference, three years earlier Dionysisu Lardner had published in the Edinburgh Review of July 1834 his article entitled “Babbage’s calculating engine” in which he made the first suggestion on multi-version programming as follows:

“The most certain and effectual check upon errors which arise in the process of computation is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods.”
Later approaches that are commonly referenced in relation to redundancy are:

1. Recovery Blocks (RB) also known as Dynamic Redundancy (DR) (Xu et al., 2002)

2. N-Version Programming (NVP) also known as Masking Redundancy and multi-version software or multi-version design (MVD). (Avizienis, 1995)

The RB approach was originally proposed in the mid 1970s by Randell and is based on the assumption that depicts all software faults result from design errors. It tries to divide a program into a series of blocks that evoke processes that are structured into operations. System structuring is based on the selection of a set of these operations to act as units of error detection and recovery, by providing extra information with their corresponding blocks, and so turning the blocks into recovery blocks.

This approach seems to have been deterred by, among other things, the added complexity and increased error proneness of system structure as described in a later paper. “The various forms of software fault tolerance that have been devised to date either require special language features (provided by a special compiler or preprocessor) or require strict, but unchecked, adherence by a programmer to a set of special programming conventions. The former approach cuts one off from the mainstream of programming language developments, the latter can be dangerously error-prone.” (Xu et al., 1995).

The second approach was introduced by Avizienis & L. Chen in 1977. Avizienis defines the approach as follows. "N-version programming is defined as the independent generation of N >= 2 functionally equivalent programs from the same initial specification. The N programs possess all the necessary attributes for concurrent execution, during which comparison vectors (“c-vectors”) are generated by the programs at certain points. The program state variables that are to be included in each c-vector and the cross-check points (“cc-points”) at which the c-vectors are to be generated are specified along with the initial specification. ‘Independent generation of programs’ here means that the programming efforts are carried out by N individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms and programming languages (or translators) are used in each effort. The initial specification is a formal
specification in a specification language. The goal of initial specification is to state the functional requirements completely and unambiguously, while leaving the widest possible choice of implementations to the N programming efforts. The actions to be taken at the cc-points after the exchange of c-vectors are also specified along with the initial specification.” (Avizienis, 1995)

The major objectives of the NVP process were to maximize the independence of version development and to employ design diversity in order to minimize the probability that two or more versions will produce similar erroneous results that coincide in time for a decision (consensus) action. If the different versions are to have independent failure behavior, it would result in dramatic increase in reliability.

Nevertheless, this was put into question when Knight & Leveson conducted an experiment and published a series of articles highlighting that s-independence assumption cannot be taken for programs developed using NVP. Since then, there has been an ongoing debate about NVP, without much conclusive result (Avizienis, 1995), (Knight, 1989), (Hatton, 1997), (Ege et al., 2001), (Eckhardt et al., 1991), (Bev Littlewood and Miller, 1989), (Littlewood et al., 2001), (Knight and Leveson, 1990), (Bhansli, 2005), (Brilliant et al., 1990), (Butler and Finelli, 1993), (Townend et al., 2001).

Some of the main arguments on both sides are summarized in Table 3-1.
Table 3-1: Summarized Debate on N-Version Programming (NVP)

<table>
<thead>
<tr>
<th>Critics</th>
<th>Supporting argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-versions are not independent: Experimental results did not support the independence of failure assumption (Knight and Leveson, 1990, Knight and Leveson)</td>
<td>“independence of faults is an objective and not an assumption of the NVP approach.” (Avizienis, 1995)</td>
</tr>
<tr>
<td>Common-mode failure is far more prevalent in software than in hardware implementations.</td>
<td>In later experiments, the Knight and Leveson reported that, a three version system with the majority voting was reported to offer only around 20 to 50 times more reliability than a single version system typical of the sample. (Hatton, 1997) Through a different way of analysis of the Knight-Leveson experiment result, Hatton reports 45 as the (pessimistic) improvement factor as opposed to the 833.6 that is theoretically evaluated.</td>
</tr>
<tr>
<td>The Consistent Comparison Problem (Knight, 1989):When versions make comparison involving the results of finite-precision calculations, it is impossible to guarantee the consistency of their results and therefore, it is possible that correct versions may arrive at completely different outputs for an application that does not apparently have multiple correct solutions.</td>
<td>The problem that they focus on, inexact computation, in the various systems being compared, is not unique to software. Analog systems have the same problem if voted or compared. No analog devices are manufactured to zero tolerance so there is always some variation. Further, if you go for diversity, there is likely to be more variation. [David Parnas – personal communication]</td>
</tr>
<tr>
<td>Results based on 20 versions of a complex aerospace application developed using four independent programming teams from four geographically separate development sites suggest that N-version programming is effective at coping with the failures of a few bad programs. However, under operating conditions where the programs were uniformly more reliable, the failures that did occur tended to be coincident and have a magnitude that substantially decreased the effectiveness of N-version systems compared to what could be achieved if failures were truly independent.</td>
<td>Townend et. al. (Townend et al., 2001) reports an experiment on safety-critical systems undertaken to compare single version systems with multiversion systems that have been developed using the same level of resource. They conclude- 1) a single-version system is much more dependable than any individual version of the multiversion system, and 2) despite the poor quality of individual versions, multi-version method still results in a safer system than the single-version solution.</td>
</tr>
<tr>
<td>Echhardt and Lee showed that there were limits to what could be achieved in diversity of failure behavior merely by allowing independent development (within a single methodology) by claiming that truly independently developed versions will necessarily fail dependently. (Bev Littlewood and Miller, 1989)</td>
<td>Bev Littlewood &amp; Douglas R. Miller, through conceptual modeling, argue that these limitations can be overcome by adopting a policy of forced diversity at the methodology level. (Bev Littlewood and Miller, 1989).</td>
</tr>
</tbody>
</table>
Despite these controversies, design diversity has been adopted in a limited few industrial sectors. Examples cited include Airbus A320/30/40 aircraft and various railway signaling and control systems. On the contrary, Boeing decided against software diversity for its own 777 aircraft, claiming that it would require restrictions to communication between software and system engineers (Littlewood et al., 2001).

3.6.2 Response to Undesired Events and Exception Handling

Another common form of fault-tolerance is through proper response to undesired events or exception handling. The concept has been introduced since early 1970’s as “response to detected errors” (Parnas, 1972b) and “response to undesired events” (Parnas and Wurges, 1976). Parnas et. al. used the phrase “undesired event” (UE), because:

1. They want to include all events that result in a deviation from normal behavior,
2. The term ‘error’ often led to the objection that ‘errors’ should not be handled but corrected.

The Parnas et. al. idea of handling undesired events was based on the hardware “traps” for error detection and recovery. This has become a foundation for most programming languages of today which mainly call it ‘exception handling’. According to Stuart Faulk, “one sees the same conceptual structures in today’s programming language. For example, the exception handling paradigms of C++ and Java follow the conceptual model first enunciated in this paper” (Hoffman and Weiss, 2001, PP228-229).

In many robust software programs, normal code is only a small fraction of the program, as the code for error detection and handling could take up to two-thirds of a program (Garcia et al., 2001), or even as much as 80% of the code (Parnas, 2006). The available solutions in this area are not yet fully satisfactory for various reasons which include difficult flow-of-control issues and difficulty of exception handling mechanisms in concurrent systems (Garcia et al., 2001).

3.7 Assessment of Existing Design Approaches

It is not uncommon to depict the evolution of design methods, especially those related to imperative languages, to have gone through the following phases as – structured methods, object-oriented design methods, component-oriented methods, subject-oriented methods, aspect-oriented methods, service-oriented methods, etc. The distinction, when it exists
beyond the terminology, may be visible if one looks it from the methodology and/or
technology level as it is usually related to specific techniques, tools, (mainly of
programming languages) and/or methodologies. Beyond languages or technologies, most
terms are synonymous, and may be include the following groupings.

- Decomposition/modularization principle: {information hiding, encapsulation,
  abstraction, polymorphism, separation of concerns, loose-coupling}
- Units of modularization: {virtual machines, modules, abstract data types, classes,
  components, aspects, macros, services}.
- Connection or communication data structures among different decomposition units:
  {variables, objects, connectors}.
- Runtime scheduling units: {tasks, processes, threads, events}.
- Deployment and distribution units: {assemblies, libraries, components}.
- Organizational units: {packages, namespaces}.

However, if one views the various approaches at the principle level, one finds only a
couple of design principles as covered in the previous section. Notably, the structured
constructs were introduced with the main goal of proving programs to be correct while
the information hiding principle came with the main goal of design for change.

The structured constructs became the popular methodologies in the 1970’s and 1980’s for
the improved understandability of software systems over their predecessors, i.e., non-
structured way of programming - although the initial goal of proving programs correct
has not come to be a major industrial practice. However, as the complexity of software
grew, the information hiding principle became better positioned for mastering complexity
and enhancing reusability.

This principle manifests itself through the development of various methodologies
namely: – object-oriented, component-oriented, subject-oriented, and aspect-oriented.
The difference among these methodologies is basically on how to reduce the information
to be communicated among programmers at the time of decomposition.

In object-oriented methods, an object requesting a service from another object will
normally use a reference to the server object. All public properties of the server object are
visible to the client object. In component-oriented models, communication of each
component with another became the concern of the connectors, where the connectors basically represent self-standing interfaces and an underlying mechanism of making run-time binding. The client component may thus view only a partial interface of the server component. In aspect-oriented programming, the attempt is to make client components, more specifically their programmers, to be unaware of the server components that provide the service. However, a common goal of all these approaches is reducing the information to be communicated among developers of various components and thus improving the maintainability of the software as well as the reusability of the components.

Thus, in spite of the various technological jargons that have continued to appear in relation to software design, the underlying principles seem to have been almost stable over the past three-to-four decades. Looking from the principle level, there is not much difference among concepts like: separation of concerns using virtual machines, modularization based on the information hiding principle, object-oriented design, component-oriented design, and aspect-oriented programming. Some of the main differences are:

1. Among the differences between the structured constructs and information hiding principle is that the former had a major goal of design for correctness by proof while the later is mainly aimed at design for change.
2. Another important difference among the two is that structured constructs were concerned about structuring a single executable program while the later is concerned with organizing a large set of those programs.
3. Much of the progress over the years seems to be that of technology (i.e. mainly programming languages). Probably, one important development in this regard is the ability to avoid unnecessary distinction between data and programs. From reliability perspective, replacing one component by another can be as simple as changing one data value by another.

While producing reliable software is a major objective of all design approaches, there does not seem to be much evidence about attempts of designing software systems for improved structure based reliability prediction. The emphasis has mainly been on how to
develop as much reliable system as possible without having to predict its reliability as can be inferred from the following observations:

1. Software documentation usually depicts the functional structure of software components showing how they interact to perform their tasks. It does not normally show the reliability structure of the components. Hence, whether components are in redundant mode or not, whether their failures can be tolerated or not may not be known from the documentations. In other words, even for a properly documented software system using one of the documentation approaches in use today, it may be easy to know what each component in a software system can do when it functions correctly, but how it will affect the system, if it fails is not usually documented, and thus may not be known.

2. Many formal approaches that specify software behavior mathematically, especially from those that are based on pre/post conditions approach, state what happens if certain pre-conditions are met, without stating the effect of execution of the program if the preconditions are not satisfied. As a result, even when a component is formally proved to be correct, its availability, and hence its reliability may not be predictable, as the state of the component could be affected by events/input conditions that do not satisfy the stated preconditions for the component.

3. Although, proposals for how to respond to undesired events in hierarchically structured software systems has been provided since 1970’s and has been supported by most modern programming languages in the form of exception handling, one still encounters costly system failures due to either unhandled exceptions or mis-handled exceptions. The inquiry commissions report on the failure of flight 501 of ARIAN 5 (Lions, 1996) which is one of the most costly system failures due to a software bug, shows that the root cause of the disaster is because of unhandled exception resulted from an operand error. In (Lions, 1996), the following is stated: “In the failure scenario, the primary technical causes are the Operand Error when converting the horizontal bias variable BH, and the lack of protection of this conversion which caused the SRI computer to stop.”

4. Error communication mechanisms in many software designs are largely based on control transfer. The major way of communicating undesired events available in most
of today’s programming languages is through exceptions. This may be a very good
facility to stop execution of a virtual machine, which encountered an irrecoverable
error condition, and replace it with another machine, especially during sequential
processing. However, the mechanism is not adequate enough in an asynchronous
processing model where communication between components may not necessarily be
through control transfers. Generally, when working with software system, no error
information is encoded along with data that is shared among the various components
in the system. As a result, even erroneous data, or data that was sent to describe a
failure condition, could be interpreted as correct data, which was the case that
happened in Arian 5. In (Lions, 1996), it is stated: “Part of these data at that time did
not contain proper flight data, but showed a diagnostic bit pattern of the computer of
the SRI 2, which was interpreted as flight data.”

5. System reliability assessment generally requires the classification of system functions
between main functions and secondary functions and the assignment of the
significance level of these functions. However, prioritization of software
functionalities and/or components in terms of failure does not seem to be a standard
practice in software design except probably in safety critical systems. As a result, it is
not uncommon to encounter total system failures when one tries to make use of a
functionality of a less importance value from the user’s side, e.g., a word processor
crashing when changing the format of a paragraph inside a long document, or going
into failure mode during its auto-initiated recovery save operation.

Most of the modularization approaches discussed earlier focus on the structuring of
programs across modules and structuring of modules across components at the time of
writing or modifying. The runtime organization can be different and can vary
significantly among different design alternatives even when using similar decomposition
methods. This in turn greatly influences different software qualities. Specifically,
reliability is highly dependent on the event or process structure, since this structure
determines the presence or absence of interaction failure causes, restorability, fault-
tolerance, etc. In this respect, many of the design approaches discussed above still depend
upon ad hoc mechanism or qualitative analysis of reliability, not thoroughly dependent on
relevant reliability mathematics.
3.8 Precise Specification and Mathematical Documentation of Computer Systems

Documentation in general may include formal and non-formal ways of describing system behavior. The non-formal may include the use of plain human language as well as some supporting diagrams to describe the necessary information. These are usually known to be ambiguous and often unhelpful when detecting inconsistencies and incompleteness.

Yet, many of the ‘technical’ documentation, that are in use today, allow the mixing of formal and non-formal techniques together, and in many cases the non-formal outweighing the formal. The Unified Modeling Language (UML), that is more commonly used in the industry, is one such example. With such models, even the parts that may be classified as formal may be ambiguous.

To avoid ambiguities and inconsistencies, a useful general approach to precise documentation of computer system elements based on the concept of functions and relations has been proposed (Parnas and Jan, 1995). The research community has also developed various precise aka formal specification and documentation techniques over the years. Some of the well known methods developed in the past include Petri Nets (Adam Petri and Reisig, 2008), B-Method (Abrial), Z-Specifications (Spivey, 1989), VDM (Blaue, 1991), Module Interconnection Languages (MIL) (DeRemer and Kron, 1976), Architecture Description Languages (ADL) (Allen and Garlan, 1997), (Medvidovic and Taylor, 2000) and Trace Assertion Methods (TAM) (Wang, 1994).

Not all these methods have been developed to document the behavior of information hiding modules and their interconnections. Petri Net, which has a precise mathematical definition, was developed to model chemical processes but later was used to study concurrency issues in software systems. Others like Z, VDM and B were not initially designed to work with information-hiding modules as main targets. Hence, they lack some of the necessary properties to elegantly specify or describe information-hiding modules. Some review of the approaches is provided in (Wang, 1994) and (Parnas and Dragomiroiu, 2006).

The trace assertion method (TAM) (Wang, 1994) and its precursors, the Software Cost Reduction (SCR) method (Constance et al., 2005, Heitmeyer and Jeffords, 2007) were
originally developed to address pure black box specification requirements of information-hiding modules. These efforts have grown over decades and recently a new and improved method of specifying or describing the externally observable behavior of information hiding modules, which is called Trace Function Method (TFM) was invented at the Software Quality Research Laboratory (SQRL), University of Limerick (Parnas and Dragomiroiu, 2006, Parnas, 2009b). The method has been used in industry (Baber et al., 2005), (Quinn et al., 2006).

Since TFM documents can help to document all required information about the externally observable behavior of information-hiding modules, including known conditions that would create undesired events, without revealing hidden information that is subject to change, the author uses this method to describe the modules in the system, based on which further structural functions are introduced and structural reliability assessment are made.

3.8.1 Assessment of Existing Documentation Approaches

Unlike other engineering disciplines, the software engineering community has not yet achieved a universally accepted, precise, non-ambiguous, mathematically verifiable documentation system. There are of course various documentation methods proposed over the years. Many of the approaches that are used in the industry are usually informal or semi-formal at best. A popular example of design documentation is UML. Requirements are normally stated in informal documents. Such kinds of documentation can be full of ambiguities and inconsistencies. It is difficult to keep them up to date with the product and to detect incompleteness and redundancies.

Many formal specification techniques proposed or developed over the years introduce special notations and require special tools, hindering their propagation to the industry. Some require very complex logical expressions that compromise their readability and understandability by designers, developers and domain experts.

As a result, the program code is usually taken as the authoritative document when semantic behavior is needed. Nevertheless, program code can consist of not only
required properties, but also accidental properties that are included for implementation purpose only, including incorrect properties that should not be there.

The TFM specification is a new and promising method that can be used as a practical but precise behavioral specification of interfaces. However, it is designed to document the behavior of one interface at a time. To describe the behavior of systems consisting of many components, it needs to be supplemented by relevant structure functions.

### 3.9 Software Reliability Modeling

Software reliability modeling started as early as 1970’s. According to (Villemeur, 1992a), the first software reliability models were developed independently and simultaneously in 1971 by Jelinski and Moranda, and Shooman.

Since then a number of models have been proposed. Currently, there are two broad categories of software reliability modeling. These are:

1. Reliability growth models – classified as black-box approaches

#### 3.9.1 Reliability Growth Models

The main characteristic of this class of models is that they treat software as a black box having certain sets of inputs and outputs. The internal structure of the software is not taken into consideration. The software failure process is modeled as it evolves from version to version by fixing its initial bugs.

In these models (Villemeur, 1992a), a program is considered as a mapping from input space ($I$) to output space ($O$). If a program contains bugs, then it will be revealed when encountering input data from a certain input space denoted by $I_B$ causing a software failure. Two sources of uncertainty are identified in this process which are:

- Input data-related uncertainties as they cannot exhaustively be anticipated or tested
  A succession of spaces $I_B(1), I_B(2), \ldots, I_B(n), \ldots$, of various sizes correspond to them.
• Program-related uncertainties: the life of the software can be regarded as a succession of programs \( P(1), P(2), \ldots, P(n), \ldots \), each of them different from the previous one because of the corrections introduced.

The modeling process makes distinction between the behavior, up to the first failure when only the first uncertainty source applies, and the behavior during the life cycle where the uncertainty sources stem from inputs as well as programs.

Numerous models have been proposed over the years, involving the following general schema (Villemeur, 1992a).

• Reviewing all failures of software while being run or tested. Data collected includes:
  o Number of bugs detected;
  o Intervals of time between detected bugs
  o Cumulative duration of program execution
  o Time to correct each bug
  o Test characteristics
  o Data from earlier versions

• Predication of a reliability measure (such as failure rate) that is related to the random variable \( T_{N+1} \); this characteristic is most often written \( f(t, N_0, N, a_i) \) where \( t, N_0, N, \) and \( a_i \) denote respectively the time, the number of initial bugs, the number of failures observed and a specific parameter of the model.

The models also make two fundamental assumptions about features of software:

• Software contains a given number of bugs throughout its existence; the initial number of bugs is considered, for example, before the testing period or once the software’s ability to function is acknowledged;

• The bug number can only decrease as time passes, unless other bugs are introduced due to defective debugging.

There are various categories of these models that give rise to (Villemeur, 1992a):

1. ‘Perfect debugging’ models - models that assume that debugging (or removal of bugs) does not introduce additional bugs
2. ‘Imperfect debugging’ models – these models deviate from the above in their assumption in that debugging is imperfect
3. ‘Random debugging’ models – models that account for the randomness of debugging as it removes bugs as well as introduces new ones
4. ‘Bugs with different occurrence rates’ models – these are different from the first type by assuming that bugs have different probabilities of causing software failure
5. Parametric models – many other models in which reliability modeling is more parametric than explanatory. Poisson distributions are generally used to describe the number of faults occurring in a time interval.

Mainly differing on these factors, various reliability growth models have been developed such as: Poisson model (time-related), Musa Model, The Jelinski_Moranda and Schick-Wolverton Model, Goel and Okumoto Imperfect Debugging Model, Littlewood Models, Non-Homogenous Posion Process (NHPP) Models, Laprie’s Model, etc. Since, the focus on this research is on structure based reliability models, the details of these well-known models will not be covered further.

### 3.9.2 Architecture Based Reliability Modeling

Reliability growth models are also called black box approaches since they do not consider the structure of the software for reliability estimation. Their estimation is normally based on failure information mainly and the debugging process to some extent. For complex software systems that consist of a number of components, these models do not take advantage of component based testing and reliability information.

Architecture based models have been proposed to overcome the limitations of reliability growth models in taking advantage of software structure for reliability assessment. Rather than looking at software as a monolithic whole, architecture based models consider software to be composed of various components, whose reliability estimates are assumed to be known. The models attempt to derive software reliability by taking into consideration the structure of the software and the reliability of the components. Alternatively, architecture based models may also be used to allocate component reliability to achieve a given software system reliability figure.
Various architecture based reliability models have been proposed over the years. Katrina Goseva-Popstojanova and K.S. Trivedi review paper (Katerina Goseva-Popstojanova, 2001) details and the general goal of architecture based models as follows.

- Developing techniques to analyze the reliability and performance applications built from reusable and COTS (commercial off the shelf) software components
- Understanding how the system reliability/performance depends on its component reliabilities/performance and their interactions
- Studying the sensitivity of the application reliability to reliabilities of components and interfaces
- Guiding the process of identifying critical components and interfaces
- Developing techniques for quantitative analysis that are applicable throughout the software life cycle

The paper also discusses the commonalities and differences among the various proposals. According to this paper, the various models (about 17 of them) are based on the following common assumptions:

- Module identification: A module is conceived as a logically independent component of the system that performs a well-defined function. A module can thus be designed, implemented and tested independently.
- Architecture of the software: The software behavior with respect to the manner in which the different modules of software interact is defined through the software architecture. Interaction occurs only by execution control transfer. In the case of sequential software, at each instant, control lies in one and only one of the modules. Software architecture may also include the information about the execution time of each module.
- Failure behavior: in the next step, the failure behavior is defined and associated with the software architecture. Failure can happen during an execution period of any module or during the control transfer between two modules. The failure behavior of the modules, and of the interfaces between the modules can be specified in terms of their reliabilities or failure rates (constant or time-dependent)
3.9.2.1 Classification of Models
According to the aforementioned review paper, the various models may be classified into three different types depending on the method used to combine failure behavior with software architecture. These are:

- State-based approach,
- Path-based approach, and
- Additive approach

3.9.2.1.1 State-based models
State-based models use the control flow graph among the components of a software system to represent the architecture of the system. The basic assumption is that the transfer of control has a Markov property that may be stated as: given the knowledge of the module in control at any given time, the future behavior of the system is conditionally independent of its past behavior.

Various models have been proposed over the years that may vary, depending on the type of software to be modeled, as terminating and non-terminating systems – and/or the assumptions taken. These models represent the architecture of the software as:

- Discrete Time Markov Chain (DTMC),
- Continuous Time Markov Chain (CTMC),

The models are also further classified into two:

- Absorbing
- Irreducible

Absorbing models are used to model terminating applications, where the application operates on demand and terminates execution once the job is finished, or when failure is encountered. Irreducible models are, on the other hand more suited to continuously operating software applications such as in real-time systems, where it is either difficult to determine what constitutes a run or there may be a very large number of such runs if it is assumed that each cycle consists of a run.
The reference also classifies state-based models into two, based on how the failure behavior is combined with the architecture. The two classes are:

- **Composite**
- **Hierarchical**

The composite methods combine the architecture of the software with the failure behavior and the resulting model is solved to predict the reliability of the application. In the hierarchical model, on the other hand, the architectural model is first solved and the failure behavior is superimposed on the solution of the architectural model to obtain the predicted system reliability.

The following table summarizes the various models that have been covered by the referenced review paper.
Table 3-2: State based models

<table>
<thead>
<tr>
<th>No.</th>
<th>Model</th>
<th>Underlying method</th>
<th>Architecture representation</th>
<th>Method of Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Littlewood model</td>
<td>Software architecture of continuously operating system modeled by an irreducible SMP.</td>
<td>Transfer of control between modules is described by the probability ( q_{ij} = \text{Pr}{\text{program transits from module } i \text{ to module } j} )</td>
<td>The asymptotic analysis of this model led to Poisson process with parameter ( \lambda = \sum_{i} \pi_{i} \left( \lambda_{i} + \sum_{j \neq i} q_{ij} v_{ij} \right) ) where ( \pi = [\pi_{i}] ) is the equilibrium vector of the irreducible CTMC with the transition rate matrix ( Q = [q_{ij}] ), i.e. ( \pi Q = 0 ). The term in the parentheses ( \Lambda_{i} = \lambda_{i} + \sum_{j \neq i} q_{ij} v_{ij} ) can be interpreted as a failure rate of component ( i ) that includes the internal component failure rate ( \lambda_{i} ) and the failure due to the failures that occur during interactions with other components. Since, the steady state probability ( \pi_{i} ) represents the average proportion of time spent in state ( i ) in the absence of failure, ( \pi_{i} \Lambda_{i} ) can be considered as the equivalent failure rate of component ( i ) that takes into account the component utilization.</td>
</tr>
<tr>
<td>2.</td>
<td>Cheung model</td>
<td>Program flow graph of a terminating application has a single entry and single exit node, and transfer of control described by an absorbing DTMC with the</td>
<td>Two absorbing states ( C ) (correct) and ( F ) (failure) are added and the transition probability matrix is modified in such a way that the original transition probability ( p_{ij} ) between modules ( i ) and ( j ) is modified into ( R p_{ij} ). The failure of a module ( i ) is considered by creating a directed edge to failure state ( F ) with transition probability ( (1-R_{i}) ).</td>
<td>The absorbing DTMC can provide the average number of times each component is executed, denoted by ( V_{i} ), as a measure of utilization of a component. Then, ( R_{i}^{V_{i}} ) can be considered as the equivalent reliability of component ( i ) that takes into account component utilization. Thus, the system reliability becomes ( R = \prod_{i=1}^{n} R_{i}^{V_{i}} ), which is the hierarchical approach to reliability estimation.</td>
</tr>
<tr>
<td></td>
<td>System description</td>
<td>Parameters</td>
<td>Model details</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>------------------</td>
<td>------------</td>
<td>--------------</td>
<td></td>
</tr>
</tbody>
</table>
| 3. | Laprie model  
System composed of n components with transfer of control between components described by a CTMC  
Parameters are mean execution time of a component i and the probability \( p_{ij} \) that component j is executed after component i given that no failure occurred during the execution of component i. |  
Laprie’s model is a special case of Littlewood’s which considers only component failure where \( v_{ij} = 0 \). i.e., \( \lambda_{s} = \sum_{i} \pi_{i} \tilde{\lambda}_{i} \) |  |
| 4. | Kubat model  
Transition between modules follows a DTMC. Architecture model for each task becomes an SMP  
The probability \( q_i(k) \) task k will first call module i and with the probability \( p_{ij}(k) \) task k will call module j after executing in module i. |  
The components reliabilities are estimated as the probability that no failure occurs during the execution time of component i  
\[ R_i = \int_0^\infty e^{-\lambda t} g_i(t) dt \] where \( g \) is the p.d.f. for the sojourn time during the visit  
\( R_i \) is the component reliability and the embedded DTMC of the SMP that describes the software architecture provides the average number of times \( V_i \) each component is executed. The hierarchical treatment of this model thus reduces to that of Cheung’s model. |  |
| 5. | Gokhal et al. model  
Terminating application by an absorbing DTMC  
Transition probability obtained \( p_{ij} \) from trace data during testing |  
Hierarchical approach that estimates the component reliabilities considering time-dependent failure rates and the utilization of the component through the cumulative expected time spent in the component per execution \( V_i t_i \) |  |
<table>
<thead>
<tr>
<th>6.</th>
<th>Ledoux model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irreducible CTMC with transition rates $q_{ij}$</td>
</tr>
<tr>
<td></td>
<td>Generator matrix defines the following transition rates: from $c_i$ to $c_j$ with no failure, from $c_i$ to $c_j$ with a secondary failure, from $c_i$ to $c_j$ with a primary failure, from recovery state $i$ to recovery state $j$, and from recovery state $i$ to $c_j$.</td>
</tr>
<tr>
<td></td>
<td>This is an extension of Littlewood model to include the way failure processes affect the execution and to deal with the delays in recovering an operational state. The measures that are evaluated numerically by this model include: distribution function of the number of failures in a fixed mission, time to the first failure, point availability and failure intensity function.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7.</th>
<th>Gokhale et al. reliability simulation approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Discrete event simulation</td>
</tr>
<tr>
<td></td>
<td>Case study of a terminating application considers time-dependent failure rate of each component, finite debugging time, and fault-tolerant configurations for some of the component</td>
</tr>
<tr>
<td></td>
<td>Study of influence of different factors on the failure behavior of the application using discrete event simulation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8.</th>
<th>Yuan-Shun Model (Yuan-Shun et al., 2005),</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Marokov renewal process model</td>
</tr>
<tr>
<td></td>
<td>This model considers multiple failure states of software (as opposed to multiple software components) which are represented by Markov renewal process.</td>
</tr>
<tr>
<td></td>
<td>The case of software reliability prediction based on state-based models but taking into account failure correlation among successive software runs. They considered cases in which software exhibits noticeable degradations of performance besides being in failure or success state. Thus, in cases where there can be $n$ different types of failures, the total number of possible states for every run can be $n+1$ where the additional one state is for the successful state. To handle such cases, the authors proposed a Markov renewal model that allows the software to be in one of its multiple states.</td>
</tr>
</tbody>
</table>
### 3.9.2.1.2 Path-based models

Models of this class use information on the possible execution paths obtained either through testing or algorithmically to represent system architecture and combine that with failure behavior. The underlying idea is that modules encountered along each execution path are considered to be in series connections, i.e., the reliability per demand of the path is obtained as a product of component reliabilities. Then, the reliability of the software is obtained by as a weighted mean of the path reliabilities. The models cited under this category are given below in Table 3-3.

**Table 3-3: Path-based models**

<table>
<thead>
<tr>
<th>No.</th>
<th>Model</th>
<th>Solution method</th>
<th>Architecture representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Shooman model</td>
<td>Given the frequencies $f_i$ with which path $i$ runs and the probability of path $i$ on each run, denoted by $q_i$. System probability, $q_0$, of failure on any test run is given by: $q_0 = \sum_{i=1}^{m} f_i q_i$.</td>
<td>Knowledge of different paths and the frequencies with which each path runs</td>
</tr>
<tr>
<td>2.</td>
<td>Krishnamurthy and Mathur</td>
<td>Given component reliabilities as $R_m$, and the component trace, $tc$, of a program $P$, denoted by $M(P,tc)$, then the reliability of a path $P$ traversed when $tc$ is executed can be obtained as: $R_{tc} = \prod_{m \in M(P,tc)} R_m$. The reliability estimate of a program with respect to a test set $TS$ is $R = \frac{\sum_{tc \in TS} R_{tc}}{</td>
<td>TS</td>
</tr>
<tr>
<td>3.</td>
<td>Yacoub et al. model</td>
<td>Scenarios, tree-traversal algorithm where the breadth expansion represents logical “OR” paths and hence translated as the summation of reliabilities weighted by the transition probability along each path and the depth as logical “AND” hence translated to multiplication of reliabilities.</td>
<td>Using scenarios, a probabilistic model named component dependency graph (CDG) is constructed. A node represents component execution with an average execution time and an edge represents the transition probability.</td>
</tr>
</tbody>
</table>
3.9.2.1.3 Additive models

The additive models are based on software reliability growth models for components. By assuming that components’ reliabilities may be modeled by NHPP, then the system failure intensity can be simply expressed as a sum of component failure intensities. The models do not thus consider the architecture of the software explicitly. Table 3-4 contains models under this category.

Table 3-4: Additive models

<table>
<thead>
<tr>
<th>No.</th>
<th>Model</th>
<th>Underlying method</th>
<th>Architecture representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Xie and Wohlin Model</td>
<td>n-components developed and tested independently, NHPP</td>
<td>Known failure rate of components – leads to system failure rate as sum of component failure rates</td>
</tr>
<tr>
<td>2.</td>
<td>Everett</td>
<td>EET (extended-execution time)</td>
<td>When the underlying EET models for components are NHPP models, cumulative number of failure and failure intensity functions for the superposition of such models</td>
</tr>
</tbody>
</table>

Despite the differences in the assumptions, approaches, the various models proposed over the years have same mathematical commonalities (Katerina Goseva-Popstojanova, 2001), (Swapna, 2007). They all consider software components to fail independent and that a component failure will ultimately lead to a system failure. The main difference among the models is on how to account the relative contribution of failure from the components, while even in here, they all assume more frequently used components are given higher reliability importance.

In addition to the above list of architecture based prediction models, a number of models have also been proposed some of which based on Bayesian probability theory as in the case of (Singh et al., 2001).

3.9.3 Reliability Modeling for Fault-tolerant Software Systems

The architecture-based reliability models covered in the preceding section generally consider all software components to be in series connection. Some model developers such as (Xie and Wohlin, 1995) even claim “a fault-tolerant system can be considered as a
series system from a reliability point of view if any subsystem failure is treated as a system problem, although from a functionality point of view, the system is a parallel one.”

One potential reason for making such simplification assumptions may be because of the difficulty in reliability modeling of software redundancy. As covered previously, the proposal for substantial improvement of software reliability, through the development of diverse programs (multiversion software development), was put into question based on results from specific software trials.

This was because, the independence of failure behavior assumption could not hold true for programs that were developed independently. Specifically, the reliability of the multiversion software for some specific ‘experiments’ was not found to be as good as the estimate that could be obtained based on independence assumption.

As a result, despite the general agreement that the use of diverse versions of software would result in improved reliability over single version software, it is not clear how much the improvement would be. The difficulty of modeling N-version systems is discussed in details by Littlewood et. al. (Littlewood et al., 2001).

In an earlier paper, (Bev Littlewood and Miller, 1989), the authors have discussed a conceptual model of coincident failure rate on multiversion software. They consider two sources of uncertainties in software behavior - one related to the selection of an input to a program while the other is related to program development (seen as a selection of a program from a set of possible programs that could be generated by diverse methodologies). They developed a model for finding the probability of simultaneous failure of two diverse programs on a randomly selected input by using a score function:

\[ v(\pi, x) = \begin{cases} 
1 & \text{if program } \pi \text{ fails on input } x \\
0 & \text{if program } \pi \text{ does not fail on input } x 
\end{cases} \]

where, the random variable \( v(\Pi, X) \), represents the performance of a random program on a random input.

**Equation 3-1**

and considering a key average performance measure as:
\[ \theta(x) = \sum_p \theta(v(\pi,x)S(\pi)) = E_s(v(\Pi,x)) , \]

**Equation 3-2**

which is the probability that a randomly chosen program fails for a particular input \( x \).

Since \( \theta \) may vary from one methodology of developing a program to another in addition to its variation from one input to another, the probability of simultaneous failure of two programs on a randomly chosen input is given as:

\[ P(\Pi_A \text{ fails on } X, \Pi_B \text{ fails on } X) = Cov(\theta_A, \theta_B) + P(\Pi_A \text{ fails on } X).P(\Pi_B \text{ fails on } X) \]

**Equation 3-3**

This model shows that, at least theoretically, it is even possible for diverse programs to (Bev Littlewood and Miller, 1989) “do better than the (unattainable) goal of independent performance of versions in the single methodology case. This desirable state of affairs occurs when \( Cov(\theta_A, \theta_B) < 0 \)”. If the covariance is zero, one can have the case of independence and when it is positive one will have worse performance than independent systems.

The model only gives what is theoretically possible from the use of forced diversity but does not provide any guarantee that a specific approach would lead to a certain desired correlation property. In fact, according to the authors (Bev Littlewood and Miller, 1989), “it seems clear that it will never be the case that methodologies can be specially created to have desirable correlation properties”

A few other models have been developed to estimate the reliability of N-version systems. These models include those reported in papers (Bev Littlewood Peter et al., 2000), (Wattanapongskorn and Coit, 2007), (Zafiropoulos and Dialynas, 2004), (Ege et al., 2001) and (Ping et al., 2008). However, due to the difficulty of obtaining reliable data to support various models, there are no universally accepted models for estimating the failure rate of redundant software versions.

Besides N-version programming, there are various ways of achieving software fault tolerance including multilevel restoration systems. One model that considers the possibility of multiple level restorations is provided by Vilkomir et. al. (Vilkomir et al., 2005). The authors introduced a new simplified analytical approach to availability
evaluation of hardware/software systems with several recovery procedures based on a new ‘segregated failures’ model. The model is used to estimate the system down time as the sum of down time for various failures types that are categorized according to the lowest recovery level from which faults can be successfully recovered. The model inputs are system failure rate, restoration rates of the various recovery procedures, and the coverage factors of the recovery procedures. The model, in addition to allowing for manual assessment of availability of a complex system, it helps to apply sensitivity analysis and to find the weakest links of a system reliability architecture and recovery strategy.

3.10 Assessment of Existing Software Reliability Models
Black box models do not take information about the structure of software systems are not effective for predicting reliability of complex software systems that consist of a number of components and their interactions.

Most of the architecture based software reliability models proposed over the years consider software systems as a series connection of components. Their major difference lies in their attempts on how to account for the possible variation of reliability contributions from the different components. The common assumption taken by almost all models is that most frequently used components are also the most important for the reliability structure (with incorrect implications such as: the most frequently invoked ‘Log’ event makes the ‘Logging/Tracing’ component included for capturing maintenance information much more important than a ‘Security’ component that handles the least frequently invoked ‘Login’ event).

The precise interdependency among components, their failure mode and effect and possibilities of fault tolerance are not taken into account. As a result, they attempt to model software architecture by the ‘transfer of control’ among components based upon which the relative contribution of the components is estimated. Katerina Goseva-Popstojanova writes, “the software behavior with respect to the manner in which the different modules of software interact is defined through the software architecture. Interaction occurs only by execution control transfer. In the case of sequential software, at each instant, control lies in one and only one of the modules. Software architecture
may also include the information about the execution time of each module.” (Katerina Goseva-Popstojanova, 2001)

Essentially, this may be considered as the case of estimation of what classical reliability theory calls the duty cycle (Bazovsky, 1961) of components. This, on its own, is not a good measure of reliability contribution, since some of the most frequently used components may have faults that can be tolerated, while other rarely used components may have critical faults that lead to system failure. Based on an empirical reliability assessment of a case study, Katerina Goseva-Popstojanova already point out the fact that “relationships between faults and failures are complex and almost unexplored in the literature” (Katerina Goseva-Popstojanova et al., 2005).

The models, while mathematically plausible, usually lack the detailed behavioral information about the structure of the software system, the various components in the system, the interactions among the components, the causes of failure for components and the role played by each towards the system reliability. This leads to gaps between the assumptions many architecture-based reliability models may take and the possible software architecture and behavior that may be achieved.

Reliability improvement concepts, such as redundancy, protection, and fault-tolerance are not usually included in these models. Also, most models assume statistical independence among component failures, which is difficult to achieve in software design, and nearly all approaches avoid the modeling of interaction failure causes except one (Kishor S. Trivedi and Fricks, 2003) that looks into failure caused due to missing of deadlines, but without using structural information.

3.11 Chapter Summary
This chapter has thoroughly discussed the three areas that are related to design for reliability prediction. These areas are: software design and construction (S&C), specification and description (S&D) and structure based reliability prediction (SRP). Many decades of work have been spent in each of these areas and thousands of papers have been published that makes complete coverage of each them in one chapter of a thesis infeasible. However, the main approaches, goals, limitations and the state-of-the-art where relevant are discussed thoroughly.
In software design and construction, *design for change*, is a principle that withstood the passage of time and technology spanning from its early inception 1970’s to the most recent architecture focused designs and aspect-oriented programming paradigms. Targeting software reliability improvement, various fault-tolerant designs have appeared which were covered in two categories: software redundancy and undesired event/exception handling.

In reliability prediction, two major classes have evolved: reliability growth models which treat software as monolithic whole and architecture based models which consider software structure. However, there is significant gap between these models and the possible structural relations that designers can come up with.

From the three areas, precise specification and description of software design seems to be the least researched area. Various approaches proposed over the years have not penetrated software development practice. Software engineering still depends on informal and semi-formal means of communicating designs, which are known to be ambiguous, inconsistent and incomplete. Practical documentation methods that connect today’s software design approach to structural reliability prediction models are needed.
4. Analysis of the Reliability Prediction Problem

4.1 Introduction
The main objective of this chapter is to analyze the problem of software reliability prediction. It assesses the main characteristics of software failure, the approaches to mitigating failure and the challenges of reliability prediction.

4.2 Main Characteristics of Software Failure
Software fails for various reasons. The initial assumption of structured programming where correct software could be produced by structuring correct programs was challenged by the Parnas et. al. paper on “Response to Undesired Events in Software Systems.” (Parnas and Wurges, 1976). In this paper, the authors write correctness assumptions could be invalid for the following reasons:

- Even the best “structured programmer” occasionally err
- The machines which we use occasionally fail and may cause a program to fail (either directly or by causing a change in code or data)
- In practice, programs are changed and errors appear which had not appeared before
- Incorrect or inconsistent data may be supplied to the system

Three decades have passed since the observation was made and various proposals for responding to undesired events and exception handling are suggested. During this time, technological progress has been made in design methods, techniques and tools. Yet “unexpected” software failure is still common, causing substantial loss.

Hatton describes this prevalence of software unreliability, in spite of the technological developments, as follows: “Even so, many techniques that supposedly promote the goal of improved reliability have come and gone. However, these all too often arose in the complete absence of measurement, so it is far from obvious what helps, by how much, and why. In this regard, software development is much more akin to the fashion industry than an engineering industry. … Unfortunately software systems have been unusually resistant to such advances, and the defect density of software has remained about the same over the 15 years or so.” (Hatton, 1997).
In the same reference, the Hatton paper provides the following figures:

- “Three to six defects per KLOC represent high-quality software. It is rare to find a system below one defect per KLOC.”
- “Even when formal specifications techniques, 100% statement coverage, and concurrent testing plans in very high-integrity systems were used in an air-traffic control system, the resulting system still contained around 0.7 defects per KLOC”

### 4.2.1 Main Differences between Hardware and Software Failure

The main difference between hardware and software failure lies in the causes of failure. Although design errors can contribute to hardware failure, the main causes for hardware failure are usually related to physical deterioration of the hardware – or wear out. This is dependent on time where failure rate increases with age. This behavior has helped to estimate the lifetime of hardware systems and to provide a guarantee accordingly.

On the other hand, software fails almost exclusively due to design errors. Except for changes in the environment and/or the structure of software itself, through maintenance, software does not wear out. For software whose structure and environment has not changed, its state can remain the same for decades. The possibility of failure depends on the presence of design fault in the system and the occurrence of an input condition that triggers the design fault. This can happen at any time in its operational life.

There are many other differences between hardware and software failure, some of which may be given in Table 4-1 as adapted from (Wallace R. Blischke and Murthy, 2000):
Table 4-1: Comparison between Hardware and Software Failure (Wallace R. Blischke and Murthy, 2000)

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mostly caused by physical deterioration</td>
<td>Exclusively caused by design errors (errors in specification, design, coding, integration)</td>
</tr>
<tr>
<td>Mostly Failure Time Dependent (wear out, fatigue, etc.)</td>
<td>Mostly Failure Time Independent (no wear out, no stress)</td>
</tr>
<tr>
<td>Redundancy can be used to increase reliability</td>
<td>Redundancy does not necessarily lead to improvement</td>
</tr>
<tr>
<td>Can test “all” events (general physical principles, mathematical models apply)</td>
<td>The number of events is huge and events tend to be unique to the software. Also software has many more “paths” and “parts”.</td>
</tr>
<tr>
<td>Failures may be described by physical laws</td>
<td>Comparable laws do not exist</td>
</tr>
<tr>
<td>Small anomaly may lead to predictable failure or have little or no effect</td>
<td>One incorrect bit can lead to disaster</td>
</tr>
<tr>
<td>Interfaces are physical structures</td>
<td>Interfaces are conceptual</td>
</tr>
</tbody>
</table>

4.2.2 Software Failure Causes

Software may fail to function as required for a number of reasons. The different reasons, however, can be categorized into two main classes.

1. Failure Caused by Design Faults
2. Externally Induced Failures

4.2.2.1 Failures Caused By Design Faults

Most software failure is caused by design faults (also called software errors or bugs). These design faults may range from subtle, difficult and hard to uncover bugs, to simple and silly programming mistakes.

The site given in (Hower, 1996-2004 ) contains media reports of software failures and their consequences. Going through failure lists such as the one mentioned above and looking at similar others such as (O’Connor, 2002 pp 293) one can see that not all software faults that cause system failures, with huge consequences, are complex or subtle errors.
In fact, as noted by Knight (Knight, 2005), “many of the problems in safety-critical software arise from elementary mistakes.” Yu’s report (Yu, 1998) based on analysis of 600 faults on Lucent’s 5ESS switching system, which consists of several millions lines of source code, also confirms this view. According to his report, “defects included the use of un-initialized variables, misuse of break and continue statements, incorrect order of operand evaluation because of misunderstandings of operator precedence, incorrect loop boundaries, indexing outside arrays, truncating of values, misuse of pointers, and incorrect AND and OR tests.” Other reports (Lieberman and Fry, 2007), indicate a misplaced comma in a program to have caused a NASA space mission to fail.

Trivial coding errors seem to appear even when one uses mathematical proofs for ensuring the logical correctness of program design, as can be seen from Dijkstra’s report on the construction of the T.H.E. machine. “Having fears regarding the possibility of debugging we decided to be as careful as possible and – prevention is better than cure! – to try to prevent bugs from entering the constructions. This decision is at the bottom of the group’s main contribution to the art of system design. We have found it is possible to design a refined multiprogramming system in such a way that its logical soundness can be proved a priori and that its implementation admits exhaustive testing. The only errors that showed up during testing were trivial coding errors (occurring with a density of only one error per 500 instructions), each of them located within 10 minutes (classical) inspection at the machine and each of them correspondingly easy to remedy.” (Dijkstra, 1968).

The appearance of failure due to faults may vary based on the operational history of its parts. Hecht et. al, reports that “rarely executed code has a much higher failure rate (expressed in execution time) than frequently executed code during the early operational period.” (Hecht and Crane, 1994 ). Similar cases are also reported by Jung et. al. (Jung and Ye, 2004), where failure to handle exceptions which basically indicate what is thought to be rare cases become major causes of system crashes. Based on their investigation, they claim that “two-thirds of systems crashes are caused by failure to handle exceptions.” (Jung and Ye, 2004).
The case of unhandled exceptions being a cause of a major accident can also be seen from the inquiry commission report of the Arian 5 disaster (Lions, 1996). The report states “in the failure scenario, the primary technical causes are the Operand Error when converting the horizontal bias variable BH, and the lack of protection of this conversion which caused the SRI computer to stop.”

Software faults may be introduced at various life cycles of the software development processes. After analyzing over 600 faults on Lucent’s 5ESS switching system, which consists of several million lines of source code, Yu produced the distribution of faults over the development phases shown in Table 4-2.

Table 4-2: Faults introduced during different life-cycle phases (Yu, 1998).

<table>
<thead>
<tr>
<th>Phase</th>
<th>Total (%) of fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>12.2</td>
</tr>
<tr>
<td>Data design</td>
<td>6.6</td>
</tr>
<tr>
<td>High-Level design</td>
<td>18.5</td>
</tr>
<tr>
<td>Low-level design</td>
<td>15.3</td>
</tr>
<tr>
<td>Coding</td>
<td>47.4</td>
</tr>
</tbody>
</table>

This data, though specific to a particular application type, shows that nearly half of the software errors are introduced during coding. Such figures however may depend on various factors and thus may vary from system to system.

Some writers even report the majority of software defects stem from requirement or specification flaws rather than coding. For instance, a study cited in (Villemeur, 1992a p 647) puts 2/3 of software defects stemming from wrong specification while 1/3 stem from incorrect coding. The same reference also provides the Table 4-3 given below which gives percentage of defects per category based on three studies.
Table 4-3: Percentage of defects per category (Villemeur, 1992a, p 646)

<table>
<thead>
<tr>
<th>Categories</th>
<th>Compiler (1st case)</th>
<th>Real-time software (2nd case)</th>
<th>Real-time software (3rd case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>6</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Logic</td>
<td>38</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>Input/Output</td>
<td>2</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>Data manipulation</td>
<td>15</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Interface</td>
<td>13</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>Data definition</td>
<td>19</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Data base</td>
<td>1</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Others</td>
<td>-</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

When discussing about software safety issues, Nancy Leveson (Nancy, 2002) writes “most errors found in operational software can be traced to requirements flaws, particularly incompleteness. In addition, nearly all the serious accidents in which software has been involved in the past 20 years can be traced to requirements flaws, not coding errors”.

In a modular system, where software is composed of many components, failure may occur during execution of a specific component due to design faults that are introduced when the component is implemented or during interaction due to faults introduced when deciding the structure of the software.

“The Ariane 5 and Mars Polar Lander losses are examples of system accidents. In both of these accidents, the components did not fail in terms of not satisfying their specified requirements. The individual components operated exactly the way the designers had planned—the problems arose in the unplanned or misunderstood effects of these component behaviors on the system as a whole, that is, errors in the system design rather than the component design, including errors in allocating and tracing the system functions to the individual components.” (Nancy, 2002)
Generally, creating a reliable system by composing many independently developed components that might have worked well independently at component level is a non-trivial problem. Garlan et. al., reported their experience on the challenges of composing a system out of independently developed components, and used the phrase “architectural mismatch” (Garlan et al., 1995) to describe the problem. They attempted to build an Aesop system by composing four existing components:

- An object oriented database: OBST, public domain OODB
- A toolkit for constructing GUI- InterViews and Unidaw
- An event-based tool integration mechanism: Softtbench, a commercial event-broadcast mechanism
- A RPC mechanism: March RPC Interface Generator

Their initial assumption was just to put the subsystems together, a task considerably thought to be simplified by the fact they were all written in either C++ or C, had all been used in many projects, and their source codes were available. However, they were able to get their first Aesop prototype where the pieces started working together after two years of considerable taking approximately 5 person-years of effort. Garlan et al writes (Garlan et al., 1995) “But even then, the size of the system was huge (although we contributed a relatively small proportion of our own code), the performance was sluggish, and many parts of the system were difficult to maintain without detailed, low-level understanding of the implementations.”

Garlan et. al, basic conclusion is “that many of the hardest problems are best understood as architectural mismatch problems. Each component makes assumptions about the structure of the environment in which it is to operate. Most, if not all of these assumptions are implicit, and many of them are in conflict with each other”.

Architectural mismatch problem persists to exist to this date (Garlan et al., 2009, Bierhoff et al., 2007, Cot{\`e}). Garlan et. al.’s experience is a confirmation of Parnas’s incite on the influence of structure for software reliability that was written as early as 1975 (Parnas, 1975b). In that paper, Parnas used the word structure “to indicate the way that a software system is divided into modules and the assumptions that the various modules make about each other.” This implies that, unless the assumptions the various
components make about each other is explicitly stated and holds true all the time, there may not be any guarantee to produce a ‘correct’ system by composing ‘correct’ components. Even if one manages to build the system, it may lack many desirable qualities. As Parnas puts it, “even ’correct’ (in the sense of mathematically verified) software can be quite unreliable”.

4.2.2.2 Externally Induced Failures
Software normally makes one component of a complex system. It may thus fail due to undesired events from the environment. These includes failure or depletion of resources (processors, memory areas, storage devices, communication networks, and various devices) it makes use of, incorrect or inconsistent inputs/events by the users, interference from other software systems that may compete for limited resources for their normal operation, or that may be distributed to deliberately attack the normal functioning of the system, and other man-made and natural disasters.

Every software system normally assumes some type of environment, although the assumptions are usually not explicitly described/or documented. Sometimes, these assumptions may even have conflicting elements. The environment, unless it is a dedicated one for the specific software system as in embedded systems, is also a dynamic one changing because of the execution of the various software systems it supports. This may cause the environment to be significantly different from that assumed by a specific software system during its construction. As a result, even mathematically ‘correct’ software can fail due to indirect interactions with other systems that influence its environment if the proof of correctness does not take into consideration the effect of other systems in the software environment.

Even in the absence of interferences from other software systems, failure due to resource depletion could arise if parts of the problem or solution provided requires much more resources than what can be availed.

Other causes of environmentally induced failures include deliberate attacks on the software or resources it uses, and human errors.
4.3 Main Approaches in Mitigating Software Failure

Over the years, various fault-mitigating strategies have been developed. These approaches can be broadly classified into two: fault-avoidance and fault-tolerance. System solutions may normally employ a combination of these strategies.

4.3.1 Fault-Avoidance

Fault-avoidance is basically part of all software development approaches. Nobody intentionally introduces software faults into the system he/she produces. Under normal circumstances, one would expect a developer/designer to make every effort under his/her capacity to avoid faults from the software system. However, since “to err is human”, avoiding fault is very difficult if not impossible.

Therefore, various methods have been developed targeted at avoiding faults as much as possible, among which include:

- Software testing and Debugging
- Formal program verification
- Software inspection

All of these activities assume the existence of a software requirements specification.

4.3.1.1 Testing and Debugging

Software testing is the execution of software in a real or simulated environment by providing various inputs (or test sets) and comparing the response with the expected result. It is assumed that the acceptable behavior of the software being tested is known either from a technical requirements specification or from an ‘oracle’.

Testing can be conducted for various objectives that include, uncovering of bugs, demonstration of functionalities, assessment of qualities such as reliability and performance, etc. As a means of fault avoidance, testing is conducted with the intent of identifying bugs before the release of the product so that they will be corrected.

Testing is the most popular strategy in the industry as it is normally taken as a major phase in almost all software development life cycle models. Testing is also conducted during coding, usually as part of debugging.
Unfortunately, testing is a sampling process that can never guarantee the correctness of a software product. Its limitation as a fault-avoidance strategy is succinctly put by Dijkstra’s highly-quoted statement: “program testing can be used to show the presence of bugs, but never to show their absence!” (Dijkstra, 1976).

There are many reasons for the limitations of testing, among which include:

- In any realistic software system, one can only take a sample of the possible inputs to a system and apply testing. No system can be exhaustively tested and one cannot cover all possible cases.
- Some undesired events, such as deadlocks, livelocks, resource depletions and race conditions, that occur only because of concurrent action of many processes can be very difficult to create in testing environment.
- Errors in specifications – testing is normally conducted to check whether the implementation of program meets its specification or not. If the specification is itself erroneous which cannot be corrected based on the program behavior, the program will reflect the same error present in the specification.
- Incomplete specifications and implementations – some failure may occur due to incompleteness of requirements. In fact, as seen earlier, incompleteness in requirements is reported to account most errors, where “nearly all the serious accidents in which software has been involved in the past 20 years can be traced to requirements flaws, not coding errors” (Nancy, 2002). This shows a major limitation of testing as it is mainly aimed at uncovering bugs in codes, products can pass testing without any failure but can still be unreliable due to incompleteness of requirements.
- Redundant and fault-tolerant designs can hide failures and limit the effectiveness of testing as a means of detecting faults.

“Program testing can be used to show the presence of bugs, but never to show their absence.” (Dijkstra, 1976)

4.3.1.2 Formal Program Verification
Formal program verification is the process of proving programs to be correct. It has been proposed as an alternative means of fault avoidance that will overcome the limitations of
testing. In fact, the main motivation behind the restriction of structured constructs was to be able to prove programs correct.

The main idea behind formal program verification is that, given the specification of a contemplated program by a formula $\Phi$, and a description of the implementation of the program $C$, by a formula $\Phi_c$ which precisely captures the behavior of $C$, then it is believed that “$C$ implements the specification $\Phi$ if all the behaviors generated by $C$, are also allowed by $\Phi$. In logic, this amounts to the validity of the implication $\Phi_c \Rightarrow \Phi$.” The process of proving the validity of this implication is referred to as formal verification as it is claimed to establish the fact that $C$ correctly implements $\Phi$ (Wilk, 1990).

Recent advances in formal program verification include the use of modeling tools to show the existence or absence of some system properties, including possible occurrence of undesired events, such as deadlocks.

Over three decades of research effort has been spent on the idea of proving programs correct. However, it has not yet come to be a major industrial mechanism of fault avoidance. Various practical limitations, starting from the problem of verifying the specification (ensuring their correctness), verifying the verifier (in case of automatic verification), complications of reality being intractable, to people’s confidence on proof, have hindered the practicality of program verification for fault avoidance.

“Beware of bugs in the above code; I have only proved it correct, not tried it." Donald K. Knuth, (Donald, 1977)

4.3.1.3 Software inspection,
Software inspection is another mechanism of fault-avoidance, which follows a systematic review of program code by experts in order to identify bugs. Proper inspection, as opposed to “group reading experience of programs from top to bottom”, may be time-consuming and requires the production of detailed documentation (Hoffman and Weiss, 2001 p 140). This documentation has to be precise and formal to properly investigate the system.
Examples of safety critical software systems that have been inspected based on mathematical documentation include software system in a nuclear generation station that is still in operation in Darlington, Ontario, Canada (Parnas, 1994).

Unfortunately, the main output of this process as well is increased confidence on the system, but not 100% assurance of fault avoidance.

All the methods discussed above are fault-avoiding efforts. They are not competing but rather complementary approaches. At the current level, none of them can guarantee complete avoidance of faults unless the systems under investigation have small size of code and possible execution states that can exhaustively be assessed with the methods.

4.3.2 Fault-Tolerance

Fault-tolerance is a complementary strategy for mitigating faults as none of the fault avoidance methods can guarantee fault free software. Besides, as explained in an earlier section, software will have to deal with failure of the external environment which is the main source of non-determinism. Recognition of this fact led to the development of various fault-tolerance strategies, the main ones being: exception handling and redundancy, both of which were covered in section 3.6.

4.4 Major Challenges for Software Reliability Predictability

For the various reasons covered so far, perfect operation of software products of all functionalities at all times can be difficult to achieve. Therefore, the need for a mechanism of quantifying the probability of successful operation of complete or some partial set of operations by a software product arises. Basically, this is the objective of software reliability prediction. However, there are various challenges that make reliability prediction of software system hard. These challenges are discussed under four main categories: software complexity, software design, specification and documentation, and structural reliability predication.
4.4.1 Software Complexity

Complexity which is claimed to be an essential property (Brooks, 1987) of software systems is one main reason for making probabilistic assessment of software reliability hard. Among the reasons that make software complex include:

1. It is directly produced by humans, who can frequently err. Since software has to continue being designed and constructed by humans, ‘software bugs’ will continue to creep in.

2. Small change does not necessarily mean small effect. In some cases, a change of single bit in a million bits can completely change system behavior.

3. Since most requirements are not governed by physical or natural laws, but human laws, getting conflicting requirements on specific system properties is not uncommon.

4. The state space of even small software systems can be extremely large. Complex problems can have intractably large state spaces.

5. In most cases, software functions as one component of a complex system environment. It will have to interact with other software and hardware systems, besides human users. This makes its input space very large. Knowing or restricting the operational environment may not always be easy. For instance, the design of a flight navigation system that did not take into account the behavior of the system near the north pole, since no one thought of airplanes to cross that region, is reported to have caused serious accidents that resulted in loss of life.

6. In most cases, the software environment is rapidly changing. The hardware, software platforms, other third party software systems, as well as the system requirements themselves continue to change at a fast speed, invalidating assumptions that were thought to be true at construction of a software system and making what was once reliable, unreliable.

These properties pose challenges in making reliability prediction of software systems.

4.4.2 Software Design

The past three decades have shown major advances in software design and construction. A few fundamental principles, closely related, if at all different, known by such names as...
information hiding, abstraction, encapsulation, separation of concerns, loose-coupling have enabled people to produce large software systems mastering their complexity. When the principles get applied in a disciplined and systematic way supported by mathematical documentation, they can make software changeability easier.

Unfortunately, various factors in the real world scenario result in system structures that may not be as neat as one would have liked them in an ideal world. Complete information about requirements is rarely known upfront during design. Similarly, complete information about the environment, other elementary systems that a system under design is going to interact with may not be known. These and various other factors can compromise on the quality of the structure of software designs.

Software construction culture in the industry is also not helpful in improving the quality of software designs. Usually, emphasis is given to coding, resulting in systems that continue to grow in code size, counted in millions of lines of code, making the product very difficult to analyze and maintain. This, code is the ‘king’ or code is the ‘best documentation’ approach, is being further promoted by recently introduced process models, collectively known as ‘agile methods’, that seem to get popularity in the commercial software industry.

Early focus on coding, fast delivery of something that works or seems to work has serious implications on software structure, at least on the longer term. Modularization may be done arbitrarily, probably based on code size, or immediately requested requirements, without much research on changeability, separability, reliability, etc. Decisions on interfaces may not be well thought out to include possible future changes that may affect the product.

Such design and development practices are likely to result in design decisions that may not have proper structures and have no provision for fault-tolerance, redundancy, protection, and prioritization. Thus, one of the major factors of software reliability predictability is its design. Well structured software systems can be tractability modeled, no matter what complex problems they solve. On the other hand, poorly structured designs may not be analyzable by any mathematical model that is tractable and understandable and hence may not be trusted.
4.4.3 Architecture Based Reliability Modeling

Over the last couple of decades, there have been a number of proposals aimed at estimating the reliability of software systems to be obtained from the reliability of components and their interconnections. However, there is currently a wide gap between the possible structure of software systems that designers can come up with and the assumptions taken by most, if not all, architecture based reliability models proposed in the literature such as those reviewed in (Katerina Goseva-Popstojanova, 2001), (Swapna, 2007).

For structure based reliability models to be useful tools in software or computer system design, they should be able to reflect the exact structure of software/computer systems. This should include the modeling of redundancies, protections, fault-tolerance, and other types of dependencies as well as interaction failure concerns. This may not be applicable for any software that may not have a well defined structure but for those systems where the interdependencies among the various components in the system are known and precisely documented.

Even then, predication of reliability is hard as can be inferred from the results of Kappes et. al, (Kappes et al., 2000), who proved the absence of any algorithm that can enable one to calculate or even estimate the reliability of component-based systems. Using communicating finite state machine models, the authors write “there is no algorithm that can precisely or approximately compute the reliability of a network of communicating finite state machines given the reliabilities of its transitions or components. Although it may be feasible to determine the reliability of a specific system, our result indicates that the general problem of computing the reliability of an arbitrary software system is unsolvable…. The theoretical results imply that the process of computing the reliability of a component-based software system from reliabilities of elementary operations or components in that a system cannot be completely automated. Therefore, software reliability engineers will have to depend on heuristics, expertise, and creativity to arrive at methods that would obtain such reliability figures for their specialized class of software systems.” (Kappes et al., 2000).
Generally, computer system reliability prediction that includes complex interaction among software components, communication networks, and various devices is one of the most difficult reliability prediction problems. The interactions between the various system elements, including human beings that use the system is often complex. Many components are interdependent on each other in that failure in one component may propagate to a number of other components and cause failures to one or more services provided by a system. Identical software components that run on different, supposedly redundant, computing nodes may appear as common-mode failures for services and systems.

### 4.4.4 Mathematical Documentation

Reliability assessment requires a precise behavioral specification/description of the various components in the system and the interconnection of these components when providing specified services. It requires knowing the possible causes of component failures, their possible effects on other components, the operational environment of components and the presence or absence of redundancies.

Compared with other engineering disciplines, such as electrical or mechanical engineering, software engineering designs lack mathematical documents that embody the concepts of transfer functions and systems of equations that can precisely describe component as well as system behavior, without resorting to the actual product. Although, various ‘formal’ languages have been developed, their penetration of the industry is limited due to either the high learning curve they require by adding new notations and expressions, or due to their impracticality to scale to the complexity of real world applications.

In general, software reliability prediction is made much more difficult because of unknown or poorly understood and documented interactions between components. In the absence of precise description of designs that capture the exact dependencies among the various entities in software systems, it would not be surprising if reliability experts consider all the components in a system to appear in series connection, even if that may not be what the actual design is. It is generally hard to make distinctions between series and parallel connections by looking at program code.
4.5 Chapter Summary

Software can fail for a number of reasons, but mostly due to design errors. Although there are software failure mitigating approaches, their effectiveness is limited due to the inherent complexity of software systems. Design approaches can play significant role in making software reliability prediction easier or more reliable by making structural reliability modeling possible. But various factors, such as lack of complete understanding of requirements and operational environment can compromise the quality of designs, making system reliability prediction harder. Previous researches show that reliability assessment of component based software systems to be inherently hard that does not have an algorithmic solution. This difficulty of reliability prediction is further aggravated by the absence of precise but practical description of software systems that can be used to derive the reliability structure of software system.
5. Basic Concepts of Design for Reliability Predictability

5.1 Introduction:
This chapter contains the definitions of terms, basic concepts and mathematical foundations of design for reliability predictability. These are divided into two major sections: the first dealing with software design related aspects and the second discussing reliability concepts.

5.2 Basic Concepts in Software Design and Documentation
The construction of any complex software system requires the decomposition of the system into modules and their composition to create a system. Usually, the terms ‘module’ and ‘component’ are used interchangeability in referring them as units of decompositions and compositions. Although we may also use the two terms interchangeability in many cases, there are a few distinctions that may be made between the two. The most obvious one is in viewing a module as a unit of independent development and change while a component is a unit of distribution and deployment. This may be seen from the definitions given below.

A module is a “work assignment” given to a programmer or group of programmers (Parnas, 1972a). An information hiding module is a module that is characterized by the design decision (secret) it hides from all other modules in a system, making it a unit of change.

A software component may be defined as a collection of programs that is distributed as a unit for use in larger systems. A component may be part of a module or may include (parts of) several modules.

Modules or components need to connect and communicate with each other in order to perform a required input-output transformation. These connection mechanisms may be collectively named as connectors. A connector can thus be defined as a runtime entity or data structure at a specific location and time created by one or more modules through which modules communicate with each other and provide their specified functionality. Program call stacks, objects, variables and other data structures can all fall in this category.
The main goal of designing software systems is to provide some required input-output transformations, which are called services in this thesis. A service is a set of independently usable systems functions with distinctly identifiable input-output variables and specified sets of input-output mappings.

Services are realized (or provided) by using a set of components connected through their input-output variables, connectors. This author calls the network of components that are connected through connectors and behave as a unit a composite.

There may be more than one way of composing modules, and hence more than one composite to provide a specific service. Thus, a service may be defined as a set of input-output mapping functions provided by one or more composites.

The reliability of services and systems is highly dependent on the structuring of these building blocks; modules, components, connectors and composites, as well as the event and process structures that are related to the change of state variables of these entities.

The various entities identified to describe computer system behavior and their relationships must be specified and/or described precisely/mathematically in order to reveal the reliability structure of the software system. The use of functional documents for precisely specifying or describing deferent aspects of computer system has been recommended by Parnas et. al in (Parnas and Jan, 1995).

A mathematical document may be defined as a relation that maps from a domain set D to a range set R with the goal of characterizing some aspect of a system.

This definition basically makes a mathematical document or a document, a mathematical formula that has precise and non-ambiguous meaning. Such documents have been used in various safety-critical systems such as (Kathryn et al., 1978), (Mark et al., 2004) and (Heitmeyer and Jeffords, 2007).

A mathematical document can be used to write a specification or a description of a product.

1 Description: a description states properties of a product; it may include both incidental and required properties.(Parnas and Dragomiroiu, 2006)
2 **Specification:** is a description that states only required properties (Parnas and Dragomiroiu, 2006)

3 **Full specification:** is a specification that states all required properties (Parnas and Dragomiroiu, 2006)

There are two aspects of mathematical documents: content and representation.

### 5.2.1 Document Content
The content of each document is characterized by the function/relation it describes. This may vary from specification method to method as well as from document type to document type. For instance, a requirement or interface specification may consist of only the description/specification of externally accessible variables and any relevant relation. This may be restricted to one module. On the other hand, an interconnection document may refer to the connection relations among many modules. An internal design document may include internal data structures and effects. An implementation document may include program source codes, etc.

### 5.2.2 Document Representation:
The same content, i.e. mathematical formula or expression can be represented in various ways. The choice of representation depends on different factors—such as intended users whether it is for human beings or computers, purpose, whether it is needed for calculation or derivation of some desired properties or whether it is for review by experts, etc. Among the variety of representation techniques, the following have been used in this thesis.

#### 5.2.2.1 Mathematical Expressions
All documents are mathematical expressions and hence they can be written in text and formula form just like standard mathematics.

#### 5.2.2.2 Tabular Expressions
Tabular expressions are mathematical expressions represented in table format whose grid structure has precise mathematical meaning (Adam Balaban, 2007).
The main target of tabular expressions is for documents to be read by human beings. For computer representation and manipulation, they can be converted to mathematical expressions.

Probably, the most important source of software complexity comes due to the fact that software systems are constructed based on logical expressions. As a result, the mathematical expressions that would be produced by any specification method will consist of long logical expressions that can be hard to read or understand or prone to writing or reading error.

The use of tables can help to reduce such possible errors. They can be used to systematically check the completeness and consistency of specifications and descriptions. Tabular expressions have been found to be highly useful in various safety critical projects, starting from requirement documentation of A-7E avionics (Kathryn et al., 1978) to Nuclear Shut Down reactors (Lawford et al., 2001), as well as many other projects, such as those reported in (SERG, 1997), (Parnas, 1992), (Parnas, 1994), (Ying et al., 2004).

### 5.2.2.3 Matrices
Matrices can be used to define relation among various properties of computer systems. They are standard forms of solving system of equations. By using linear or logical algebra, they can be used to calculate or derive various system properties. Many system reliability assessment models, especially state-space models such as discrete, continuous and semi-Markov models also use matrices. Since this author’s work involves analysis of reliability structure, matrices are frequently used for representation as well as evaluation of system properties.

### 5.2.2.4 Graphs
A graph is a double $G=(V,E)$ where $V$ is the set of vertices and $E$ is the set of edges with each element $e_k$ is identified as an order pair $(v_i, v_j)$ of vertices. Graphs, in directed or undirected form, are another way of representing relations. Just like matrices, they can be used to calculate or evaluate system characteristics. The use of graphs for representing various aspects of software systems is common – starting from flow charts to control flow graphs, data flow graphs, activity diagrams, etc. Some of these graphs, however,
have properties that conflict with the information hiding principle as they tend to reveal implementation details or changeable aspects.

However, there are other graphs such as signal flow graphs that have a solid mathematical basis that can be used to represent as well as evaluate mathematical expressions. Such graphs will be used to study the dynamic behavior of systems including timing aspects as well as for reliability assessment.

### 5.2.3 The Trace Function Method (TFM), (Parnas and Dragomiroiu, 2006, Parnas, 2009a)

The Trace Function Method (TFM) (Parnas and Dragomiroiu, 2006, Parnas, 2009a), has been developed to provide precise specification/description of black box structures. Documents such as module interface specification (MIS) and service requirement documents (SRD) may be documented using this method. This section introduces the TFM method based on what is discussed in Parnas et. al. (Parnas and Dragomiroiu, 2006, Parnas, 2009a), with a few extension and example provided in the end.

Given an interface or a set of variables $V$, which influence the state of modules, components or connectors the runtime behavior of these entities may be given by defining output functions for every output variable in terms of the history of all input variables.

The specification/description may be provided in two forms: an upper face specification or a full-specification. Upper face specification considers only the subset of input/output variables that are externally visible for user components of the module. On the other hand, a full-specification provides all input/output variables including those in the lower face of the module which are used to communicate with the components it uses. The full specification is needed to make in depth reliability assessment.

In TFM, a software module is assumed to have access to two distinct data structures.

- a hidden (internal) data structure that stores the module’s memory of its history but is not directly accessed by users and
- a global data structure that must be accessed by the module to receive or transmit information.
In TFM, a module is considered as a finite state machine operating at discrete points in time, called events. At each event, the module will do some combination of the following:

- read some of the global variables (e.g. via input parameters),
- change its internal state, and
- change the value of some of the global variables.

Each element of the global structure must have a unique identifier (i.e. there must be a 1:1 mapping between the variables and identifiers for use in event descriptors. The values of these variables before and after each event are used to define event descriptors, and traces, which are defined as follows.

A **full event descriptor** specifies the values of every variable in the global data structure before and after the event.

An **abbreviated event descriptor** contains only the before and after values and names of the variables in the global data structure that are either read or changed during the event.

A **trace** $T$ is a finite sequence of event descriptors; it describes a sequence of events.

A **subtrace** of a trace $T$ is a sequence of the event descriptors that is contained within a trace $T$.

A **prefix** of a trace $T$ of length $n$ is a subtrace of $T$ that contains the first $n$ elements of $T$.

A **suffix** of a trace $T$ of length $n$ is a subtrace of $T$ that contains the last $n$ elements of $T$.

A **history** is a trace that accurately describes all the events that affected a module beginning with initialization.

A TFM component interface document comprises:

- a complete description of the component’s inputs, and
- a complete description of the component’s outputs, and
- a description of a set of relations describing the value of each output to the history of the values of the inputs.

There must be one relation for each of the component outputs where

- the range of each relation is the set of possible values for the associated output variable
• the domain of each relation is the set of all possible histories for that component (the trace of the component)

If the behavior being documented is deterministic, the relations will be functions.

A TFM document is *complete* if there is a relation for every output and the complete set of possible traces for which the value of each output is defined is included in the domain of every relation.

A TFM document is *consistent* (not contradicting) if each individual relation is consistently defined. This is because, every output is defined separately (dependent only on inputs and earlier values of other outputs.)

Various ‘library’ functions that are applicable for event descriptors and traces are defined and provided in the aforementioned reference. A list of these functions includes:

1. For a variable V and an event e, ‘V(e) denote the value of V immediately before the event e, and V’(e) denotes the value of V after the event e
2. The name **PGM** is reserved to contain the name of the program invoked (if any)
3. “.” is used to indicate concatenation, and “_” denotes an empty trace
4. L(T): length of T gives the number of event descriptors in T
5. r(T) (**most recent**): returns the most recent event descriptor in the trace
6. o(T) (**oldest**): returns the oldest event descriptor in the trace
7. p(T) (**precursor**): returns the trace that excludes the r(T)
8. s(T) (**subsequent**): returns the trace that excludes o(T)
9. rn(T) (**most recent n**): returns the n most recent elements of T
10. pn(T) (**precursor of most recent n**): returns the prefix of T such that T= pn(n,T).rn(n,T)
11. on(n,T) (**oldest n**): consists of the oldest n elements of T.
12. sn(n,T) (**subsequent n**): is a trace consisting of all of the elements of T subsequent to on(n,T) in the order in which they appear in T.
13 mrcall(pg, T) (most recent call): returns the most recent event descriptor in which the program pg was invoked.

14 ex(P)(T) (exists): is true iff T contains an event descriptor that satisfies event predicate P

15 rs(P)(T) (recent such that): returns the most recent event descriptor in T that satisfies P

16 os(P)(T) (oldest such that): returns the oldest event descriptor in T that satisfies P

17 et(P)(T) (extracted trace): is a trace that contains the events from T that satisfy P in the order that they appear in T

18irst(P)(T) (index recent such that): returns the index of the most recent event in T that satisfies P

19 iost(P)(T) (index oldest such that): returns the index of the oldest event descriptor in T that satisfies P

Extensions

20 re(e,T) (recent since event e): returns the subtrace that starts with event e up to the most recent event in T.

21 n(T): next event (future event) in the Trace

22 elt(e): elapsed time since event e

5.2.3.1 Examples of a trace function document:

5.2.3.1.1 Time Measurement Module

5.2.3.1.1 Output Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id&gt;.Hr</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.Min</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.TimeZone</td>
<td>&lt;Integer&gt;</td>
</tr>
<tr>
<td>Id</td>
<td>&lt;Time&gt;</td>
</tr>
<tr>
<td>ue</td>
<td>&lt;UndesiredEvent&gt;</td>
</tr>
</tbody>
</table>
### 5.2.3.1.1.2 Input Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM</td>
<td>{NEWTIME, ++, --, :=}</td>
</tr>
<tr>
<td>&lt;id&gt;.Hr</td>
<td>&lt;Integer&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.Min</td>
<td>&lt;Integer&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.TimeZone</td>
<td>&lt;Integer&gt;</td>
</tr>
<tr>
<td>Timeid</td>
<td>&lt;Time&gt;</td>
</tr>
</tbody>
</table>

### 5.2.3.1.1.3 Event Descriptors:

<table>
<thead>
<tr>
<th>Program name</th>
<th>'in</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEWTIME</td>
<td></td>
<td>(PGM: NEWTIME, oid')</td>
</tr>
<tr>
<td>&lt;id&gt;.HR:=</td>
<td>&lt;integer&gt;</td>
<td>(in, Hr')</td>
</tr>
<tr>
<td>&lt;id&gt;.MIN:=</td>
<td>&lt;integer&gt;</td>
<td>(in, Min')</td>
</tr>
<tr>
<td>&lt;id&gt;:=</td>
<td>{∅, &lt;Time&gt;}</td>
<td>(in, id', Hr', Min', TimeZone')</td>
</tr>
<tr>
<td>++</td>
<td></td>
<td>(++&lt;id&gt;, hr', min')</td>
</tr>
<tr>
<td>--</td>
<td></td>
<td>(--&lt;id&gt;, hr', min')</td>
</tr>
</tbody>
</table>

### 5.2.3.1.1.4 Auxiliary Functions

\[ hr(T_{<id>}) \equiv \]

\[
\begin{array}{|c|c|}
\hline
(T= _) & PGM(r(T) = NEWTIME) \\hline
\neg (T= _) & 0 \\hline
(r(T_{<id>})= (\langle id \rangle.Hr := in)) & 0 \leq \text{in}(r(T)) \leq 24 \\hline
\neg (0 \leq \text{in}(r(T)) \leq 24) & \neg \text{hr}(p(T)) \\hline
(r(T)\equiv (\langle id \rangle.MIN:=)) \lor (r(T)\equiv (\langle id \rangle.TIMEZONE:=)) & \neg \text{hr}(p(T)) = 23 \\hline
(r(T)\equiv ++(\langle id \rangle) & (\text{min}(p(T))=59) \land hr(p(T)) = 23 \\hline
\neg (\text{min}(p(T))=59) & \neg hr(p(T)) \land hr(p(T)) = 23 \\hline
(r(T)\equiv --(\langle id \rangle) & \text{min}(p(T))=0 \land hr(p(T)) = 0 \\hline
\neg (\text{min}(p(T))=0) & hr(p(T)) = 0 \\hline
\end{array}
\]

\[ \neg (T= _) \land PGM(r(T) = NEWTIME) \]

\[
\begin{array}{|c|c|}
\hline
(r(T)\equiv (\langle id \rangle.MIN:)) & 0 \leq \text{in}(r(T)) \leq 59 \\hline
\neg (0 \leq \text{in}(r(T)) \leq 59) & \text{min}(p(T)) = 59 \land hr(p(T)) \\hline
(r(T)\equiv (\langle id \rangle.Hr := in)) \lor (r(T)\equiv (\langle id \rangle.TIMEZONE:=)) & \text{min}(p(T)) = 0 \land \neg hr(p(T)) \\hline
(r(T)\equiv ++(\langle id \rangle) & \text{min}(p(T))=59 \land \neg hr(p(T)) \\hline
\neg (\text{min}(p(T))=59) & \text{min}(p(T)) = 0 \\hline
(r(T)\equiv --(\langle id \rangle) & \text{min}(p(T))=0 \land \text{min}(p(T)) = 0 \\hline
\neg (\text{min}(p(T))=0) & 1 + \text{min}(p(T)) \\hline
\end{array}
\]

\[ \text{min}(T) \equiv \]

\[
\begin{array}{|c|c|}
\hline
(T= _) & PGM(r(T) = NEWTIME) \\hline
\neg (T= _) \land PGM(r(T) = NEWTIME) & 0 \\hline
\neg (T= _) & 0 \\hline
(r(T)\equiv (\langle id \rangle.MIN:)) & 0 \leq \text{in}(r(T)) \leq 59 \\hline
\neg (0 \leq \text{in}(r(T)) \leq 59) & \text{min}(p(T)) = 59 \land hr(p(T)) = 0 \\hline
(r(T)\equiv (\langle id \rangle.Hr := in)) \lor (r(T)\equiv (\langle id \rangle.TIMEZONE:=)) & \text{min}(p(T)) = 0 \land \neg hr(p(T)) \\hline
(r(T)\equiv ++(\langle id \rangle) & \text{min}(p(T))=59 \land \neg hr(p(T)) \\hline
\neg (\text{min}(p(T))=59) & \text{min}(p(T)) = 0 \\hline
(r(T)\equiv --(\langle id \rangle) & \text{min}(p(T))=0 \land \text{min}(p(T)) = 0 \\hline
\neg (\text{min}(p(T))=0) & 1 + \text{min}(p(T)) \\hline
\end{array}
\]
5.2.3.1.1.5 Output functions

\[
\begin{align*}
\text{id(T)} &\equiv \\
(T=\_ & PGM(r(T) = \text{NEWTIME}) \\
\neg(T=\_ & \neg PGM(r(T) = \text{NEWTIME})
\end{align*}
\]

\[
\begin{align*}
\text{ue(T)} &\equiv \\
(T=\_ & PGM(r(T) = \text{NEWTIME}) \\
\neg(T=\_ & \neg PGM(r(T) = \text{NEWTIME})
\end{align*}
\]

5.2.3.2 TFM for Requirement Specification of Keyboard Checker Software: (Parnas, 2009b)

Figure 5-1 shows the requirement specification of a keyboard checker used at Dell’s assembly plan in Limerick Ireland as documented in (Parnas, 2009b) and (Baber et al., 2005). According to the first reference “the requirements for this system were documented in two memoranda totaling 21 pages. Although it had been in use for years, and was believed to be correct, the construction of a systematic document revealed several ambiguities, errors, and omissions. The figure, supplemented by definitions of the predicates applied within it, contains all of the requirements information, without ambiguities, and is far easier to use as a source of details than the original, informal documents”.

\[
\begin{align*}
\text{id(T)} &\equiv \\
(T=\_ & PGM(r(T) = \text{NEWTIME}) \\
\neg(T=\_ & \neg PGM(r(T) = \text{NEWTIME})
\end{align*}
\]

\[
\begin{align*}
\text{ue(T)} &\equiv \\
(T=\_ & PGM(r(T) = \text{NEWTIME}) \\
\neg(T=\_ & \neg PGM(r(T) = \text{NEWTIME})
\end{align*}
\]

\[
\begin{align*}
\text{5.2.3.1.1.5 Output functions} &\\
\text{<id>.hr} = Hr(T_{<id>}) \\
\text{<id>.min} = \text{min}(T_{<id>}) \\
\text{<id>.sec} = \text{Sec}(T_{<id>}) \\
\text{id} = \text{id}(T)
\end{align*}
\]
5.3 Mathematical Basis for Description of System Structure
Most of the system structural behavior that this author is interested in studying can be represented in the form of relations, connection matrices and/or graphs, involving basic set theory, probability theory, Boolean algebra & linear algebra. Below are definitions for some of the concepts that are less obvious.

5.4 Relations and Characteristic predicates:
A relation is a set of tuples. Different relations between various entities are characterized by their characteristic predicates. A characteristic predicate describing a relation is a predicate which holds true (1 in Boolean algebra) only for tuples that are members of the relation and false (0 in Boolean algebra), otherwise. For a binary relation $R$ on a set $S$, and $a, b \in S$

$$R(a, b) = \begin{cases} 1, & \text{if } a R b, \text{ or } b \in R(a) \\ 0, & \text{otherwise} \end{cases}$$

Equation 5-1
5.5 Matrix/Vector Representations

When binary relations involve a finite set of entities in their domain and range, we can conveniently describe them in binary matrices whose rows correspond to the domain of the relation (in vector form) and columns represent the range of relation (in vector form) and the entries of the matrix take the value of the characteristic predicate of the relation on the tuple created from the row and column index.

If binary relation R is defined from domain A of size m, to range B of size n, then R may be represented by an m x n Boolean matrix:

\[ R(A, B) = r(A^T \times B) = [r(a_i, b_j)] = [r_{ij}], \]

where

\[ r(a_i, b_j) = \begin{cases} 1, & \text{if } a_i R b_j \\ 0, & \text{otherwise} \end{cases} \]

Equation 5.2

Ternary relations, relations that involve three variables may also be represented in matrix form where the rows and columns represent the first two variables and the entries of the matrix contain the values of the third variable. For a ternary relation R, from sets A and B to set C, its matrix representation can be provided as:

\[ R(A, B, C) = r(A^T \times B \rightarrow C) = [r(a_i, b_j)] = [r_{ij}], \]

where

\[ r(a_i, b_j) = \begin{cases} \{c | (a_i, b_j) \in R \}, & \text{if } c \in R \\ \phi, & \text{otherwise} \end{cases} \]

Equation 5.3

5.6 Relational Composition

Two relations can be composed to form a third relation if the range of the first relation is a subset of the domain of the second relation.

Let A, B and C be sets and R and G be relations with signatures given as

\[ R: A \rightarrow B \]
\[ G: B \rightarrow C \]

Equation 5.4

Then, the composition, R o G (or R(G)) is a relation which can be considered as either a binary relation involving (A, C) or a ternary relation involving (A, B, C).

As a binary relation from A to C the signature of R.G. may be given as
R@G: A \rightarrow C \quad \text{Equation 5-5}

And its meaning is given by

\[ R \circ G = \{(a,c) \in A \times C \mid \exists b \in B : (a,b) \in A \times B \land (b,c) \in B \times C\} \subseteq A \times C \quad \text{Equation 5-6} \]

When a binary relation represents the composition of two binary relations as shown above, there is loss of some information. In some cases, we may be interested not only in knowing ‘a’ and ‘c’ are related but also in knowing that they are related through ‘b’. To retain such information, we use a ternary relation whose signature may be written as:

\[ R \circ G: A \times B \times C \]

Its definition is given as follows. For \( a \in A, b \in B, c \in C, \)

\[ R \circ G = \{(a,b,c) \in A \times B \times C \mid \exists b \in B : (a,b) \in A \times B \land (b,c) \in B \times C\} \subseteq A \times B \times C \quad \text{Equation 5-7} \]

### 5.7 Compositional Operators

Relational composition for multi-entity relations represented in binary matrices is almost identical to standard matrix multiplication in linear algebra, except for variations on the sum and product operators.

Let \( R \) and \( G \) be multi-entity relations represented in \( m \times n \) and \( n \times p \) binary matrices as shown below

\[
R(A,B) = r(A^T \times B) = [r(a_i,b_j)] = [r_{ij}],
\]

where

\[
r(a_i,b_j) = \begin{cases} 
1, & \text{if } a_i R b_j \\
0, & \text{otherwise}
\end{cases}
\]

\[
G(B,C) = g(B^T \times C) = [g(b_i,c_j)] = [r_{ij}],
\]

where

\[
r(b_i,c_j) = \begin{cases} 
1, & \text{if } b_i G c_j \\
0, & \text{otherwise}
\end{cases}
\]

Then, depending on our choice of the result of the relational composition to be given in either binary or ternary relation, we can have two sets of matrix product operators that give \( m \times p \) matrices.
5.7.1.1 Binary composer (*)
When this operator multiplies two binary matrices, the result is a binary matrix whose entries are formed from matrix multiplication rules where the logical and ‘∧’ is used as the product operator and logical or, ‘∨’ as the sum operator:

\[ R^*G = [r_{ij}], \]

where

\[ r_{ij} = r(a_i, c_j) = \bigvee_{k=1}^{n} (a_i, b_k) \land (b_k, c_j) \]  \hspace{1cm} \text{Equation 5-10} \\

5.7.1.2 Ternary Composer (⊗)
This operator retains the intermediate information by using the set union ‘∪’ operator on the joining set, the vector B in the above example, as the sum operator and the logical and ‘∧’ operator as the product operator.

\[ R \otimes G = [r_{ij}], \]

where

\[ r_{ij} = r(a_i, c_j) = \bigcup_{k=1}^{n} b_k \otimes (a_i, b_k) \land (b_k, c_j) \]  \hspace{1cm} \text{Equation 5-11} \\

where, ⊗ is a binary operator from a Set-Boolean pair to a Set.

\{(Boolean, Set) \rightarrow Set, (Set, Boolean) \rightarrow Set\}

For s : set and \{0,1\} : Boolean literals

\[ \otimes = \{((0,s),\emptyset),((s,0),\emptyset),((1,s),s),((s,1),s)\} \]

\text{Equation 5-12}

5.8 Basic Concepts in Reliability Prediction
The main objective of this section is to introduce the basic concepts of classical probability and reliability theories that will be used in the thesis.

5.8.1 Reliability and Dependability
“Reliability of an entity is the ability of the entity to perform its required function for a specified time period when operating under stated environmental conditions. The term
‘entity’ is used to denote any component, subsystem, system or equipment that can be individually considered and tested separately.” (Villemeur, 1992b)

Close to reliability, there are related concepts that are collectively called dependability. Dependability is considered as the science of failure and encompasses the knowledge, assessment and their control (Villemeur, 1992b). It can be characterized by the following four concepts:

- Reliability: \( R(t) = P[E \text{ not failed during } [0,t]] \)
- Availability: \( A(t) = P[E \text{ not failed at instant } t] \)
- Maintainability: \( M(t) = P[\text{the maintenance of } E \text{ is completed by time } t] \)
- Safety: is the ability of an entity not to cause, under given conditions, critical or catastrophic events

### 5.8.2 Dependability Measures (Villemeur, 1992b)

For any given entity \( E \) (component, system, etc.), let \( T \) be the random variable measuring the uptime of the entity, that is the time between some reference time 0, for instance the time in which the entity is put to use, until the entity fails. The various dependability measures can be expressed in terms of the cumulative distribution function, \( F(t) = P[T \leq t] \).

- The reliability function of the entity is given by
  \[
  R(t) = P[E \text{ not failed during } [0,t]] = P[T > t] = 1 - P[T \leq t] = 1 - F(t)
  \]
  \[
  \text{Equation 5-13}
  \]

- The unreliability function is given by the cumulative distribution function
  \[
  \overline{R}(t) = 1 - R(t) = F(t)
  \]
  \[
  \text{Equation 5-14}
  \]

- The failure density function \( f(t) \) is written:
  \[
  f(t) = dF(t)/dt = d\overline{R}(t)/dt = -dR(t)/dt
  \]
  \[
  \text{Equation 5-15}
  \]

- The mean time to failure \( MTTF \), is given as the expected value of \( T \) and may be derived from
  \[
  MTTF = \int_{0}^{\infty} t f(t) dt = -\int_{0}^{\infty} \frac{d\overline{R}(t)}{dt} t dt = \int_{0}^{\infty} R(t) dt + [tR(t)]_{0}^{\infty}
  \]
Suppose that MTTF is defined; \( tR(t) \) approaches zero when \( t \) tends to infinity. Hence,

\[
MTTF = \int_{0}^{\infty} R(t) \, dt
\]

Equation 5-16

Similarly, given the cumulative distribution for repairing time \( T \) by \( M(t) \), the repair density \( g(t) \) is defined; we have

\[ g(t) = \frac{dM(t)}{dt} \]

Thus, \( g(t) \, dt \) is the probability of the repair being completed during time interval \([t, t+dt]\) given that the entity failed at time \( t=0 \). The mean-time-to-repair, MTTR, is thus deduced:

\[
MTTR = \int_{0}^{\infty} t \, g(t) \, dt = \int_{0}^{\infty} t \, \frac{dM(t)}{dt} \, dt = \int_{0}^{\infty} [1 - M(t)] \, dt
\]

Equation 5-17

- **Failure rate**: is the limit, if it exists, of the ratio of the conditional probability that the instant of time \( T \) of a failure of an entity falls within a given time interval \([t, t+\Delta T]\) to the length of this time interval when \( \Delta T \) tends to zero given that the entity has not failed over \([0,t]\).

\[
\Lambda(t) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} P[E \text{ failed from time } t \text{ to } t+\Delta t \text{ given that it did not fail over time period } [0,t]]
\]

Using the theorem of conditional probabilities, we get

\[
\Lambda(t) = \frac{-dR}{dt}(t) \geq 0
\]

\[
\Lambda(t) = \frac{f(t)}{R(t)}
\]

Equation 5-18

This failure rate is also called ‘instantaneous failure rate’. The following relations can be obtained from the above.

\[
\int_{0}^{t} \Lambda(u) \, du = -\int_{0}^{t} \frac{dR(u)}{R(u)} \, du = -\int_{0}^{t} \frac{dR(u)}{R(u)} = -\int_{0}^{t} d \ln(R(u)) = - \ln(R(t))
\]

\[
R(t) = \exp \left[ -\int_{0}^{t} \Lambda(u) \, du \right]
\]

\[
f(t) = \Lambda(t)R(t) = \Lambda(t) \exp \left[ -\int_{0}^{t} \Lambda(u) \, du \right]
\]

Equation 5-19
• **Repair rate.** This is the limit, if it exists, of the ratio of the conditional probability that instant \( T \) corresponding to the completion of the entity repair be included within a given time interval \([t, t+\Delta t]\) to the length of this time interval when \( \Delta t \) approaches zero assuming that the entity was failed over time period \([0,t]\).

\[
m(t) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} P[E \text{ repaired between time } t \text{ and } t + \Delta t \text{ given that it failed over } [0,t)]
\]

Using the theorem of conditional probabilities, we get

\[
m(t) = \frac{dM}{dt}(t)
\]

\[
m(t) = \frac{1}{1 - M(t)} \geq 0
\]

**Equation 5-20**

The following relations also hold.

\[
\int_0^t m(u)du = -\int_0^t \frac{dM(u)}{1 - M(u)} du = \int_0^t \frac{d(1-M(u))}{1-M(u)} = -\int_0^t d\ln(1-M(u)) = -\ln(1-M(t))
\]

\[
1 - M(t) = \exp\left[-\int_0^t m(u)du\right]
\]

\[
M(t) = 1 - \exp\left[-\int_0^t m(u)du\right]
\]

\[
g(t) = m(t)\exp\left[-\int_0^t m(u)du\right]
\]

**Equation 5-21**

• In case of exponential distributions, the above equations can be written as:

\[
R(t) = e^{-\Lambda t}
\]

\[
MTTF = \frac{1}{\Lambda}
\]

similarly, assuming the repair rate \( \mu(t) \) is a constant \( m \), we have **Equation 5-22**

\[
M(t) = 1 - e^{-mt}
\]

\[
MTTR = \frac{1}{m}
\]
5.8.3 Signal Flow Graphs for Solving Reliability Problems

In most cases, solving reliability problems requires finding the solution of a system of equations representing different properties of systems, such as state transition rates, and probabilistic dependencies of events.

One useful mathematical tool that has been in use for many decades especially by Electrical Engineers is the signal flow graph (Lorens, 1956). It has also been used for solving probability (Rade, 1973), (Howard, 1971) as well as reliability problems (Burroughs and Happ, 1962), (Butler, 2000).

Its importance comes from well-known graph reduction techniques, its ability to map physical or logical network structure into equations, and the possibility of selectively solving for a specific transfer function.

Two properties of signal flow graph are interesting from reliability analysis point of view: as a means of solving a system of linear equations, and a convenient way of representing statistical dependencies among various elements (events, components, states, etc.) of interest for evaluating probabilities.

5.8.3.1 Solving System of Equations Using Flow Graphs

Given a system of $N$ linear equations with $N$ unknowns stated as (Marek and CZ, 2007)

$$x = Ax + b$$

Equation 5-23

where $A$ is a square $N \times N$ coefficient matrix, $x=[x_1, \ldots, x_N]'$ is an $N \times 1$ vector of unknowns and $b$ is an $N \times 1$ vector of free terms, assumed to be not equal to zero. By introducing an extra scalar variable $x_0=1$ that will be treated as an additional unknown with a fixed value, the system of equations may be re-written as:

$$\begin{bmatrix} x_0 \\ x \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ b & A \end{bmatrix} \begin{bmatrix} x_0 \\ x \end{bmatrix} = \begin{bmatrix} C \end{bmatrix} \begin{bmatrix} x_0 \\ x \end{bmatrix}$$

Equation 5-24

A signal flow graph $G$, associated with the system of equations given above is defined as a weighted directed graph $G$ having the set of $N+1$ vertices identical with the set of unknowns $\{x_0, x_1, \ldots, x_n\}$. There is an edge in $G$, with weight $c_{ji}$, directed from vertex $x_i$
to vertex $x_j$ if and only if $c_{ji} \neq 0$. The vertex $x_0$ is a source vertex (i.e. it does not have incoming edges). Any unknown $x_i$ of the system of equations can be computed as the transfer function of the associated signal-flow graph $G$, from $x_0$ to $x_i$ either by graph reduction techniques or by using the well known Mason’s rule (Lorens, 1956).

For a signal-flow graph $G$, the transfer function between $x_i$ to $x_0$ is given by:

$$\frac{x_i}{x_0} = \frac{\sum_j P_j \Delta_j}{\Delta}$$

Equation 5-25

where $x_0=1$ by convention. $\Delta$ is the determinant of the graph, $P_j$ is the transmittance of the $j$-th directed path, and $\Delta_j$ is the determinant of the maximal subgraph of $G$ that is vertex-disjoint with the $j$-th directed path, obtained by removing all vertices incident with this path. The determinant of a directed weighted graph is defined as:

$$\Delta = \sum_r \prod (1 - a_{ii})$$

Equation 5-26

where the first sum includes transmittance of all loops, the second sum includes products of transmittances of all pairs of disjoint loops, the third one products of transmittance of all triples of disjoint loops, and so on. $S_r$ is the product of 1 minus the self-loops disjoint from the loops in $L_r$.

$$S_r = \prod (1 - a_{ii})$$

Equation 5-27

The transmittance of a directed path or loop in a weighted directed graph is equal to the product of the weights of all edges of this path or loop.

5.8.3.2 Probability Analysis using Flow graph (Rade, 1973).

The flow graph can be used to represent and solve the occurrence of an event which may be given as a combination many other events. The basic translation from probability equation to flow graph are shown below.

Probability of an event (a,b,c), i.e. the thee events connected by ‘AND’

The conditional probability $p(b|a)$ of event (b) given event (a) is defined by SFG (1)
Provided that \( p(a) \) is not zero.

For several events, we have

\[
\begin{align*}
\text{p(a)} & \rightarrow \text{p(b|a)} \rightarrow \text{p(a,b)} \rightarrow \text{p(cla,b)} \rightarrow \text{p(a,b,c)} \\
\end{align*}
\]

If the events are statistically independent, then

\( p(b|a) = p(b) \), \( p(cla,b) = p(c) \). Hence the above SFG simplifies to:

\[
\begin{align*}
\text{p(a)} & \rightarrow \text{p(b)} \rightarrow \text{p(a,b)} \rightarrow \text{p(c)} \rightarrow \text{p(a,b,c)} \\
\end{align*}
\]

Probability of an event \((a+b+c)\), i.e., the three events connected by ‘OR’

\[
p(a+b) = p(a+b,a) + p(a+b, \overline{a}) = p(a) + p(\overline{a}, b)
\]

For three events, \( p(a+b+c) \)

\[
\begin{align*}
\text{1} & \rightarrow \text{p(a)} \rightarrow \text{p(a+b)} \rightarrow \text{p(a,b)} \\
\end{align*}
\]

If the events \( a, b \) and \( c \) are mutually exclusive, SFG simplifies to:

\[
\begin{align*}
\text{1} & \rightarrow \text{p(a)} \rightarrow \text{p(a+b+c)} \rightarrow \text{p(\overline{a},b)} \rightarrow \text{p(\overline{a},\overline{b},c)} \\
\end{align*}
\]
With the use of conditional probabilities to evaluate $p(\bar{a})$, $p(\bar{a}, b)$ and $p(\bar{a}, \bar{b}, c)$, we get:

5.9 Chapter Summary

This chapter has introduced basic concepts and principles that are to be used in later chapters. The concepts cover software design, documentation principles and reliability prediction areas. They include modules, components, composites, services, mathematical documents, tabular expressions, TFM specification, graphs, signal flow graph, dependability measures, etc. The definitions for some of them such as composites and services, and part of the relation composition operators that is provided in this chapter may be different from what is given in the literature. For most others, the concepts are directly adapted from referenced sources in the three areas of research.

The signal flow graph concept is very well known in mathematics and other engineering disciplines, but not so much in software engineering. It will be used in making dynamic property analysis, such as timing and resource availability as well as modeling of reliability functions. The reliability or dependability measures are standard text book materials and are covered for ease of access. Tabular expressions are also well known documentation approaches in some safety critical systems, if not yet fully popularized in
the software industry. The TFM specification method is relatively the most recent invention, in which not much reference is available in the literature. It will be the basis for specifying or describing the interfaces of modules and components. Hence, this method is supplemented with examples.
6. Design Imperatives Derived from Reliability Theory

6.1 Introduction
The objective of this chapter is to identify a set of design imperatives, which the author calls DRP Imperatives, if applied should result in improved reliability predictability of computer systems. Their identification is based on structural reliability theory and failure behavior of software systems.

6.2 System Reliability Structure
To identify the design restrictions for improved reliability predictability, let us first study a simplified view of the reliability structure of systems composed of a set of interacting components. Three elementary configurations are shown in reliability structure diagram form in Figure 6-1, which are given for analysis purpose, not as complete representation of reality:

1. Series Systems– Figure 1.a shows a system composed of a series connection of n components. An example of such configuration is a system that consists of n components where there is no fault-tolerance, protection or redundancy role played by any of the components.

2. Series-Parallel Systems: Figure 1.b shows series-parallel configuration where n-components connected in series are then connected in parallel. The configuration basically creates two redundant systems and may be referred to as system-level redundancy. Example: the development of two-multiversion systems through multiversion programming approach may result in systems of this configuration.

3. Parallel-Series Systems: Figure 1.c shows parallel-series configuration where n-pairs of parallel components are connected in series. Since the redundancies are at module levels, we may call this configuration modular redundancy.
Figure 6-1: Reliability structure diagram for three elementary system configurations

For statistically independent components\(^2\), system reliabilities for the three configurations may be obtained as a function of component reliabilities using the following well-known formulas.

---

\(^2\) Statistical independence may not usually hold for software components, but that will be discussed later.
Equation 6-1

\[
R_{S1} = \prod_{i=1}^{n} R_{i}
\]

\[
R_{S2} = \prod_{i=1}^{n} R_{i} + \prod_{i=1}^{n} R_{2i} - \prod_{i=1}^{n} R_{i} R_{2i}
\]

\[
R_{S3} = \prod_{i=1}^{n} (R_{i} + R_{2i} - R_{i} R_{2i})
\]

6.3 DRP Imperative 1: Design for Modular Redundancy and Fault Tolerance

Equation 6-1 provides us some insight on what to seek for in design for improved reliability. If statistically independent components of similar reliability are used in the above three configurations, it can easily be seen that:

\[R_{S3} \geq R_{S2} \geq R_{S1}, \text{ for } n>2\]

This shows the advantage of modular redundancy over system level redundancy and single version non-redundant systems for \(s\)-independent components. Hence, an important design principle can be stated as:

- Identify design restrictions that support modular redundancy.

Modular redundancy requires the ability of replacing one module by another upon failure (standby redundancy) or the ability of running multiple modules in parallel (active redundancy).

In both cases, the main required property is changeability of modules in a system which is very close to the requirements of design for change that gave rise to the information hiding (IH) principle.

The main difference between design for change and design for reliability predictability is for reliability to be improved through redundancy the changeability requirement can be a run time property (i.e. while the system is under operation) rather than a maintenance property (i.e. when the system is not operational or is in idle state).

This property is dependent on both system decomposition and composition approach. For instance, if a designer decomposes a system into two modules A and B and then composes them into one system in a way that does not allow changeability (except through re-programming) by making them interdependent, then the system reliability is a
product of the reliabilities of the two components, since the two components have become inseparable i.e.

\[ R_{S1} = R_A \cdot R_B. \]

**Equation 6-2**

Alternatively, the designer may identify abstract interfaces \( I_A \) and \( I_B \) that have to be implemented by both components and use connectors that link component A to \( I_B \) and component B to \( I_A \). A system created by connecting the two components through their abstract interfaces may still have a reliability structure similar to the previous one, i.e., \( R_S = R_A \cdot R_B, \) if both have to collaborate for the system to work.

However, there is a major difference with the first approach in that, the designer can now include two more components say \( A_2 \) and \( B_2 \) implementing the same interfaces \( I_A \) and \( I_B \) respectively. This creates the possibility of obtaining four different systems \{AB, A_2B, AB_2, A_2B_2\}.

If these four combinations are considered as redundant paths where the successful execution of any one of them ensures continuity of system service, then for non-interfering components, the reliability of the system may be given as:

\[ R_{S2} = (R_A + (1-R_A)R_{A2|A'}) (R_B + (1-R_B)R_{B2|B'}). \]

**Equation 6-3**

Where \( R_{A2|A'} \) refers to the conditional probability of component \( A_2 \) operating when component A has failed and the B’s are assumed to not interfere with A’s.

For non-interfering statistically independent components, we have

\[ R_{A2|A'} = R_{A2} \text{ and } R_{B2|B'} = R_{B2} \]

Substituting this in Equation 6-3, we get:

\[ R_{S2} = (R_A + R_{A2} - R_A R_{A2}) (R_B + R_{B2} - R_B R_{B2}). \]

**Equation 6-4**

The above analysis implies the following two design approaches.

- During system decomposition, apply the principle of information hiding. That is, partition all design decisions as secrets of separate modules where the change of a module does not affect other modules.
During system composition, identify connection mechanisms that can support runtime changeability or replacement of components. This requires the use of abstract interfaces that may be implemented by different modules which may need to be substitutable to each other and/or bind to generic connectors that can connect to different types of modules, and support parallel execution and/or replacement. Both of these approaches may not be entirely new as they have been known both from design for change objective (Parnas, 1972a) as well as design for reusability objective (Erich et al., 1995).

The main difference in here is to adapt these design properties for supporting redundancies (whenever needed), fault-tolerance, and restricting inter-dependencies to achieve a desired reliability structure. To this effect, the principles of information hiding and abstraction are re-enforced by design for reliability predictability.

6.3.1 Statistical Dependencies and Interferences

Equation 6-4 harbors a difficult to prove assumption – statistical independence of component failure behavior. Unfortunately, redundancies in software components may suffer from statistical dependencies as well as interferences that may be caused by the following reasons among others:

- Shared finite computing resources – if many components that can independently perform well are put together to share a finite set of resources, then interaction failure could arise due to the possible depletion or exhaustion of resources, deadlocks, livelocks that results from sharing of resources.

- Shared design errors - execution of two identical software components on different computing nodes may provide some limited redundancy against secondary failure causes and interaction failures but not effectively with primary failure causes since both components will most probably fail at the same time for exactly identical inputs.

- Coincident failures - reliability improvement proposals through multiversion programming (Avizienis, 1995) have been challenged by the occurrence of coincident failures (Knight, 1989). There are a number of reasons why independently developed multiversion systems may fail dependently which include:
shared complexity of problem domain, limited computing power of machines and similar thought processes.

Hence, equations such as the one given in Equation 6-4 that assumes statistical failure independence between components do not provide realistic estimates for software reliability.

In the presence of statistical dependencies, the reliability expression given in Equation 6-3 by $R_{S2}$ may not be significantly better than that obtained for $R_{S1}$ in Equation 6-2. For instance, if the supposedly redundant components are fully dependent, then the probability of survival of a component on the condition that the corresponding redundant component has failed can be zero. Thus, we may have, $R_{A2|A'} = R_{B2|B'} = 0$ giving rise to $R_{S2} = R_{S1} = R_A R_B$

There can even be cases where the supposedly redundant systems may perform worse than simple series system. This happens when the interconnections among the four components allow failure of any of the components to affect the others.

To derive the reliability formula for redundant components that could interfere with each other, let us partition all failures of each component into two modes: critical failure mode – a failure mode that can affect all or critical system components and a derated failure mode- a failure mode that affects only the component and those that directly depend on its outputs but not the redundant.

Let the probability of being in any of the three states {normal, derated, critical} for each component $x \in \{A, A_2, B, B_2\}$ in the above system be given by $p_{xn}$, $p_{xd}$, $p_{xc}$, with $p_{xn} + p_{xd} + p_{xc} = 1$. The possible set of system states can be evaluated as:

$$s = [(p_{An} + p_{Ad} + p_{Ac}) \cdot (p_{A2n} + p_{A2d} + p_{A2c})] \cdot [(p_{Bn} + p_{Bd} + p_{Bc}) \cdot (p_{B2n} + p_{B2d} + p_{B2c})]$$

$$= [(p_{An} \cdot p_{A2n} + p_{An} \cdot p_{A2d} + p_{An} \cdot p_{A2c}) + p_{Ad} \cdot p_{A2n} + p_{Ad} \cdot p_{A2d} + p_{Ad} \cdot p_{A2c} + p_{Ac} \cdot (p_{A2n} + p_{A2d} + p_{A2c})] \cdot [(p_{Bn} \cdot p_{B2n} + p_{Bn} \cdot p_{B2d} + p_{Bn} \cdot p_{B2c} + p_{Bd} \cdot p_{B2n} + p_{Bd} \cdot p_{B2d} + p_{Bd} \cdot p_{B2c} + p_{Bc} \cdot (p_{B2n} + p_{B2d} + p_{B2c}))]$$

**Equation 6-5**
The operational states are states that do not have any of the critical failure modes from any of the four components, and those states in which at least one component from each redundant pair is in normal mode. Hence, the reliability of the system is given by:

\[
R = \left[p_{An}p_{A2n} + p_{An}p_{A2d} + p_{Ad}p_{A2n}\right] \cdot \left[p_{Bn}p_{B2n} + p_{Bn}p_{B2d} + p_{Bd}p_{B2n}\right]
\]

Equation 6-6

With the presence of critical failure mode, it is possible for the above equation to be less than the reliability of non-redundant systems, i.e. a system consisting of component A and B connected in series. This can be shown by simplifying the above equation with assumption of identically distributed failure behavior for components. Let us represent the probability of being in any of the three modes as \(p_n\), \(p_d\), \(p_c\). Equation 6-6 can be re-written as:

\[
R = \left(p_n^2 + 2p_np_d\right)^2 = \left(p_n^2 + 2p_n(1 - p_n - p_c)\right)^2
\]

\[
R = p_n^2\left(2 - p_n - 2p_c\right)^2
\]

Equation 6-7

Since a system consisting of two components in series connection has a reliability of \(p_n^2\), for the above system to perform better than a system without redundancy, the following condition need to be satisfied.

\[
p_n^2\left(2 - p_n - 2p_c\right) > p_n^2
\]

\[
2 - p_n - 2p_c > 1
\]

\[
p_c < \frac{1}{2}(1 - p_n) = \frac{1}{2}(p_c + p_d)
\]

\[
p_c < p_d
\]

Equation 6-8

Equation 6-8 shows us that unless the critical failure modes of components is not less than their derated failure mode, redundancy can even backfire by performing worse than series systems.

Interferences or interaction failures can occur in different ways, one of which may be due to complexity. Some of software reliability improvement techniques, such as recovery block models (Randell, 1975) did not succeed due to the complexities they introduced to the system (Xu et al., 2002).
Thus, the reliability structure of software systems is not only a function of the components that have formed the system but also the type of interconnection among them. Some interconnections can cause failure dependencies even among supposedly redundant software components.

As a result, the following two other goals may be stated for improved reliability predictability:

*Imperative 2*: Ensure non-interfering behavior of interconnections, especially in case of redundancies.

*Imperative 3*: Identify properties that improve the statistical independence of components or develop mathematical models that capture the correlation among redundancies.

### 6.4 DRP Imperative 2: Ensure Non-interfering Behavior of Connections

The second design imperative requires us to design interconnection mechanisms that avoid interference of components with each other, especially in cases of redundancies.

Various interaction failures can arise when a network of components share a finite set of resources for their computation. These include:

- Race conditions where assumptions about the input or environment change before an output of a component is computed or consumed,
- Resource depletion and congestions – where computing resources such as processors, memory, communication channels, etc. may not be available to undertake a particular task at a time point because of being used by other components in the system
- Deadlocks or livelocks – where two or more processes hold some shared resources and wait for other additional resources in a way that none of them are able to progress any further
- Missing of deadlines – where component responses may come too late to be useful for a particular task
- Undesired dependent failures – where the failure of some components that could have been tolerated causes system failure due to unrestricted dependencies among components in the system
- Incorrect event orders – where events occur in an order that may cause system failure.

The possible occurrence or non-occurrence of these undesired events is dependent on the structuring of connectors – input/output variables of modules - and events or processes that change these variables.

For instance, the use of unrestricted pointers for communication among modules can lead to one module over-writing on critical data that is required by other modules leading to system failure. Similarly, scheduling all events to occur in one sequential process can result in undesired dependent failure in which any component that could not terminate causes complete system failure. Additionally, missing of deadlines will be more probable for time sensitive components if all events have to be sequentially ordered by design.

The partitioning of events into communicating concurrent processes where each component can operate on its own process space can improve fault-tolerance, where troubled components may be removed, restored or even replaced while the rest of the system continues its operation without interruption. Concurrency can also improve reliability with respect to meeting deadlines through prioritized allocation of processor time to time-sensitive processes and avoiding unnecessary waiting times during input-output bound operations. However, concurrency can introduce its own reliability risks such as possible occurrence of race conditions, deadlocks and livelocks unless protective mechanisms such as synchronizers are put in place.

Thus, avoidance or reduction of interaction failures requires the organization of events into process structures that minimize such failure causes especially to critical system components and the use of connectors that have protection mechanism against race conditions, undesired component dependencies, undesired event orders, deadlocks and livelocks.
6.5  DRP Imperative 3: Improve Statistical Independence among Possible Redundant Components and Services

The third design imperative is to identify properties that improve the statistical independence among redundant components and services when there are any.

This is generally a difficult problem that cannot be easily proved or tested. This is because, whether we have redundancies or not, an important goal of software design is to make every module as reliable as possible, that is, it must satisfy all the requirements specified by the interfaces it provides. If modules are constructed to be reliable, there would not be sufficient failure data that can be used to test for statistical independence among possible redundancies. Moreover, proving things to be independent based on data is impossible.

Yet, one may encounter many applications with high dependability requirements where some form of redundancy is essential to improve confidence on the systems even if it may be quite difficult to quantify the reliability gain. Software redundancy may be provided in various ways that fall into two categories: single version systems, divers systems.

6.5.1 Redundancy in Single Version Systems

Most software engineering problems are complex and require the construction and/or composition of a large set of modules. As a result, much of software design is done through the decomposition of software into reusable components that can be composed in various ways to provide different services.

Usually, flexible structures that use proper abstraction and information hiding concepts can allow not only replacement of one component by another component that provides equivalent functions but also the replacement of one component by another that has identical input-output variables and communication protocols but potentially different mapping function. Such replacements can generally be used to create different services from a set of components or to produce multiversion products that can be deployed on different platforms.
The same approach can be used to support modular redundancy without increased complexity. There are many reasons why such practices are more common compared to the development of multiversion-redundant systems among which include:

- The principle of information hiding and similar concepts such as separation of concerns help designers to tame the complexity of software systems by building components that hide complex internal interactions behind simple abstract interfaces. Properly harnessed, these principles can be used to create diverse redundant modules that can be interconnected in different ways to improve system reliability. For instance, depending on the application type, diversity may imply the use of different coordinate systems, different algorithms, different internal components, etc.

- Providing (primitive) component redundancy through component replication and/or independent instantiation, reconfiguration of components or services in response to undesired events, and in some cases, restoration of failed instances to their initial states are relatively easily achievable in software systems compared to hardware systems provided that the interconnection structure is designed to support such changes. The effectiveness of these redundancies however depends on the nature of possible causes of undesired events or failure.

- In some cases, the effectiveness of primitive redundancies obtained from component or system replication can be improved through diversity of inputs. Two software systems that work in two independent input domains at any specific point in time may have a chance of having independent failure behavior even if they use identical software components.

- Protection against undesired events (fault-avoidance) and fault-tolerance of component failures are also techniques that may be used to improve system reliability without necessarily employing diverse redundancy. Note that non-interfering protective components can have exactly similar effect as redundant components in a system. Consider a system of two components, A and B, where component A plays a protective role in that the probability of B failing if component A is working properly is zero. However, the system may operate with component B alone with reliability of $R_B$, even if A fails or is not included. Then,
the system reliability that consists of A and B may be given by the following equation which is equivalent to a parallel connection of two independent components.

\[ R_s = R_A + (1 - R_A)R_B = R_A + R_B - R_AR_B \]

**Equation 6-9**

- For certain class of problems, where there are no known algorithms that are capable of giving solution using prevailing technologies, there is an upper limit to the maximum reliability that can be achieved. Neither diversified redundancy nor any type of formal verification technique can increase the reliability of such systems beyond the maximum reliability that can be achieved by correct implementation of the best algorithm/solution available for that problem.

- For other systems that are feasible to implement and have precisely specified requirements, most of the components are expected, at least theoretically, to have deterministic input-output behavior which makes it possible to construct even ‘bug free’ software systems. However, this does not imply 100% reliability due to uncertainties resulting from failure or depletion of resources, incompleteness of requirements, unavoidable imprecision in number representations and computations and occurrence of other undesired events. In fact, since undesired events can occur at any time, they usually get superimposed on almost all events making almost all components non-deterministic.

### 6.5.2 Redundancy in Diverse Systems

A software product that does not have any form of redundancy or fault tolerance can become a single point of failure or common mode failure in a complex system environment. The effectiveness of replicated redundancy, protections and other forms of fault tolerance can be limited to protecting against secondary failures but may not be good enough to protect against primary failure.

This may not be acceptable in applications where there is high dependability requirement. In such environments, diversified redundancy could be the only choice for improved confidence on the system, where the diversity is not necessarily achieved through N-version programming (NVP) paradigm, but also through forced diversity through design
decisions. The effectiveness of such redundancy may be improved by the following diversity requirements:

- Diversity in algorithms,
- Diversity in data representations, coordinate systems
- Diversity in inputs,
- Diversity in composite structures
- Diversity in services or system functions
- Diversity in development tools

Unfortunately, such diversity requirements do not guarantee statistical independence among redundant components. As a result, while it is possible to achieve arbitrary reliability requirements through redundant design of arbitrary unreliable components in hardware systems because of independence assumption, it is generally hard to achieve reliable software systems from unreliable components due to increased complexities, interactions and statistical dependencies.

Thus, the aforementioned solutions need to be supplemented by precise mathematical documentations and reliability structural models that can reveal the reliability structure of software systems, giving rise to the fourth imperative.

### 6.6 DRP Imperative 4: Describe Component and System Behavior Precisely

Whether we have redundancies or not, mathematical documentation of the behavior of components and systems is essential for reliability assessment since one cannot predict reliability without precise information about its components and their interconnections. In fact, it is not possible to talk of reliability of components or services in the absence of a reference specification against which the behavior of the products can be measured.

A mathematical documentation approach that is suitable for documenting requirements and design with information hiding behavior is introduced in (Parnas and Jan, 1995) and includes – System Requirement Document (SRD), Module Guide (MG), Module Internal Design Document (MIDD), and Module Interface Specification (MIS). Recently, a new interface specification method, the Trace Function Method (TFM) (Parnas and
Dragomiroiu, 2006), is invented to document the behavior of information hiding components precisely.

These documents are adapted and supplemented with additional system documents as described below to help in revealing the reliability structure of software systems.

6.6.1 Module Guide
The modularization process generally results in a hierarchy of modules based on the ‘is-secret-of’ relation and is documented using a module guide. The module guide consists of a formal tree structure and informal text description of the hidden secrets, and is the only semi-formal document in the collection of mathematical documentations used in this thesis.

6.6.2 Interface Documents
Once the modules of the system are identified, the set of interfaces that are used to interconnect the various modules of the system are defined. This process determines the existence or absence of certain inter-module connections, the preferred inter-dependencies among the various modules, undesired event communication and response strategies, fault-tolerance mechanisms and the possibility of supporting redundancies.

The interfaces comprise a set of input-output variables that can be used to create a network of components. The interfaces are specified in TFM. This approach can be considered to be equivalent to providing transfer function of components.

6.6.3 Connection Documents
Based on this author’s interest to support modular redundancies (which could possibly be as future evolution), fault-tolerance, and polymorphism, interfaces are considered as independent entities in the systems. This requires defining two relations between modules and interfaces: the PROVIDES and REQUIRES relations. PROVIDES specifies the relation between interfaces and their implementer modules whereas REQUIRES specifies the relation between interfaces and user modules.

Note that the design decisions about interfaces and their interconnections basically determine almost all of the system structures- inter-module connections, input-output connections, the event/process structures, fault-tolerance, protection and redundancy
mechanisms, etc. As a result this task requires a lot of information that may not be easily obtained, for instance, timing properties of components in order to decide on the process structure. Thus, the design process may have to be supported by mathematical models, prototypes and simulations or the specifications and descriptions may have to be developed following a rational (iterative and incremental) process approach.

The module-interface relations can be used to derive service-module dependency matrix. At runtime, modules are used to create component instances that are connected through their input-output variables to create a composite structure that makes required input-output state transformation. This can be represented using input-output connection matrices.

Superimposed over input-output connection matrices, exist event structures, where a change of state (an event) at one component is propagated to one or more other components in the network as change of state of other variables (events). The author captures the relation between events in a composite structure in the form of event-flow-equations or event-flow-graphs.

6.7 DRP Imperative 5: Model and Predict System Reliability

The set of documents described above can be used to derive the reliability structure of systems and services by first producing a system reliability structure document.

A system reliability structure document is a document where all services provided by a network of components are identified, for each service the role-played by the various components are listed out. Additionally, based on the procedures to be discussed in detail in chapter 8, all possible interaction failure causes are identified and represented as virtual components. Possible redundancies to the real components or the virtual components, if exist, are identified along with the possible source of diversity, independence or correlation among them.

Additionally, the document may consist of all the necessary information about each service demand rate and operational rate, the possible redundant services that may be provided by the system, the structural similarity or difference among the redundant
services, and the possible cross-service interferences. Service level redundancies can be indicated by simple predicate expressions over services.

These documents can be used to derive the reliability structure of each service provided by the system. The mathematical derivation of this process is discussed further in chapter-8.

6.8 Chapter Summary

In this chapter, the author identified the corner stones of design for reliability predictability, which are named as DRP Imperatives:

1. Design for Modular Redundancy
2. Ensure Non-interfering Behavior of Connections especially among Redundancies
3. Improve Statistical Independence among Redundant Components and Services
4. Describe Component and System Behavior Precisely/Mathematically
5. Model and Predict System Reliability

The approach on how to achieve these imperatives will be the focus of the next two chapters.
7. DRP Imperatives and Software Architecture

7.1 Introduction
This chapter elaborates the DRP imperatives identified in the previous chapter by defining and relating them to a number of structure functions. This is important since, it is the key to master the complexity of software systems, which is a pre-requisite for reliability predictability.

Today, a single software product can typically come (out of production) with binary code size of many giga bytes. Its reliability is not only determined by almost every bit of this gigantic program code, but also the extra many more giga bytes of data it generates or works with, and the various unquantifiable interactions it undertakes with its environment. And all this, is in complete digital / discrete representation where there is no continuity or coherence of anykind, and the combined state-space is simply unimaginable to think of let alone tractably model.

The only mechanism of mastering such complexity of software and predicting its reliability is through the introduction of various structures that enable us to undertake multilevel and multiview analysis of its various properties including its reliability. It is indeed through such type of reasoning about its reliability that software has continued to be deployed in critical and non-critical applications.

The focus of this chapter is to introduce such structure functions and to show how they are used in reliability assessment. The various software structures are mathematically defined as relations between the different elements of a system. These functions capture both the static and dynamic aspects of the system under construction.

7.2 Software Architecture
Software architecture is a term that has become common in recent years to describe the relation among various aspects of software systems, although there is no universally accepted definition. The reliability modeling community also uses the same term to describe the interaction among modules that is used to predict their reliability. Therefore, in order to avoid ambiguities, this author first looks into some of the definitions given in the literature and adapts the ones that are used in the rest of the thesis.
Among the various definitions include those that view architecture as:

1. a way of defining the system in terms of computational components and their interactions (Shaw and Garlan, 1996);

2. the structure of the components of a program/system, their interrelationships, and the principles and the guidelines governing their evolution over time (Garlan and Perry, 1995);

3. the overall design of a system, integrating separate but interesting issues of a system such as independent provisions for independent evolution and openness combined with overall reliability and performance (Szyperski, 2002);

4. the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them (Bass et al., 2003).

5. the complete and detailed specification of the user interface, “where architecture tells what happens, implementation tells how it is made to happen” (Fred, 1975)

6. Bass et al. (Bass et al., 2003) points out Dijkstra’s introduction of the layered structure as one of the major foundations of software architecture, and gives highest credit to Parnas for building the foundations for architecture citing his contribution in various concepts such as: information hiding (Parnas, 1972a), recognition of existence of different structures (Parnas, 2002), uses structure (Parnas, 1979), undesired event management (Parnas and Wurges, 1976), program families (Parnas, 1976), influence of structure on reliability (Parnas, 1975b).

7. From the reliability modeling community, architecture is widely used, with a typical definition given as: “The software behavior with respect to the manner in which the different modules of software interact is defined through the software architecture. Interaction occurs only by execution control transfer. In the case of sequential software, at each instant, control lies in one and only one of the modules. Software architecture may also include the information about the execution time of each module” (Katerina Goseva-Popstojanova, 2001).
The variations in the above definitions show the absence of an agreed-upon definition in the software engineering community. Partly this is due to the nature of the field which has to deal with various abstract concepts, ideas, design decisions, etc., that are not easy to formalize mathematically, in addition to the concrete final but invisible product, the program code. For instance, when following a hierarchical decomposition of modules based on the information they hide, one may come up with modules that only exist in the design space (e.g. module guide) but not in actual implementation as it may further be decomposed into various smaller size modules that have concrete implementations.

The other reason for the confusion of software architecture is the presence of various entities in a software system and their inter-relationship giving rise to different software structures. From the lowest representation level, *bits*, to the highest representation level *system*, software can be viewed in different ways and at different levels of detail. Its various entities are inter-related in a number of ways, giving rise to different structures. It is this collection of structures that this author refers to as software architecture.

While structure may be defined as a function or a relation whose domain and range are some elements or parts of a software product, architecture can be defined as a set of structure functions each of which is defining some aspect of a software product.

The different structure functions that are relevant to understanding software reliability will be precisely defined and their relationship with the design imperatives will be discussed in the sections to follow.

“If architecture has a fundamental principle, it is this one, which Parnas called information hiding”, (Bass et al., 2003)

### 7.2.1 Hierarchical Aggregates

A software system can be considered to have many parts among which include: statements, variables, programs, modules, interfaces, connectors, components, composites, events, processes, services and systems. The distinction among these elements is mainly a result of different levels of aggregation, where the underlying relation may have temporal, spatial or logical aspects. Many of these terms and concepts have appeared in various software engineering literature, some in relation to specific
technologies. However, the definitions and relations that are used in this thesis are given in section 5.1, and are further elaborated below and in the subsequent sections. Table 7-1 gives a *rough categorization* of the different parts, relating them through a ‘parts-of’ relation, forming *hierarchical aggregate structure*. Figure 7-1 shows the logical relationship among the various aggregates.

<table>
<thead>
<tr>
<th>Aggregate</th>
<th>Constituents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs</td>
<td>Statements</td>
</tr>
<tr>
<td>Connectors</td>
<td>Variables</td>
</tr>
<tr>
<td>Modules</td>
<td>Programs</td>
</tr>
<tr>
<td>Components</td>
<td>Modules</td>
</tr>
<tr>
<td>Composites</td>
<td>Components</td>
</tr>
<tr>
<td>Processes</td>
<td>Events</td>
</tr>
<tr>
<td>Services</td>
<td>Composites &amp; Processes</td>
</tr>
<tr>
<td>Systems</td>
<td>Services</td>
</tr>
</tbody>
</table>

**Table 7-1: Hierarchical Aggregation of Software Parts**

**Figure 7-1: Logical relationship between different aggregates of software system.**
The main relation depicted in Figure 7-1 is that of aggregation. A set of programs that must be modified or changed together form a module. A set of modules that may have to be deployed on one computing node may be put in a component while a set of components that must be connected by connectors to make state transformation make a composite. A connector itself can be considered as an aggregate of variables that may have to coexist in the memory system as a unit. A change of state of one or more of these variables can be considered as an event. This change of state normally results due to execution of a program. A set or sequence of events that may have to be scheduled as a unit may be organized as a process. A service is provided by one or more composites running a set of processes. A system is constructed to provide a set of services using a network of resources.

The reason for calling the hierarchical aggregation rough categorization is that the aggregation relation is neither precisely nor quantitatively defined. In some cases, the distinction between an aggregate and its constituent is dependent on the choice of observation or level of analysis. For instance, a module may have a number of inner modules that may be hidden from other client modules.

Similarly, behind what may be considered as an event at one level, there can be a number of processes that are not visible at the given point of analysis. When an input variable of a component changes (an event occurs), the component is expected to compute an output value and change its output variables. In some cases, the time elapsed between change of the input and output variables at a component may be small enough or irrelevant for the purpose of analysis, that it can be ignored. Probably, this is a motivating factor for Liu et al to write “each component is viewed as a hardware-like device in which an output value can change instantaneously when input values change and all components operate synchronously rather than in sequence”, in her TFM based correctness and consistency analysis (Zhiying Liu et al., 2009).

This is one way of abstraction, ignoring what happens inside a component or on its lower face which may consist of a network of a large set of other components and many processes, and only studying what is visible for an external user component. However, in cases where the time elapsed between the change of input variable and change of output
variable is significant enough to allow the occurrence of other undesired events, such as missing of deadlines, race conditions, etc., reliability assessment cannot be correct if done based on such abstractions.

Therefore, the approach taken by this author differs from the above in that an event is considered as a change of state of a variable, which may be an output variable of a component and an input variable of one or more components, the change being instantaneous for either case. However, each component may take some computation time (or response time), to produce a response and make change to its output variable, whether working concurrently with others or not. This provides room to make analysis of timing issues which are found to be among the most important factors that affect the reliability of many systems, such as real-time systems, as was experienced in the case study which will be described in chapter 9 and 10.

This, however, does not mean that one has to study timing issues by considering events as small as those related to change of each single bit of information. A useful way of analyzing timing behavior has long been achieved through the theory of aggregation of variables which served as a basis for the development of the near-decomposability theory (Courtois, 1977). The theory, which was originally developed for performance analysis, can be used for studying timing and other properties, including correctness, for the purpose of reliability prediction.

In fact, hierarchical aggregation is a concept that, when applied in system design, helps us to master the complexity of software systems and perform multilevel analysis of reliability, performance or other qualities. By moving up and down the hierarchy, one can analyse the system from the required behavior of the services (i.e. observing the forest as whole instead of the individual trees) or analyze the state changes at the individual variables or the behavior in each of the individual programs (i.e. looking at the trees in the forest individually), respectively. Figure 7-2 shows a hierarchy of aggregates interacting or communicating with each other.
Figure 7-2: Hierarchical Aggregate Structure

Symbols

- Encloses group of strongly interacting resources that can be analyzed independently of other resources outside the group (aggregate).
- Aggregation of aggregates (Multilevel aggregation)
- Aggregative function that would give estimates for the aggregate failure rates and time behavior.

**Legend:**

- \( \lambda_{ij}, \mu_{ij}, R_{ij} \): \( j^{th} \) resource with \( i^{th} \) aggregate with failure rate \( \lambda_{ij} \) and service rate \( \mu_{ij} \),
- Communication rates between resources.
- \( f \): Aggregative function that would give estimates for the aggregate failure rates and time behavior.
Besides the hierarchical aggregation relation described above, there are various other structure functions that can be defined among different elements of a software system to describe its various aspects. The definition of these functions and their relevance in design for reliability predictability is discussed in the subsequent sections.

### 7.2.2 Type Structures

Typing is a classification mechanism of variables, connectors, component instances and modules according to their externally observable properties, permissible operations, storage requirements and functionalities. A type identifies a class of entities that have the same set of externally observable properties and permissible operations. This can then be used to define a relation between entity classifiers, which consist of names of (abstract) interfaces, modules, and data structures, and resource identifiers – consisting of variables and connectors.

Subtype is a class of entities that have a common base type, where every property and operation are identical with entities of the same base type, and have additional sets of properties and/or permissible operations that makes them different from their base type. Subtyping relations can be used to create type hierarchies. Similar to the definition given in (Zhiying Liu et al., 2009), a relation $R_{st}$ is a subtype relation if for two type elements, $A$ and $B$, we have:

$$(A, B) \in R_{st} \iff A \text{ is a subtype of } B$$

**Equation 7-1**

The subtype relation is reflexive and transitive. From here on, when we refer to the relation by default, we refer to its reflexive and transitive closure form.

The importance of defining types is to ensure the consistency of data and objects that is communicated across a network of components. Error can occur, if output data represented in one form is interpreted in another incompatible form by an input receiver. Similarly, an output value from one component that cannot fit into the input variable of another component can result in loss of information with severe consequences. Generally, the type structure is used to avoid such undesired events from occurring by restricting the substitutability of one variable for another, for only a class of variables.
Different implementation languages have their own type structure to be used for syntax analysis and error identification during program compilation. The need for type structure at system design and documentation level arises for the following reasons:

1. The interface and the modular structure, which are main outputs of the design process, define most of the type structure in the system.

2. The type structure can be extended to analyze the possible occurrence of input-output mismatch, not only among software components but also among other system elements – between software and hardware components, or between software and human operators.

3. Type checking in programming languages is mostly limited to checking the conformity of signatures of interfaces and programs which is basically syntax analysis. However, the interest in this thesis is semantic definitions of interfaces and hence a need for extension of the type system.

4. The development of a system generally involves making design decisions on the connectivity among all components, including hardware components as well as human users in a system not just software modules, and hence the type structure can be used for analyzing the consistency of such connections.

As will be seen in section 7.3.1.6 with input-output connection matrix, the type structure is one way of ensuring non-interfering behavior of connections, DRP imperative 2, or detecting the presence of inconsistencies.

### 7.2.3 Modular Structure

The first DRP imperative that was identified in 6.3 tells us to identify design mechanisms that support modular redundancy, which was found to be a similar objective to design for change. Hence, the principle of information hiding is the basis for the decomposition of a system into modules in DRP. A systematic application of this principle, allows identifying the modular structure of the system, which usually, but not necessarily, results in a hierarchy of modules, based on the ‘is-secret-of’ relation.

The modularization process influences reliability structure, as it helps to:
• separate modules that may not have to work together in certain system configurations,
• identify the minimal set of modules that must work together to keep a system or service operational,
• identify possible modular redundancies,
• identify support and communication modules that are used to create connections across different components.

The modular structure of the system may be described by one relational document, the module guide. The domain of the relation is Module and the range of the relation is System. The relation is part-of relation that needs to be based on the ‘is-secret-of’ relation, according to the information hiding principle.

The characteristic predicate for the Module Guide is given by:

\[ PF(M, S) = \begin{cases} 
    True, & \text{if module M is part-of system S} \\
    False, & \text{otherwise}
\end{cases} \]

Equation 7-2

Complex systems may give rise to hierarchy of modules where some modules will have sub-modules.

Emphasis is given on the application of the information hiding principle so as to be able to change modules, which can be used a means of providing modular redundancy, when found necessary. Of course, not every system may have redundancy requirement from the very beginning, but it can still be ready for redundancy from the very beginning, i.e., at design time by identifying changeable secrets and hiding them by possibly changeable modules.

### 7.2.4 Interface Structure

An interface consists of the set of assumptions modules can make about each other with respect to their input-output variables, communication protocols and transfer functions. Its definition may be given by the sets of inputs, the set of outputs, the protocol of communication and set of transfer functions where each output is described as a partial function of the history of all inputs that must be provided by implementing modules.
TFM is one convenient way of precisely specifying or describing the semantic behavior of interfaces.

With respect to reliability structure, interfaces play a critical role since they are the references against which the reliability of modules is measured. A correctly implemented module will always comply with the specifications of the interfaces it must implement by properly using the interfaces of its environment. Hence, any deviation of module behavior from its specification can be considered to be a failure of the module to satisfy its requirements. However, all compliance to specification does not imply 100% reliability since specifications of modules can include notifications about undesired events, such as inability to continue operation due to failure of inner parts of the component or reporting of inputs outside of the domain of the component. Besides, reliability analysis at system level, may always need to include unreliability (incorrectness) of the specifications themselves.

Additionally, interfaces determine the connectivity among various modules in a system. They can be designed to enable or disable possible connections among various components in the system. This basically determines the possible reliability structure that may exist among a network of components.

To support diverse component redundancy, and for component substitutability, interfaces need to have self standing existence in the system. This results in the need for defining various relations among themselves as well as with modules. Note that, giving interface self standing position is already common in component software (Szyperski, 2002) and in most programming languages of today. However, mostly, the definition of those interfaces is limited to syntax, mere listing of variable names and types, without giving the semantic behavior.

7.2.4.1 Sub-interfacing
One relation that exists among interfaces is sub-interfacing. A sub interface is a subset of an interface. Its relevance is in restricting dependency among components. In many, cases, a module may need to communicate its various input-outputs with different parts of its environment. In order to restrict the module-interdependencies, it is usually useful to use only a subset of its interfaces that are relevant for communication with different
parts of its environment. An interface, whose input-output variable set is a subset of another interface, is called a sub-interface. For any two interfaces $I_A$ and $I_B$, we can define the sub-interface relation $R_{si}$ as follows:

$$I_A \subseteq I_B \iff (I_A, I_B) \in R_{si}$$  \hspace{1cm} \text{Equation 7-3}$$

Since the interface structure can also be used for the classification of modules and connectors, one can consider the sub-interface relation to be a subset of the subtype relation, and formally state this as follows:

$$R_{si} \subseteq R_{ts}$$  \hspace{1cm} \text{Equation 7-4}$$

### 7.2.4.2 Polymorphism

Two or more interfaces can have identical type and name of input variables, output variables and communication protocols differing only in their transfer functions. Such interfaces may be called abstract interfaces. Different modules that implement the same abstract interface are replaceable by each other in a composite structure. However, such replacements create different systems or services that have a different input/output transfer function.

An example of pair-modules, implementing the same abstract interface, could be:

- a compression-decompression module with an encryption-decryption module both implementing a stream interface,
- a stack module and a queue module both implementing same container interface say with get and put access programs.

### 7.2.4.3 Diversified Redundancy

Two or more modules may also implement the same interface, i.e., including the same transfer function, where the variation among them is in the implementations which may include variations in internal components and algorithms. Modules that implement the same interface are also replaceable by each other but the replacement does not change the system input-output transfer function. Such replacements may be done as a means of improving reliability through diversified redundancy.
Examples of diverse modules that may be replaceable with each other for the sake of redundancy include two communication modules with a peripheral device that have the same interface for communication, but one uses a wireless channel while the other uses cable connection.

Design decisions on the interfaces and its structure are critical in implementing DRP imperatives as it will determine the presence or absence of diversified modular redundancies and the presence or absence of non-interfering connections.

7.2.4.4 Module and Interface Connection Matrices

The interface structure generally determines the interconnectivity among the various modules of the system. Thus, the module-to-module or interface-to-interface connectivity of a system may be defined via the relations between interfaces and modules.

A system described by a module-interface relation is a quadruple: $SI = (M, I, PR, RQ)$ Where

- $I$: set of interfaces (in vector form of size $m$).
- $M$: set of modules (in vector form of size $n$).
- $PR$ (short form for PROVIDES): is a relation from modules to interfaces ($n \times m$) matrix defined by the following equation:
  $$PR(M, I) = pr(M^T \times I) = [pr(m_j, i_k)] = [pr_{jk}]$$
  where
  $$pr(m_j, i_k) = \begin{cases} 1, & \text{if module } m_j \text{ implements interface } i_k \\ 0, & \text{otherwise} \end{cases}$$

  \textbf{Equation 7-5}

- Every module is a provider of at least one interface, which implies that each row of the PR matrix contains at least one cell whose value is 1.

- $RQ$ (short form for REQUIRES): is a relation from modules to interfaces ($n \times m$ matrix) defined by the following equation:
  $$RQ(M, I) = rq(M^T \times I) = [rq(m_j, i_k)] = [rq_{jk}]$$
  where
  $$rq(m_j, i_k) = \begin{cases} 1, & \text{if module } m_j \text{ uses interface } i_k \\ 0, & \text{otherwise} \end{cases}$$

  \textbf{Equation 7-6}
Both PR and RQ may be many to many relations. Two incidence matrices from Modules to Interfaces may represent these two relations where the entry of a cell takes the value 1 if the characteristic predicate is true and 0 otherwise.

Alternatively, both Requires and Provides relations may be expressed by a single directed or undirected incidence matrix obtained as follows:

\[ A_d = PR(M,I) - RQ(M,I) \]
\[ A_u = PR(M,I) \lor RQ(M,I) \]

**Equation 7-7**

A module connection matrix may be defined as a ternary product \( \otimes \) defined in 5.7.1.2 of the PR matrix and RQ\(^T\) matrix.

\[ Q_f(M,M) = (PR(M,I) \otimes RQ^T(M,I)) = [q_{i,j}] \]
\[ q_{i,j} = \bigcup_{k=1}^{m} I_{kj} \otimes (pr_{i,k} \times rq_{j,k}) \]

where

\( I_{kj} \) is the \( k \)th interface provided by the \( i \)th module and required by \( j \)th module

**Equation 7-8**

This matrix may be considered as system framework. It shows the possible sets of modules, interfaces and their possible interconnections. When viewed as a graph, its vertices represent the modules in the system, and its edges represent the possible interconnections among the modules. From this graph, a module may be viewed as a mapping function from a set of interfaces (required) to a set of interfaces (provided).

The module connection matrix can be used to derive some basic system characteristics such as connectivity among modules and completeness of module requirements in terms of interfaces in a given environment.

Given a module connection matrix \( Q_f \) of a system, let

\[ RQ^{sf} = RQ(M,I) \otimes I = [rq_{i}] \]
\[ rq_{i} = \bigcup_{k=1}^{m} I_{k} \otimes rq_{i,k} \]

**Equation 7-9**
and
\[ RQ_i^Q = \left[ \bigcup_{i=1}^{n} q_{i,j} \right] = Q_i^T \otimes I_n^C \]

where \( Q_i^T \) is the transpose of the module connection matrix
\( I_n^C \) is an nx1 column vector consisting of all 1’s.

**Equation 7-10**

\( RQ_i^M \) is a vector consisting a set of interfaces required by each module from its environment and \( RQ_i^Q \) is the set of interfaces provided to each module by its environment as described by the connection matrix \( Q \).

The set of incomplete modules, modules for which the connection matrix does not provide all the required interfaces, if exist, may be obtained as follows

\[ IM_Q = \bigcup_{j=1}^{n} RQ_i^M \setminus RQ_i^Q \]

**Equation 7-11**

where ‘\( \setminus \)’ is the standard set difference operator.

Alternatively, we can obtain the incomplete modules as follows. First obtain the interfaces that are not provided by any module as follows:

\[ NPI = \neg PR^T \cdot I_n^C \]

where \( PR^T \) is the transpose of the provides matrix
\( I_n^C \) is an nx1 column vector consisting of all 1’s.

**Equation 7-12**

The set of incomplete modules is then given by the vector given below whose cell entries are 1:

\[ IM_Q = RQ \cdot NPI \]

**Equation 7-13**

Additionally, if the provides and requires matrices also include all user components in the system, the set of irrelevant modules, modules which provide interfaces that are not used by any of the components in the system can be obtained as follows:

First obtain non-required interfaces, NRI, and then find the set of unused modules.
\[ NRI = -RQ^T \cdot 1_n^C \]

where \( RQ^T \) is the transpose of the required matrix \( 1_n^C \) is an nx1 column vector consisting of all 1’s.

**Equation 7-14**

\[ UM = PR \cdot NRI \]

**Equation 7-15**

Similar to the module-connection matrix, an interface connection matrix may be defined as given below:

\[
Q_M(I, I) = (RQ^T(M, I) \otimes PR(M, I)) = [q_{i,j}]
\]

\[
q_{i,j} = \bigcup_{k=1}^{m} M_{kij} \otimes (r_{k,j} \times pr_{k,j})
\]

**Equation 7-16**

The interface-connection matrix may also be viewed as a graph in which the vertices represent interfaces and the edges represent the modules.

### 7.2.5 Service-Module Dependency Matrix:

A service, in this thesis, is a set of independently usable systems functions with distinctly identifiable input-output variables and specified sets of input-output mappings. A service thus makes the primary choice for defining reliability functions. Its requirements, which are assumed to be specified by a set of interfaces, are realized through a composition of modules. Therefore, the set of service interfaces are just a subset of the interfaces provided by the modules defined in the system.

If \( Q \) is a considered as a graph describing an interface connection of modules, then the connection structure of any particular service provided by a system must be described by a connected graph \( H \) which is a sub-graph of the system connection graph \( Q \), \( H \subseteq Q \), where \( IM_{H}=\emptyset \). The last restriction is included because, for any service to have a predictable behavior, every module used in the service must have at least one provider for each of the interfaces it requires from its environment.

The reliability of each service must be defined distinctively for the following reasons:

1. When systems are composed from a set of information hiding modules connected through abstract interfaces, it is possible to create many different services by
changing only some of the modules. The reliability of each of the services is mainly a function of the reliabilities of the modules that are included in the service.

2. In some cases, certain modes of operation of modules may be acceptable for some services but not for others. As a result, two or more services that have almost similar modular structure can have different reliability structure. For instance, a streaming service may use modules that implement lossy compression algorithms for transmitting video and audio signals whereas such modules would be unacceptable for transmitting device control signals or other messages where the loss of even a single bit completely alters the message.

3. Depending on the nature of the application, different services may have different levels of reliability requirements. Some may be considered to be critical whose failure may result in catastrophic conditions while others may not be considered critical. For a system that provides a prioritized set of services, the system may be considered to be in degraded but not failed state so long as those of high priority services are functional even if some of the low priority services are in failed state.

4. In applications that require increased reliability through redundancy, different services having different input/output behavior and modular structure could be a better way of getting redundancy with reduced correlated failures. Traditionally, redundancies based on functional dissimilarity have been used in safety critical systems for improved redundancy.

The basis for defining the reliability structure of services is a service-module dependency matrix, $S$. This may be represented by a path matrix which consists of a binary matrix where the rows represent different possible paths of services and the columns represent the modules used by the services in each path.

### 7.2.5.1 Deriving Service-Module Dependency Matrix from RQ & PR Relations

The RQ and PR relations can be used to derive the service-module dependency matrix. For the procedure described below, the following assumptions have been made about RQ and PR relations:

---

$^3$ Paths in here refers to the sets of modules (not program statement) used by services
• If a module requires more than one interface, it is assumed to be a case of dependency in which all interfaces have to be provided in any given path for the module to have a predictable behavior.
• However, if an interface is provided by more than one module, the different provider modules are considered to offer alternative paths, which may be due to modular redundancy or polymorphism.
• Additionally, the system is assumed to be complete with respect to the module’s requirements in the system, i.e. the set of incomplete modules is empty set. The service-module dependency matrix can then be obtained by identifying the closure set of the provider modules for a set of required interface as described below: (In some of the following notations, set of items are represented by their characteristic vector which is a Boolean vector and the set operations are converted to Boolean operators accordingly).

1. Identify the set of interfaces, $I_S^0$, that specify the requirements of each independently usable service. This may be represented in a vector form which contains value 1 for those interfaces in the set and 0 for others.
2. Create a tree $T$, whose $n^{th}$ node contains $\{I_n^0, M_n^T, I_p^T, M_p^T\}$, where $I_n^T, M_n^T$ represent the required interfaces and provider modules identified at node $n$, and $I_p^T, M_p^T$ represent those that have been identified at node $n$ and in all its ancestor nodes.
3. Start with a root node containing $\{I_S^0, \emptyset, I_p^0, \emptyset\}$ representing the set of interfaces and set of modules identified for the first node.
4. For each node in $T$, find the provider modules of the required interfaces in the node. Since there can be more than one provider module for each interface giving rise to alternative paths,
   4.1 find all provider modules for the interface
   4.2 remove those that have already appeared in ancestors nodes.
   4.3 evaluate the Cartesian product of the modules sets for each interface in order to obtain alternative paths in which every required interface has at least one provider module in the set.
$$M = \{M_{n+1,i},...,M_{n+1,m}\} = \prod_{m \in \mathbb{I}_n} (PR[I]^\wedge \neg M_p^T) \otimes 1^C_n$$  
Equation 7-17

where $M_{n+1,i}$ is a set of modules that provides all the required interfaces at the node $M_p^T$ the characteristic vector corresponding to the modules identified in ancestor nodes. $PR[I]$ represents column $I$ of the matrix $PR$ $\wedge \neg$ when used on characteristic vectors is equivalent to the set difference operator $\setminus$ $\prod$ represent the Cartesian product operator on sets

4.4 Create children nodes to the current node corresponding to each of the elements in set $M$. For each child node, the node module set is $M_{n+1,i}$ and the set of required interfaces in the node is obtained as:

$$I_{n+1}^T = RQ_n^T \left( m \in M_{n+1,i} \right)^\wedge \neg I_p^T.$$  
This is because, every interface required by each of the modules in the node must be included as required interface but the interfaces already encountered in ancestor nodes need not be included, since their provider modules are already being included. Then, update the path module and interfaces sets as follows:

$$M_p^T = M_p^T \vee M_{n+1,i}^T$$

$$I_p^T = I_p^T \vee I_{n+1}^T$$

5. Repeat steps in 4 long as the set of required interfaces at each node is different from empty set.

6. At the end of step 5, the leave nodes of the traversal tree contain the different possible paths for the specific service under consideration. The number of leave nodes corresponds to the number of paths, and the path module vector contains the modules used for that specific path.

7. Repeating steps 1 to 6 above for all services provided by the system results in a forest where the leave nodes contain the possible paths for the various services.

8. The service module dependency matrix can then be obtained taking the path module vectors of each leave node as a row vector in the matrix.

Example:

Consider a system composed of four modules \{M1, M2, M3, M4\} and two interfaces \{I1, I2\} with module-interface relations given by the following matrices. Let M1 and M2 provide interface I1. Let modules M3 and M4 require interface I1 and provide interface I2. Let I2 be externally accessible to the users of the system.
To find the path matrix, we first start with a tree $T$ having a root node of \{12, $\phi$, 12, $\phi$\}

$$M = \{M_{12}^{T}, \ldots, M_{n+1,m}^{T}\} = PR[12]^{T} \otimes I_{n}^{c} =$$

\[ \begin{bmatrix}
M_{1} & M_{2} & M_{3} & M_{4} \\
0 & 0 & 1 & 1
\end{bmatrix} \otimes 
\begin{bmatrix}
1 \\
1 \\
1 \\
1
\end{bmatrix} = \{M_{3}, M_{4}\} \]

Find the required interfaces by each of the elements of the identified module set.

$$I_{1,1}^{T} = \bigvee_{m \in M_{12}^{T}} RQ^{T}[m] \wedge -I_{p}^{T} = RQ^{T}[M_{3}] = \{I1\}$$

$$I_{1,2}^{T} = \bigvee_{m \in M_{12}^{T}} RQ^{T}[m] \wedge -I_{p}^{T} = RQ^{T}[M_{4}] = \{I1\}$$

Then, create two children nodes with contents given by $c1=\{I1, \{M3\}, \{I1, I2\}, \{M3\}\}$ and $c2=\{I1, \{M4\}, \{I1, I2\}, \{M4\}\}$

Repeat the above procedures for the children nodes at $c1$ and $c2$. In both cases, the required interface is $I1$. The provider modules for $I1$ are $\{M1, M2\}$. Both $M1$ and $M2$ have no other required interfaces. This gives rise to the following child nodes.
Since all the required interfaces on the leave nodes are empty set, we have identified all the paths. In matrix form, the different paths may be shown as follows:

\[
S(P,M) = \begin{bmatrix}
P1 & 1 & 0 & 1 & 0 \\
P2 & 0 & 1 & 1 & 0 \\
P3 & 1 & 0 & 0 & 1 \\
P4 & 0 & 1 & 0 & 1
\end{bmatrix}
\]

Where the 1st column and row, containing P’s and M’s is given as headers or indices of the matrix.

7.2.5.2 Deriving Service-Module Dependency Matrix from Interface Connection Matrix

If each of the rows of the RQ relation have at most 1 non-0 value or the various entries of each row are alternative paths in which a module requiring more than one interface actually works with any of the interfaces it is provided with, then the interface connection matrix can be used to obtain the service-module dependency matrix as described below.

The goal in here is to derive the service-module dependency paths as a transfer function from the set of required interfaces to the set of provided interfaces. For the system of equations to be well defined, every module must have not only provided interfaces but also required interfaces in the matrices. For this, a dummy interface I₀ can be added to the interface set to which an RQ relation can be defined with every module that does not have any other required interface:

Given a set of required interfaces, \( I^n_S \) the set of provider modules may be obtained as:

\[
M^n = PRM, I)I^n_S
\]

Equation 7-18

These set of provider modules may have its own new required interfaces. Thus, the set of required interfaces need to be updated by adding the required interface of the modules.

\[
I^{n1}_S = RQ(M,I) ∩ M^n ∨ I^n_S
\]
Equation 7-19
Substituting Eq. 15-3 in 15-4, we have

\[ I_s^{n+1} = R \bar{Q} (M, I) \otimes (PR(M, I) I_s^n) \lor I_s^n \]

Equation 7-20
Since the ternary operator (\( \otimes \)) is basically a matrix product operator and satisfies associative property, Equation 7-20, can be re-written as follows:

\[ I_s^{n+1} = ((R \bar{Q} (M, I) \otimes PR(M, I)) I_s^n) \lor I_s^n \]

\[ I_s^n = Q_M (I, I) I_s^n \lor I_s^n = (Q_M (I, I) \lor I_m) I_s^n \]

where \( I_m \) is an mxm identity matrix.

Equation 7-21
Equation 7-21 is a recursive equation which identifies the possible paths among the various interfaces. The matrix product is similar to the standard matrix product except that the standard sum is replaced by a logical ‘or’ (a set union) operation and the product is replaced by a concatenation or logical ‘and’ operation. The equation is similar to those used for evaluating switching functions (Deo, 1974).

The set of module paths that connect any two interfaces can be obtained by solving the above equation which is given as \((Q_M (I, I) \lor I_m)^{m-1}\).

\[ T = (Q_M (I, I) \lor I_m)^{m-1} \]

can be considered as the transmission matrix between the various interfaces in the system. An entry \( t(i,j) \) in \( T \) contains, the set of possible paths between interface \( i \) and \( j \) where each path is a composition of modules that have to work together to map the required interface \( i \) to the provided interface \( j \). The entries from \( I_0 \) to any service interface \( I_s \) in the transmission matrix provide the service paths.

For the example described earlier, we have:

\[
RQ(M, I) = \begin{bmatrix}
1 & 0 & 0 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
\]

\[ I_0 \quad I_1 \quad I_2 \]

I0 1 0 0
I1 1 0 0
I2 0 1 0
I3 0 1 0
\[
PR(M, I) = \begin{bmatrix}
M1 & 0 & 1 & 0 \\
M2 & 0 & 1 & 0 \\
M3 & 0 & 0 & 1 \\
M4 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
Q_M(I, I) = (RQ^T(M, I) \otimes PR(M, I)) = [q_{i, j}]
\]

\[
q_{i,j} = \bigcup_{k=1}^{m} M_{kij} \otimes (r_{k,j} \times pr_{k,j})
\]

\[
Q_M(I, I) =
\begin{bmatrix}
I0 & 0 & 0 \\
I1 & 0 & 0 \\
I2 & 0 & 0 \\
\end{bmatrix}
\otimes
\begin{bmatrix}
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

\[
Q_M(I, I) =
\begin{bmatrix}
I0 & 0 & M1 + M2 \\
I1 & 0 & M3 + M4 \\
I2 & 0 & 0 \\
\end{bmatrix}
\]

\[
Q_M(I, I) \lor I_m =
\begin{bmatrix}
I0 & 1 & M1 + M2 \\
I1 & 0 & M3 + M4 \\
I2 & 0 & 0 \\
\end{bmatrix}
\]

\[
(Q_M(I, I) \lor I_m)^2 =
\begin{bmatrix}
I0 & 1 & M1 + M2 \\
I1 & 0 & M3 + M4 \\
I2 & 0 & 0 \\
\end{bmatrix}
\otimes
\begin{bmatrix}
1 & M1 + M2 & 0 \\
0 & 1 & M3 + M4 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

\[
(Q_M(I, I) \lor I_m)^2 =
\begin{bmatrix}
1 & M1 + M2 & (M1 + M2)(M3 + M4) \\
0 & 1 & M3 + M4 \\
0 & 0 & 1 \\
\end{bmatrix} = (M1+M2)(M3+M4) = M1M3+M1M4+M2M3+M2M4.
\]

\[
t(I0, I2) = (M1+M2)(M3+M4) = M1M3+M1M4+M2M3+M2M4.
\]
Therefore, possible path sets from I₀ to I₂ are \{M₁M₃, M₁M₄, M₂M₃, M₂M₄\}

Alternative to computing \((Q_m(I, I) \lor I_m)^{m-1}\), a flow graph model can be used to derive the transfer function from I₀ to any service interface I₅. Below is a flow graph model of our example.

![Flow-graph representation of an interface connection matrix.](image)

The set of possible paths that connect I₀ to I₂ are
\[\Phi = \{M_1.M_3, M_1.M_4, M_2.M_3, M_2.M_4\}\]

The above structure in which parallel modules exist is given as an example only. In most cases, modules may have common interface at one side but different interfaces on other sides. As a result the possible set of paths represent the modules required by a service may have a cardinality of 1.

### 7.3 Dynamic Structure Functions

The connection matrices discussed in the previous section (the interface-connection matrix or the module-connection matrix) are static descriptions of a system structure. The runtime behavior of systems that is necessary to make in-depth reliability analysis is not sufficiently described by these matrices. This is because at run time, a module can be used to create or instantiate a number of different component instances that are interconnected to various component networks to provide different services. The reliability of each component instance not only depends on the behavior of the creator modules but also on the behavior of its neighborhood, which may vary from service to service. This makes it necessary to study the runtime configuration of a network of component instances that appear together while providing one or more services.

In the sections below, the author introduces and discusses two dynamic structures, the input-output connection matrix and the event-flow equations.
7.3.1 Connector Structure

At runtime, module interfaces appear as connectors that link various components together. These connectors are nothing but variables which are uniquely identifiable resource locations. Whether it is data or message passing, procedure call, function return, event notification, etc., it is all done by one program component from a module setting the values of some variables from where another program possibly from a different module reads.

While such level of communication is what normally happens in the software product, it does not give enough room to study important design imperatives that improve the reliability structure of software systems. Hence, connectors are discussed at a separate level as a set of input-output variables that are created by modules for communication and coordination among each other. There are various types of connectors among which include:

- **Call Stacks**: most procedural abstractions that have been the basis for structured design methods, and existing object-oriented programming languages use program call stacks as a major connection mechanism between different programs across modules.

- **Objects/instances**: these are sets of input/output variables that are created and destroyed in the memory system as a unit by a specific module for communicating its inputs and outputs to other modules in the system.

- **Connector components**: while every module may be used to create instances for providing its service, some modules may exclusively be designed to provide connection/communication service among other components in the system. Examples of such connectors include event channels/synchronizers, communication channels, queues or other container components from which inputs are received and to which outputs may be placed.

- **Primitive variables (data objects) and files**: these are passive entities on to which modules can write and from which others read.

The different types of connectors have different properties that affect the reliability structure of software systems, as shall be discussed in the following subsections.
7.3.1.1 Component Cooperation and Interferences

Connectors differ in the number of components they support for communication and cooperation at a specific point in time. Two categories may be identified with respect to this property:

- Binary connectors – connectors that can connect only two components at a time. Examples of binary connection includes procedural calls where a program component pushes its input onto a stack from where the called program reads and the results are pushed back onto the stack for the caller to read it from.

- N-ary Connectors - connectors that can possibly connect more than two components at a time. Objects and connector components can be considered to be n-ary connectors. Any component that is given access to these connectors can provide inputs to or read outputs from them. The connection and communication can involve more than two components at a time and the direction of dependencies can be restricted to comply with certain interface definitions.

This property has an important implication in relation to the organization of components to satisfy a given reliability structure. In many cases, one may need to include different components in a system specifically aimed at improving system reliability. Examples of such components include fault detectors or component/system health monitors, protective components, backup and restorative components, etc.

Including such components as part of critical system components can make the system components complex and hence error prone possibly reducing system reliability. Including the components in a linear sequential path, as provided by binary connectors, between critical system components, can make the system directly dependent on these components in that their failure can cause failure of critical system components and hence the system. The use of n-ary connectors that allow the connection of more than two components at a time where each component works with its own thread of control, can result in a structure where the critical system components are supported by reliability improvement components with reduced interference.
7.3.1.2 Fault-tolerance, Restorability and Failure Dependency

Although all connectors may be categorized to be shared variables, there are variations in their properties that imply different failure dependency among components. Some connectors have coupling effect that makes all connected components to appear in series connection in their reliability structure, in at least some of their failure modes, irrespective of the reliability role that is supposed to be played by the components.

For instance, connection through transfer of a single thread of control, which is still among the predominant types of connecting software components, makes all components in a composite structure to appear in series connection mode, in that failure of any of the components to pass the execution control to the next component may result in failure of the whole set of components. Such type of connections can make even supposedly redundant or supportive components induce failure to critical system components.

On the other hand, if communication is done through connectors that do not force transfer of control, then failure of any component can affect only those components that are specifically designed to depend on its output.

Consider the case where A connects to B through its abstract interface but without transferring control so that it remains active to observe B’s response. Then, A has the chance to detect some classes of failure of B, such as non-responsiveness. This provides a chance of restoration of component B, although its replacement is only possible if there is a redundant component, preferably diverse, to component B that implements the same abstract interface. In cases where A is not the direct user of B’s output, or B’s output is of not critically important for the service being provided, component A can continue providing with its own service even after the failure of B, giving a chance for fault-tolerance or service degradation instead of service disruption.

Fault-tolerance and component restorability can further be improved by connecting components through an independent connector that can be attached and detached to the components as necessary. Consider the case A and B, both designed to work with a connector having interface type C as their input/output channel. At a time when the two components need to be connected, C may be instantiated together with A and B, and attached to both components. Both components can then work on their own speed
communicating indirectly through C. Other supportive components can also monitor the process by connecting to C. In case of occurrence of undesired events detected by any of the three components A, B, C or other support components that monitor the process, the failed component (which can be any one of the three components) can be replaced by other compatible redundant components or new instances. In this case, the restoration applies for all components A, B and C, and the requirements for redundant components to A and B are provision of compatible functions and ability to connect to C. If C is a container for the data/objects being processed by components, it means that the data/objects can be transferred to other redundant components that may takeover the task of the failed components.

7.3.1.3 Protection from Interaction Failure Causes
Connection of multiple components at a time can be a desired property for coordination of various components as well as for fault detection, monitoring and component restoration. However, if multiple components that concurrently execute share states that may be modified as part of the execution process, then interaction failures such as race-conditions, deadlocks, or similar undesired events can occur, unless appropriate protection mechanisms are put in place.

The possibility of encountering interaction failures or preventing them from occurring is dependent on connector properties. Passive connectors such as files and primitive variables do not support coordination among their user components on their own and hence may allow the occurrence of interaction failures if they have to be modified by more than one component at a time. Components sharing such passive connectors must employ their own synchronization mechanisms among themselves so that they do not interfere with each other. However, implementation of synchronization mechanisms in every system module can make the modules more complex and error prone. In order to keep the main system components simple and so as to reuse the protection mechanism throughout the system, one convenient way of putting protections against some known undesired events is through connectors.

Call stacks avoid race conditions by forcing all schedulable events to occur in sequential order. But their support for other properties, such as cooperation of more than two
components can be poor. On the other hand, other connector components, such as event and communication channels can allow the inclusion of protective services that provide synchronization, event notification, queuing and other relevant services.

7.3.1.4 Undesired Event Communication and Management

Another reliability concern that is affected by the type of connectors used is the communication and management of undesired events.

Binary connectors that work with control flow logic restrict the communication and handling of undesired events to occur between two components at a time. Such connectors usually force a service provider component that was executing a given task to interrupt its execution when encountering some undesired event and return the control to its client component. There are many limitations associated with such type of undesired event communication and management, among which include:

1. Because of the interruption of execution, the server component cannot continue doing some other subtask related to the given task while the reported undesired event is handled by other components. Resumption of execution from where it was interrupted is usually difficult if not impossible.

2. Although there are only two components communicating through the binary connector, the component that has put the task on the connector may not actually be the main client component that consumes the results of the computation. As a result, if it receives the undesired event information, it may not have the necessary handler for the specific undesired event. This may result in unhandled undesired events that could lead to component and system failure.

3. To solve the above problem, it is possible to force all components to have undesired event handlers that may arise in any possible scenario. However, this can make each of the modules complex and error prone. Some components may have an empty handler that simply hides the undesired event and denies others which could have taken the correct response. To avoid these problems, it may be better to design special modules for the detection, management and handling of undesired events. This requires connectors that can connect multiple components at a time.
4. In environments where the communicated information is subjected to interpretation (such as type conversion) by receivers, mixing of computed results with failure information on the same connector can result in misclassification that can have severe consequences. An example of system failure caused by the misinterpretation of error data (information that was communicating failure) as correct input data is what happened in Arian 5, “Part of these data at that time did not contain proper flight data, but showed a diagnostic bit pattern of the computer of the SRI 2, which was interpreted as flight data.” (Lions, 1996),

The use of n-ary connectors can overcome some of the limitations listed above. Components can be designed with separate connectors for receiving inputs, providing outputs, and communicating undesired events. The connectors can be shared among a selected number of components that may be assigned to collaborate in the handling of undesired events.

7.3.1.5 Combining Connector Types
The above section shows the implication of different connectors in affecting the reliability structure of software systems. The connectors basically determine the failure dependency among various components.

Any design may include a combination of various connectors in the system. Passive connectors can be used without any problem so long as the maximum number of writers at any point in time is restricted to one, and the proper sequence between writing and reading events is ensured by some other mechanism, for instance through process structures. Programs within a module may call each other through call stacks. Modules may be used to instantiate various objects and connectors to form a composite structure. Components in a composite structure may be connected through independent connectors that also have protective mechanisms against possible interaction failures.

However, all such design decisions must be (and can be) done without compromising the simplicity of designs. This is an important criterion for improved reliability prediction. One of the main lessons learnt from the success of the information hiding principle is its ability to hide complex interactions behind simple abstract interfaces. A composite, constructed through a composition of components and connectors, with many of the
features discussed above can have a simple external interface when used by other components in the system since most of the components and their connections remain hidden.

Nevertheless, since the connection structure of composites determines the reliability of the service they provide, the structure must be mathematically analyzed. For this, an input-output connection matrix is introduced and discussed below.

7.3.1.6 Input-Output Connection Matrix
A set of component instances forming a network and behaving as a unit is called a composite. The connection matrix formed from the input-output variable connection of the components in the composite is called an Input-Output Connection Matrix. This matrix can be derived from module-component, module –interface and interface-variable relations as follows.

Let C be the component instance vector created by a set of modules M, and let V be the variable Vector interconnecting the components in the system. The behavior of each component instance is derived from its creator modules and the connection must conform to interface specifications given in TFM.

From the TFM specification of each interface, we can obtain the following incidence matrices on a set of interfaces I and variables V:

\[ IV(I,V) = iv(I^T \times V) = [iv(i_j,v_k)] = [iv_{i_k}] \]

where

\[ iv(i_j,v_k) = \begin{cases} 
1, & \text{if variable } v_k \text{ is an input variable in interface } i_k \\
0, & \text{otherwise} 
\end{cases} \]

Equation 7-22

\[ OV(I,V) = ov(I^T \times V) = [ov(i_j,v_k)] = [ov_{i_k}] \]

where

\[ ov(i_j,v_k) = \begin{cases} 
1, & \text{if variable } v_k \text{ is an output variable in interface } i_k \\
0, & \text{otherwise} 
\end{cases} \]

Equation 7-23

For each component instance C, created by module M, its universal set of input (output) variables is obtained from the set of input (output) variables of the implementer module and the set of output (input) variables of the client and/or server modules in its neighborhood, respectively. Thus, we can write:
\[ IV(C, V) = PR(M, I) \times IV(I, V) \lor RQ(M, I) \times OV(I, V) \]
\[ OV(C, V) = PR(M, I) \times OV(I, V) \lor RQ(M, I) \times IN(I, V) \]

Equation 7-24

The Input-Output Connection matrix can then be obtained as:

\[ Q_{io} = (OV(C, V) \otimes IV(C, V)^T) = [q_{i, j}] \]

\[ q_{i, j} = \{ v | v \in OV(C_i) \land v \in iv(C_j) \} \]

Equation 7-25

The above statement implies that an input variable of one module is an output variable of another module and vice versa. There are obvious design imperatives that are implied by the statement. That is, for each variable that interconnects two components, any one of the following is possible:

- One shared variable that acts as an output variable for one component and input variable for another (one or more) component,
- An input variable that is part of one component but is connected to another component as an output variable at a specific point in time
- A connector that transmits signals from the output variable to the input variable.

In all cases, the two components may have separate names for the variables, but an important restriction for predictable system behavior is that their types must be compatible, i.e. the type of an input variable must be a sub-type of the type of an output variable. The type structure may be updated to reflect the input-output behavior of the components by including or excluding tuples into the relation.

For instance, if a device expects an input key sequence to be in alphanumeric combination but the output of an other device is restricted only to numeric digits, then, the two devices are incompatible and this fact may be stated through exclusion of (alphanumeric, numeric) pair from the subtype relation, i.e.,

\[(alphanumeric, numeric) \notin R_t\]

Equation 7-26

Similarly, if we have a 64 bit register A and 32 bit register B, in a system, we may allow the data to be copied from the 32 bit register to 64 bit register, but not vice versa. Thus, the subtype relation may contain (A, B) but not (B, A) indicating that transfer of output
from B to A is acceptable but not vice versa, unless the data in the 64 bit is known to be limited to the first 32 bits that can be copied to the B register safely.

Given an input-output connection network, where each component’s input-output type is defined precisely, the consistency of the network or the compatibility among all the components in the network can be checked by applying the type-subtype relation, $R_{ts}$, on the input-output connection matrix $Q_{IO}$ as follows:

$$TY(Q_{IO}) = [ty_{i,j}]$$

$$ty_{i,j} = (q_{i,j} = \{\}) \text{ } \cap \text{ } \prod_{v \in q_{i,j}} R_{ts}(Type (v(iv(C_j))), Type (v(ov(C_i))))$$

where

$$q_{i,j} = \{\}$$ and $$R_{ts}(Type (v(iv(C_j))), Type (v(ov(C_i))))$$

are logical operations that evaluate to either true or false. $\prod$ evaluates the Boolean conjunction of all operands.

**Equation 7-27**

Type compatibility for network of components can be obtained by evaluating the above matrix, i.e.

$$TY(Q_{IO}) = 1 \Rightarrow \text{Compatible}$$

$$TY(Q_{IO}) \neq 1 \Rightarrow \text{Incompatible}$$

Where 1 is a matrix of all ones.

**Equation 7-28**

The implication of the above set of equations is that, if components are communicating with each other, then the input received by one component should be a subtype of the output provided by another.

Another important significance of the input-output connection matrix is in showing the object/data/signal flow in a network of components. It shows the input received by each component including the source of the input, and the outputs provided to other components.

Given an input-output connection matrix, all the inputs and outputs of each component can be obtained as follows
\[ O_c^Q = \bigcup_{j=1}^{n} q_{i,j} = Q \otimes 1_n^T \]
\[ I_c^Q = \bigcup_{i=1}^{n} q_{i,j} = Q^T \otimes 1_n^T \]

where \( Q \) is the connection matrix, \( Q^T \) is its transpose, 
\( 1_n^T \) is an nx1 column vector consisting of all 1’s.

**Equation 7-29**

\( O_c^Q \) is the set of all outputs of each component in the network and \( I_c^Q \) is the set of all inputs received by each component in the network. One question that these two sets help to answer is whether the set of inputs to each component is complete enough to produce the set of outputs from the components in the network.

Given behavioral component documentation such as TFM for each component, then it is possible to obtain the possible set of inputs that produce the outputs given by \( O_c^Q \). Let \( I_m \) be a function that produces the set of minimal\(^4\) inputs that are required to produce a given output from a component (based on component behavior documentation). Applying to the output vector, \( O_c^Q \), we get

\[ I_c^M = I_m(O_c^Q) \]

where the vector operation implies that the function is to be applied for each component independently.

Note that the vector \( I_c^M \) will generally consist of a set of minimal inputs correspond to each vector. The outputs \( O_c^Q \) in the component network are feasible if

\[ \exists I, I \in I_c^M \land I \subseteq I_c^Q \]

**Equation 7-30**

This test, however, is only a necessary condition for the outputs to be feasible outputs. This is because the connection matrix does not provide additional information on the ordering of the inputs (or the events) that are the actual determining factors for the outputs from each component.

\(^4\) An input set for producing a given output is said to be minimal if any of its proper subsets cannot produce the required outputs.
Yet, the fact that the input-output connection matrix shows the precise interdependency of components that provide a given service makes it to be one major source of information for deriving the reliability structure of services. The matrix can be used as a dependency graph based on which reliability formulas for estimating the reliability of connecting any selected set of input-output variables can be formulated.

7.3.2 Event and Process Structures

An event is a change of the state of one or more variable in a system at a specific point in time. In a connected system, one or more components can respond to an event. Their response may be done by reading some values from input variables, compute results, update their internal states and writing onto their output variables. This in turn gives rise to other events which may involve different sets of components. The process may continue until certain designated set of tasks are accomplished, a process terminating event occurs, or in some cases indefinitely, for unlimited period of time.

As a result, many events are casually related to each other, where an event at one component is a cause for the occurrence of another event in another component. Some of the events in the computer system occur due to changes in the environment, including passage of time. A subset of the events corresponds to conditions in which one or more components in a system become unable to function properly. This may be caused by failure of components or unavailability of some required resources to perform a given task. Such events are called undesired events and their occurrence may result in the partial or complete loss of some system functions unless an appropriate response averts the risk.

7.3.2.1 Partitioning of Events into Process

Many of the events that occur in a system result from program executions and hence their occurrence order may usually be pre-defined by design decisions, or in some cases depends on runtime configuration. Some events have to be ordered sequentially so as to get the required state transitions of the underlying state variables. Others may be allowed to occur in parallel or in some arbitrary order. Some events may have to occur together always, and others may only occur together occasionally.
In complex system environments that involve a large number of components, the event sets may need to be organized into separate nearly independently schedulable units called processes. Processes may thus be considered to be subsets of the event set. But in some cases, the distinction between events and processes is a function of the level of observation. What may be considered as an event and response at one component may internally involve a number of events or even processes that may be hidden from the level of analysis being considered.

The main objectives partitioning of events into processes are for scheduling or allocating resources, prioritization of tasks, fault-tolerance, improving performance as well as reliability. Though, in many programming languages process partitioning is associated with memory allocation, this is not a requirement. There can be more than one process that performs the tasks of any given component instance as there can also be processes that interconnect a number of component instances. A process is a unit of scheduling and may consist of a sequence of events. The sequencing of events and the state of a process may be considered to be controlled by some control variable, say an event or program counter.

The partitioning of events into processes has a significant implication on system reliability. A large class of system failures arises due to interaction failure causes. These are undesired events that may cause system failure even when each of the components performs according to their specification. As will be seen later, the possible occurrence or non-occurrence of these interaction failure causes is dependent on the organization of input-output variable as well as the event structure.

Additionally, the process structure determines many reliability related properties such as prioritization, where certain processes are given higher priority in resource allocation than others, fault-tolerance, where the failure of some processes is tolerated, and dependency, where undesired events that block the successful execution of certain event sequences will result in dependent failures on those components that were expecting events from the interrupted processes.

For instance, the successful execution of some critical programs and hence the occurrence of their respective events, may be essential for the proper functioning of a system,
whereas other events may not be so essential in that their non-occurrence may simply result in system degradation but not failure. An obvious design decision would be to separate the two event-sets in to two processes and assign different priority levels.

One important criterion that needs to be considered in making design decisions on structuring of events into different parallel processes is the timing behavior of components. Components may greatly vary in their response time. System stability requires that each component’s demand rate to be lower than its execution rate or in other words the time between demands for execution should be greater than the component response time. Such stability requirements may be considered as reliability requirements since unstable systems can be unreliable. Specifically, race conditions may occur if the demand rate exceed execution rates for any component and no mechanism is put in place to synchronize the two.

Additionally, some states of components may be time sensitive in which certain actions have to be completed, meeting a given deadline. Missing of such deadlines may be an undesired event that may result in component and/or system failure. Thus, the organization of events into different processes and their prioritization with respect to scheduling or resource allocation can be done to maximize system reliability (or the reliability of selected critical system functions) by minimizing undesired events caused by race conditions and missing of deadlines.

In general, the event or process structure is one of the most important structures that affect system reliability, some of which are discussed below.

7.3.2.2 Fault tolerance
If all events in a system are structured to occur as one sequential process controlled by a single ‘program counter’, the chance of tolerating the failure of any component is very small. This is because failure of any component to return the ‘program counter’ in a given required state can cause failure to all other components in the system that share the same program counter. As a result, components connected in a sequential process sharing the same program counter can be assumed to be in series connection at least partially with respect to their fault modes that affect the program counter, irrespective of their functional structure or intended reliability role.
On the other hand, allowing components to work in separate process space, while communicating events and objects through shared connectors can improve failure independence among the components. This may not completely eliminate all failure dependencies since those components that are users of outputs of failed components can still be affected by the failure. But it provides the following advantages:

- Failure dependencies will be restricted only to those components that are dependent on the outputs of failed components, while other components that may be part of the composite structure but need not use the outputs of malfunctioning components can continue to operate. This helps to tolerate failure of some of the components, especially those that play protective and supportive roles.

- Monitoring the health of components and taking restorative or replacement action in case of occurrence of undesired events or failure can be achieved relatively easily when components work in different process space than on a sequential mode of operation.

In general, fault-tolerance and component restorability require the partitioning of activities into parallel communicating processes which does not necessarily mean separation of memory spaces. However, unsynchronized parallel processes can have their own reliability risks such as race conditions and deadlocks. To prevent failure due to such risks, the connectivity among different components can be made by using connectors that provide protective service against such conditions.

Consider a system of two components A and B where A plays a protective role if included but the system can still work without it albeit in a degraded mode. An obvious design decision would be to include component A in the system so long as it is properly working but to tolerate its failure by either excluding it or its output from the system if it is not working properly. In this case, the system reliability is what is given earlier in Equation 6-9 by

\[ R_S = R_A + (1 - R_A)R_B = R_A + R_B - R_A R_B \]

Equation 7-31

However, if the only way of connecting A and B is through a sequential process with shared program counter, where component A can fail in two modes:
1. Mode 1: in a way that does not affect the program counter and hence B directly, but is not able to provide its protective service, making B susceptible to probabilistic failure function given by \(1 - R_B\).

2. Mode 2: in a mode that affects the shared program causing failure to both B and the system.

The system reliability can be modeled using the following flow graph where I is the starting state, O-final state without failure, \(\text{ue}\) is the final state with failure or the occurrence of undesired event.

![Figure 7-4: State Transition Diagram for an Interfering Protective Component](image)

Since the two links connecting the input I to the output O are mutually exclusive, we can write, the reliability of the system as the sum of the probability of success of each of the two links.

\[
R_S = R_A + F_{A1} R_B = R_A + (1 - R_A - F_{A2}) R_B = R_A + R_B - R_A R_B - F_{A2} R_B
\]

**Equation 7-32**

where

- \(R_B\) is the probability of successful execution of B, represented by transition B
- \(R_A\) is the probability of successful execution of A, represented by transition A
- \(F_{A1}\) is the probability of failure of A in mode 1, represented by transition \(\overline{A}_1\)
- \(F_{A2}\) is the probability of failure of A in mode 2, represented by transition \(\overline{A}_2\)
- \(F_{A2} + F_{A1} + R_A = 1\)

By comparing Equation 7-31 with Equation 7-32, one can see that effect of not being able to fully tolerate the failure of component ‘A’ worsens system reliability by a value of \(F_{A2} R_B\). It is even possible for the ‘protected’ system to perform worse than the ‘unprotected’ system that consists of only component B. Thus, the condition for improving system reliability through protection without fault-tolerance may be given as follows.

146
\[
R_A + R_B - R_A R_B - F_{A2} R_B > R_B \\
\frac{R_A}{R_A + F_{A2}} = \frac{R_A}{1 - F_{A1}} > R_B
\]

**Equation 7-33**

If all the failures of the protective component are critical, i.e. if \( F_{A1} = 0 \), then the requirement for improving reliability through protection is to have \( R_A > R_B \).

The reliability of a system of two components A and B in which A does not play a protective role to B, may be given by \( R_S = R_A R_B \), if failure of A is not tolerated, or by \( R_S = R_B \), if A’s failure is tolerated and it failure does not interface with B.

Thus, fault-tolerance of components is one way of improving system reliability. A convenient way of implementing fault-tolerance is through partitioning of the event sets into multiple parallel processes, where the failure of some of the processes is tolerated so long as they are detected and their incorrect outputs is not used by the surviving processes. The failed processes may be terminated or re-started.

The need for forced termination or suspension of certain non-critical processes (processes whose failure does not result in the loss of a critical system function) may arise even in cases where the processes to be terminated are not in the failed state on their own but they are sharing finite resources with critical processes (processes whose failure results in the loss of a critical system function) and the shared resource is not sufficient enough to support both classes of processes. This scenario may also be considered as a special case of fault-tolerance.

**7.3.2.3 Redundancy**

The process structure is also the main mechanism of providing software redundancy, active or standby. In some applications where high availability/reliability of service is required and there is no resource constraint to cause interaction failures, it is possible to configure a number of processes to run concurrently preferably in a distributed environment, each of which taking identical or equivalent tasks and providing their results. If the different processes are obtained by running identical modules and working on identical inputs, the redundancy can effectively be only against secondary failure
causes. However, if the processes run diverse modules, or there is diversity in the inputs, the redundancy can be effective against primary failures too.

If the different processes are running concurrently or simultaneously, the type of redundancy is active redundancy. In some cases, a process may be initiated to take over the work of another process that encountered some undesired event, and many processes may be cascaded one after the other. Such type of redundancy can be considered as standby redundancy.

### 7.3.3 Connection and Event Structures for Fault Tolerance, Component Restorability and Redundancy

For a composite structure that provides a specific service, one way of increasing service reliability is through fault tolerant design, a set of design decisions that allow a system to continue providing its service even in the presence of failure of some of its components.

Obviously, no useful system can withstand the failure of all of its components at once. But design decisions can be made to tolerate the failure of some of the components or at least some of their failure modes. These decisions may also include provisions for detection and restoration of failed components while the system is providing its service.

As discussed in the previous sections, the ability to tolerate failure or to restore failed components largely depends on the properties of connectors used to connect the components and the event/process structure employed.

Specifically, the partitioning of events into concurrent processes is seen to improve the chance of fault-tolerance and provide redundancy that can include component restoration, provided that the connectors have the required property to do so. To achieve component restoration, replacement, and network reconfigurations in case of occurrence of undesired events, the connectors can be attachable and detachable to the components, even while the system is operational. This enables the composite structure to be extended by connecting new component instances when needed or contracted by detaching existing components if not needed. The state of each component is dependent only on the state of the input-output connectors attached to it and the objects contained in the connectors, but
not necessarily on the other components that may be attached or detached to the composite structure.

There are a number of advantages to this approach that are related to reliability, such as:

1. Software components that work towards a particular task can work in parallel behaving similar to hardware devices with improved failure independencies.
2. Any required service can be provided by connecting only a selected set of components that are relevant to the specific service and excluding others that are not required.
3. The failure of some components can be tolerated, if their outputs is not among the critical outputs of the service. In certain cases, failed components may also be restored to their initial state or replaced by alternative ones by attaching replacements to the same input/output connectors.
4. Different components may be configured to play different roles – input provision, component health monitoring and controlling, and output consumption – by simply connecting to the relevant connectors without having to modify the computing component. This will help to restrict the dependency among components in a particular service.
5. Redundant components, if exist, can be configured to work from the same input source and provide outputs in separate output connectors which can then be compared using output comparators.
6. Different services may be provided by changing the input/output transformation components and/or the connectors by a composite structure.

To discuss this author’s approach for fault-tolerance and restoration in detail, we first define what a connection enabled component is.

A connection enabled component is a component which requires the attachment of input/output connectors through which it provides its services. A composite structure is created by instantiating a set of connection enabled components and connectors and connecting the components through connectors. Each component can then read its inputs from the input connector and provide its output to the output connector working in its own speed. Process synchronization issues are solved at the connectors and creating and
managing the structure of the component network can be the responsibility of resource managers or some other coordinating components. Besides the input channels (IC) and output channels (OC), components communicate with their environment through monitoring and control event channels as well as the input-output objects that pass through the component. Figure 7-5 shows a connection enabled component in block diagram form.
Many connection-enabled components can be connected or cascaded together by sharing the connectors to create a composite structure. Figure 7-6 is an example of a composite structure consisting three connection-enabled components.

Figure 7-5: Block diagram of a connection enabled component

Legend:
- Component: input/output transformation component
- IC: input connector for receiving input objects
- OC: output connector – for providing processed objects
- M&C: Monitoring and Control – for receiving control signals and notifying relevant internal state changes
The above composite structure consists of three processing component instances, three monitoring and control channels and four input/output connectors. Since each component works in its own process space, output $O_1$ is dependent on components 1 and 2 while output $O_2$ is solely dependent on component 3. For a service in which outputs obtained from $O_1$ are considered as primary outputs but those obtained from $O_2$ as secondary outputs, failure of component C1 or C2 results in service failure but failure of C3 results in service degradation.

The connection also facilitates monitoring and detection of undesired events occurring in any of the components through the M&C channel. This can be used to restore or replace any of the components upon failure, with minimal disruption to the rest of the components in the system. For instance, if component 1 reports failure or occurrence of undesired event or if monitoring signals, example its response time, indicate problem, then the component may be detached from its input-output connector channels and a replacement component, either a diverse redundant component or a newly instantiated version of the same component, can be inserted in its place. Performing this could be a responsibility of another component. Meanwhile, both component 3 and component 2 can continue with their processes so long as their input channels have incoming data.

In case, where component 1’s absence may be tolerated, in that component 2 can directly work with inputs supplied at IC$_1$, then component 1 can be removed and component 2 can be connected to IC$_1$ after it completed processing the data in IC$_2$. This reconfiguration allows some of the failure of component 1, at least those that can be detected, to be tolerated.

Events and processes generally represent the active aspects of computer systems whose effect determine the success or failure of the provision of the required services. Together with connector structures, they determine the applicability of many of the design imperatives: modular redundancy, non-interfering behavior of connections and to some extent statistical dependencies. As much as their organization determines the fault tolerance and redundancy behavior of software systems, they are also causes for most, if not all, of interaction failures. Therefore, whether there will be improvement in reliability with a specific organization of connectors and processes or not, and to quantify the effect
of event structures towards the system reliability, one has to model their relation and behavior mathematically, which is the objective of the next section.

### 7.3.4 Event-Flow Equations

The event-flow equation is introduced to study the possible event sequences and their relative occurrence times in a composite structure. Since component instances are finite state machines, their outputs are a function of the sequence of events that occurred on each instance. In a stochastic environment where components may fail to behave as specified, resources became unavailable or deadlines times passed before a component completes its task, devices fail, and external interferences (for instance in communication signals) occur, the events that can occur on each component instance can include undesired events.

While the use of parallel processes improves the chance of fault-tolerance, it introduces other reliability concerns that may arise due to interactions. Probably the majority of undesired events which are also difficult to uncover through testing result from design errors in structuring of the events. Among the undesired events that are affected by event structures are:

- Race conditions where some input variables may change before a component completes computing an output value, or assumptions about the environment change before an output gets consumed,
- Missing of deadlines where one or more outputs become in undesired state due to delays in computing or communicating,
- Deadlocks where two or more processes may be forced to hold and wait for resources for each other in a way that no one can progress, and
- Undesired dependencies where the failure of some components that could have been tolerated, causes failure to other critical components as a result of structuring the events in a sequential order.

The author calls these types of undesired events interaction failure causes since their occurrence normally involve two or more components or processes and the design error may not be specifically associated to a particular component but in the way the composite structure is created.
Various design solutions may be provided to address these interaction failure causes but each of them may introduce its own concerns. For instance, putting all activities in a sequential process, if at all possible, may address the issue with race conditions but it will create unnecessary interdependencies among all components in the system that failure of any one of them will result in system failure. Besides, the failure rate of time sensitive components, with respect to missing deadlines, can increase due to the long wait for recurrent events in sequential processes. Allowing concurrent executions and synchronizations through hold and wait conditions can lead to deadlock or live lock conditions unless any of the necessary conditions for their occurrence is guaranteed not to occur by design. Providing each component with its own resources as it requests may result in depletion of resources, causing unavailability of resources for other incoming events. Thus, the study of interaction failure causes and possible solutions requires the assessment of the various concerns and the effectiveness of their solutions.

The TFM description for components provides the complete component output behavior including the conditions where undesired events occur. For a composite structure that involves a number of interconnected components, it is necessary to study whether each component receives the necessary event sequences to enable it to provide the required outputs and generate outgoing events that activate other components in the network, or whether undesired events could interrupt its functioning.

This can conveniently be done through event connection matrix, which is implicitly superimposed over the IO connection matrix discussed in 7.3.1.6. Since any output going out of a component into another component is a result of an event occurred in the system, the event connection matrix can be obtained from the IO connection matrix by deriving the events associated with each output communicated between components.

\[
Q_E = E(Q_{io}) = [q_{i,j}^E]
\]

\[
q_{i,j}^E = \{ e | v \in q_{i,j}^{io} \land v = O(T_{<C_j,>}) \land e = r(T_{<C_j,>}) \}
\]

**Equation 7-34**

Where T is the trace function and r(T) is the most recent event function as defined in the TFM method.
Just like inputs and outputs, we have two types of events. Events that correspond to transferring an output from a component to another (or updating an output variable) may be called outgoing events, and represented by $E_{n+1}$. Events that are associated to getting inputs to a component may be called incoming events and represented by $E_n$. This distinction is based on the direction of the event with respect to a particular component.

The event connection matrix, when viewed as a graph, shows the events as edges between the components. Thus, the same event $e$ can be an outgoing event for one component and an incoming event for another. This implies that, converted into sets $E_n$ and $E_{n+1}$ are equal.

The outgoing and incoming events for each component may be obtained from the event connection matrix by row and column projections similar to the output input variable.

$$E_{n+1} = Q_e \otimes 1_n^T$$
$$E_n = Q_e^T \otimes 1_n^T$$  \hspace{1cm} \text{Equation 7-35}

To understand the event interdependencies and estimate event arrival rates to the various components, we can consider each component as an event transformer from incoming events to outgoing events. While the outgoing and incoming events are related through logical operators, the departure rate of the events can be expressed as a linear function of the arrival rate of incoming events. Thus, for each component in a connection matrix, we may write a set of linear equations, written in matrix form as follows.

$$E_{n+1}^i = W_i \cdot E_n^i$$

\hspace{1cm} \text{Equation 7-36}

Where $W_i$ is a non-negative matrix, which may be called a component event transmission matrix, and may be obtained from each component’s interface specification/description document and $E_i$’s represent the set of events that arrive at and depart from the $i^{th}$ component in the system being studied.

For a composite structure that consists of the input-output connection of $n$-components, we can write the following system of equations.
\[ E_{n+1} = W.E_n \]

where

\[
\begin{bmatrix}
E_{n+1}^1 \\
E_{n+1}^n
\end{bmatrix} = \begin{bmatrix}
W_1 & \cdots & 0 \\
0 & \cdots & W_n
\end{bmatrix}
\begin{bmatrix}
E_n^1 \\
E_n^n
\end{bmatrix}
\]

Equation 7-37

This may be followed by the expansion of each of the E vectors into the basic events (the individual elements) and re-arrangement of the columns of W along with the rows of E, so as to make the two event vectors equivalent \( E_{n+1} = E_n \), for all events, except for the time, where events \( E_n \) correspond to the occurrence of events at time \( n \), \( E_{n+1} \) corresponds to events propagated through components and assumed to occur at time \( n+1 \).

The above equations provide the possible occurrence of events as a function of other events in a composite structure. Each event arriving at a component may or may not change the state of the component depending on the component’s behavior and its history. The possible system state-transition may be studied by combining the above equation with the input-output connection matrix.

The event-flow-equations can be used to assess whether certain states are reachable or not and to obtain estimates of dynamic properties such as component demand rate, component response rate, propagation delays and component utilization rate as discussed below.

7.3.4.1 Component Demand Rate:

Component demand rate may be obtained as the multiplicative reciprocal of the expected time between two successive event arrivals. Assuming the time of arrival of events to be a random variable. Let \( e^+ \) represent an incoming event to a component and \( e^- \) represent an outgoing event from the component, and when used with a subscript, \( k \), let \( k \) represents the time of occurrence. Then, a component demand rate may be defined by:

\[
\eta_c = E \left( \frac{1}{t(e^+_{c,n+1}) - t(e^+_c)} \right)
\]

Equation 7-38

where \( E \) in here represents the expected value of the random variable in the bracket.
In a composite structure, events may be assumed to flow from component to component according to the event flow structure discussed above. As a result, the demand rate for a particular component at any time can be obtained as a sum of the demands coming from all components in its environment, that include external triggers or independent event sources, and or other components in the neighborhood, i.e., dependent sources. The independent event sources may represent users of the system under study or its environment. To estimate the component demand rates in a composite structure, based on what was developed in Equation 7-37, and including the effect of independent sources, a system of equations may be written as follows:

\[ \sum_{i=1}^{m} \omega_i \eta_i + S_c = \eta_c \]

\[
\begin{bmatrix}
\eta_1 \\
\eta_2 \\
\vdots \\
\eta_m
\end{bmatrix} =
\begin{bmatrix}
\omega_{11} & \omega_{12} & \ldots & \omega_{1m} \\
\omega_{21} & \omega_{22} & \ldots & \omega_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
\omega_{m1} & \omega_{m2} & \ldots & \omega_{mm}
\end{bmatrix}
\begin{bmatrix}
\eta_1 \\
\eta_2 \\
\vdots \\
\eta_m
\end{bmatrix} +
\begin{bmatrix}
S_1 \\
S_2 \\
\vdots \\
S_m
\end{bmatrix}
\]

\[ \eta = \Omega \eta + S_c \]

Equation 7-39

The above equations provide a system of m linear equations with m unknowns and may be solved by using one of the well known linear equation solving approaches among which includes signal flow graph methods discussed in 5.8.3.

The solution to the above equations can be used to find the estimated demand rate at each component. By varying the event rates from external event sources between zero and some values, it is possible to study the effect of different loading combinations on the various components in the system. If the demand-rates at some of the components are found to be zero when some of the external sources are set to zero, it means that those components are not accessible from the components that have non-zero event rate sources.
7.3.4.2 Component Service Rate:
Estimate of component service rate may be obtained as the multiplicative reciprocal of
the expected time between a departure event and an arrival event:

$$\gamma_c = E\left(\frac{1}{t(e^n_{c,n}) - t(e^*_n)}\right)$$

Equation 7-40

7.3.4.3 Response Time and Component Utilization Rate:
Response time is the sum of component service time and waiting time. Components may
generally differ in their response time. Some components may require processing time
that is directly proportional to the size of the load to be processed. For instance, the
transmission time required by communication components for a given data is directly
proportional to the size of data and inversely proportional to the transmission bit rate. On
the other hand, a component that has to refresh a given display screen, may take an
almost equal time for each refresh operation irrespective of the data size displayed on the
screen. Other components, for instance, those that provide searching or sorting
operations, may have time requirements that may take logarithmic or other transcendental
function of the input size. Queuing theory provides different response time behavior for
components of different operational behavior.

In general, while we know the processing time of components to be some function of
component demand rate and execution/service rate, represented by $\tau_c = f(\gamma_c, \eta_c)$, this
function can be a complicated unknown function as it is dependent on the type of
components, the mode of service provision, the environment in which the service is
provided etc. Thus, we may take a simplified approximation and assume the processing
time to be directly proportional to the input/demand rate and inversely related to
component service rate:

$$\tau_c = \eta_c / \gamma_c.$$
The ratio of demand rate to service rate is also known as component utilization rate in performance related literature. Component utilization rate is a measure of the fraction of the time in which the component is busy.

Component utilization rate is given by:

$$\rho_c = \frac{\eta_c}{\gamma_c}$$  \hspace{1cm} \text{Equation 7-42}$$

### 7.3.5 Event-Flow-Graphs (EFG)

An alternative form of writing the event-flow-equations discussed in the previous section is using directed-graphs, which the author calls event-flow-graph. An event-flow-graph (EFG), is a directed graph representation of an event-flow-equation. EFG may be represented as double: $\text{EFG}=(V, E)$ where the vertex set $V$ represents events, and the edge set $E$, represent the conditional dependency and time ordering among a pair of events. Any incoming edge to a node represents a trigger for the event whereas each of the outgoing edges from a node represents the effect of the event in triggering other events. Labels on edges represent the conditions for trigger, which may include prior occurrence or non-occurrence of other events in the graph.

Alternatively, EFGs may represent the linear-version of the event-flow-equations, where the edges represent the weighting factors for the demand rates that flow through the links in the direction of connection.

#### 7.3.5.1 Example of Event Flow Equation/Graph

Figure 7-7 shows a simplified version of composite-structure among a few selected components in a mobile streaming system used as a case study in this research. The
actual system consists of much more components and interactions than the events shown in here.

Figure 7-7: Event structure in a simplified version of MSS

With event sets observed at the selected level of analysis given as \( E = \{ p, b, a, n, e, x, t, s, r, f, d, ds, j, c, r, u, po \} \), network inputs \( I = \{ j, c, r \} \), network outputs \( O = \{ po, pc, ds \} \) and probability of occurrence of some of the events given by \( P = \{ p_f, p_{ld}, p_c, p_f \} \), the event sequences in the above system may be written as a system of Boolean difference equations as follows.

\[
\begin{align*}
n_{n+1} &= j_n \\
u_{n+1} &= j_n \\
e_{n+1} &= n_n \\
x_{n+1} &= e_n \cdot u_n \lor r_n \quad \text{(event x can only be generated after events e and u, which can come at any order for new jobs and if there is a repeat order on previous jobs)} \\
f_{n+1} &= (x_n \lor p_{d0}) \neg c_n \quad \{ \neg c_n \rightarrow \text{if a job is not cancelled at time n, p_f \rightarrow probability of task f not completed} \} \\
d_{n+1} &= f_n \\
t_{n+1} &= d_n \\
b_{n+1} &= \neg c_n (d_n \lor p_{ld} a_n) \\
p_{n+1} &= b_n \\
p_{o(n+1)} &= q_{ld} p_n, \quad \text{where } q_{ld} = 1 - p_{ld} \\
a_{n+1} &= p_n
\end{align*}
\]
The above system of equations shows the possible event sequences that can provide the required outputs. The equations can alternatively be represented in event-flow-graph, which is shown in Figure 7-8.
Interpretation of Event Flow Graph and its Relation with TFM:

- An event occurs when one or more variables change. Components respond to events by computing new values and making changes to their output variables, giving rise to occurrence of new events. Although, the event flow graph is developed by using TFM descriptions, module-interface relations and input-output connection matrices, discussed earlier, the types of variables and the effect of the change on the variables are not shown in the event flow graph since they can be obtained from the detailed TFM descriptions.

- Time elapses at the nodes and increases with the direction of the arrow. If there are two events, e1 and e2, both occurring at component C, then the relation e2n+1 = e1n, or
the directed edge from e1 to e2 in the event flow graph, is equivalent to saying, the sub-trace \( \{e1,e2\} \in T_{\leq C} \) in TFM or if \( e2=r(T) \) then \( e1=r(p(T)) \)

- While traversing the graph following the direction of the arrow, the conditions may change from edge to edge due to the differences in time. For instance, two consecutive edges both containing condition c, may represent two different values, i.e. the value of c at two different time points.

- When there are loops or feedbacks in the flow graph, they represent the existence of recurrence, i.e. existence of similar events occurring repeatedly until certain task is completed or due to arrival of new tasks.

- Since time is always increasing, recurrent events at least differ in their time of occurrence, if not in other input variables. Hence, at anytime events from any part of the graph can occur concurrently (in parallel) or in arbitrary order so long as the conditions for their occurrence are satisfied.

- The occurrence of events can be deterministic, probabilistic and/or conditional to the previous occurrence or non-occurrence of other events. For the example given above:
  - Event j occurs whenever a user wants to use the system. It is the starting event for a new job to be processed. Its occurrence at any time is probabilistic which depends on the usage profile of the system.
  - Events n and u both follow event j deterministically (for the time being, we differ the analysis of undesired events that can occur at any time and make all transitions non-deterministic). Similarly event e occurs after n.
  - The occurrence of x requires the prior occurrence of e and u or u and e in any of the two orders. Such requirements are necessary to avoid undesired events where objects reaching at a given component may not be in the state that enables them to produce the required output at the next event in the event flow graph. (If we were to draw a Boolean network with logic gates to replace the event flow graph, each node would represent an ‘OR’ gate whose inputs are enabled by the incoming events. In cases where incoming events are labeled with occurrence of other events, the labels can be removed by replacing the ‘OR’ gate with ‘AND’ gate. Example, unlabeled
links ‘ux’ and ‘ex’ can be joined with an ‘AND’ gate whose output is then
joined to the ‘prx’ edge with an ‘OR’ gate.}

7.3.5.3 Transfer Functions in Event-Flow-Graph
Parallel to the well-known signal flow graphs (that are used for solving system of
equations and various network problems including probability and reliability problems
(Burroughs and Happ, 1962), (Rade, 1973), the event-flow-graph introduced in this
chapter can be used study the dependency among various events in the system and to
evaluate the transfer function between any two points of interest. Two types of transfer
functions can be obtained: logical transfer functions using Boolean algebra and linear
transfer functions using linear algebra.

The linear form of the event flow graph where the edges represent the weights and nodes
represent demand rates can be used to estimate the demand rates and other timing
properties, such as timing delays encountered when traversing the graph from one node to
another. This form gives rise to the linear transfer function and can be obtained using the
standard signal flow graph evaluation techniques that were covered in section 5.8.5.

The logical transfer function can be defined as the ‘transfer function’ between two
events/nodes as the logical sum of the paths (in the event flow graph) traversed between
the two events, where each path represents the sequence of events that must occur
between and including the two nodes. Events that appear on edges as conditions for
transition take the value represented by their respective nodes. The approach is similar to
the linear case, the main difference being the use of logical ‘OR’ operator to combine
inputs coming to a node in place of the addition operation in standard signal flow graphs
and the use of ‘AND’ operator in stead of product operator when traversing edges.
Additionally, determinant values that are used for scaling in linear equations has no
meaning in the Boolean equations and may be ignored by convention.

For instance, the transfer function from j to po in the above can be obtained as:

\[ t(j \to po) = j \cdot (u \cdot e_n + n \cdot e \cdot u_n) \cdot x \cdot \{ f, d \} \cdot b \cdot p \cdot \{ a, b, p \} \cdot po \]

\[ e_n = j \cdot n \cdot e \]
\[ u_n = j \cdot u \]
\[ t(j \rightarrow po) = j.(u.j.n.e + n.e.j.u).x.(f.d)*b.p.(a.b.p)*.po = j. u.n.e.x.(f.d)*b.p.(a.b.p)*.po \]

where \( \{ \}^* \) is used to represent the presence of possible recurrence.

### 7.3.5.4 Event Flow Graph and Design Approach

The event flow graph introduced above can be considered as a mathematical tool for describing the event/process structure of a given system and studying some of its dynamic properties. Unlike flow charts, control flow graphs or other similar program flow graphs, it does not dictate or imply ‘procedure or process oriented design’. Thus, it does not compromise changeability of components. For instance, Job module can work with any other module that provides equivalent interfaces to Exec module and Exec module can work with other modules having equivalent interface behavior as Job module. Such component replacements may be done in response to undesired events for restoration or switchover to redundant components if exist.

Many components may be designed to be multi-faced (may have different faces on different sides). Thus, replacement of one component by another may be done in order to obtain different system functions/services. In this case, each service may require its own event flow graph, since the relative importance of different outputs may vary from service to service.

### 7.3.5.5 Event Flow Graph and Reliability Analysis

In order to keep the above example simple, the possible undesired events that affect the system outputs as well as many of the components that provide protective role against some of these undesired events are not included.

By including all the relevant components, as well as the effects of the undesired events, the event flow graph can be used to analyze the reliability of the system for different modes of operation.

For instance, in the above system, the reliability of the system may be defined as the probability of obtaining all or part of the desired outputs for the given set of system inputs. The system consists of outputs represented by the set \( \{ po, pc, sd \} \). From the three outputs, \( po \), represents the class of completed jobs, which are the main outputs the service
is designed for, pc represents the class of cancelled jobs by the user and, sd represents all reporting or status display of the jobs by the system.

A multistage/multimode system reliability analysis can thus make distinction between the probabilities of getting all outputs correct, po and pc correct, po and sd correct and po correct. The highest reliability of the system at any time is given by the function that considers only the correctness of po, since this has the minimal set of components between system inputs and outputs.

The event flow graph can also be used to study the system timing behavior through analysis of possible event arrival rates and component response rates. These values can be used to estimate the probability distribution of time which affects the failure rate of some of the time sensitive components in the system. For such analysis, the same event flow graph can be used but edges will represent the rate of change of inputs that are communicated from one component to another, and the nodes represent the time spent in processing each event (or the execution rate). Such analysis is similar to electrical circuit analysis where rate of change of inputs (or event arrival rates) corresponds to current, program components to resistors and the time spent to voltage drops.

### 7.4 Chapter Summary

In this chapter the author has thoroughly discussed the various structures of software systems and their relation to the design imperatives identified in the previous chapter. To master the complexity of software systems, its various entities and concepts must be dealt as hierarchical aggregates. The software engineering literature usually does not provide clear definitions and relationships among the various elements of software product unless the literature is on a specific language or tool. As a result, confusion arises in identifying the distinction between say: programs, modules, components, and services. The hierarchical aggregate concept introduced in here will help to clarify the different concepts.

While the hierarchical aggregation is a rough categorization, various precise structure functions have been introduced that help to understand the connections among the different entities. The modular structure, type structure, connector structure and event/process structure have been defined and their relevance in system reliability
discussed. Connection and event structures for improved fault-tolerance, component restorability and redundancy have been discussed.

Mathematical documentation techniques have been developed for the various structures. The module-interface connection matrices can be used to derive service-module dependency matrix which is the main input for structure based reliability modeling. The input-output connection matrix along with the event-flow-graph is used to conduct dependency and reliability analysis among various components and events in the system. The various structure functions introduced in this chapter will be the basis for reliability modeling in the next chapter. Their application in a real case study will be the covered in chapter 9.
8. Document Based Structural Reliability Modeling

8.1 Introduction
The last DRP imperative identified in chapter 6 is modeling and predicting system reliability. Although various modeling approaches have been developed over the years, they lack the necessary connection with the design or the actual behavior of the system. As a result, they either make simplifying assumptions that do not depict the reality, or they become too complex to be trusted.

This chapter discusses a novel approach to reliability assessment of software systems that is based on precise description of component and system behavior. This author views reliability assessment from at least three different levels, component reliability, interaction reliability and service reliability. Component reliability is a measure of the probability of each component not meeting its interface specification, which is assumed to be known. Interaction reliability is dependent usually on many unknown factors such as availability resources, service demand rates, etc. Stochastic approaches are used to obtain failure density functions corresponding to them. The component reliabilities, interaction reliabilities and the connection information are used to develop deterministic models for reliability assessment of services and systems. Figure 8-1 shows the context in which a network of m components provide a set of n services sharing a finite set of resources.

Figure 8-1: Stochastic Operational Environment
8.2 Modeling Component Reliability

The reliability of a component is the ability of the component to perform its required functions for a given period of time when operating in a specified environment.

Components are subjected to two kinds of failure, primary and secondary failure. Component primary failure is a deviation of the component output from its specified requirements on the condition that it receives all the required inputs. Secondary failures are failures of a component to provide correct output due to undesired events caused by its environment, lack of required resources or incorrect inputs provided to it.

The reliability of a component can thus be given as a product of the probability of no undesired inputs from its environment and the probability of successful operation of the component given a proper functioning environment.

\[ R(t) = R_p(t)R_s(t) \]

\[ \lambda(t) = \lambda_p(t) + \lambda_s(t) \]

Equation 8-1

This is based on the competing risks model discussed in (Wallace R. Blischke and Murthy, 2000). According to this model, if a component is subject to \( K \) multiple causes of failure, then its failure rate may be given as:

\[ \lambda(x) = \sum_{i=1}^{K} \lambda_i(x), \text{ where } \lambda_i(x) \text{ is the failure rate associated cause } i \]

Equation 8-2

A software component can be replicated and used as part of many different systems. Hence, while the primary reliability function of the component remains the same across different replicas or instances, the secondary reliability differs from system to system and is a function of the specific component network structure where it is connected.

This requires having clear distinction between what constitutes part of the component and part of its environment, since both may contain sets of software components. The environment of a component includes software and hardware components that are connected to the component through its specified interfaces but their design and implementation is separate or independent from the component. Those components that are designed to be part of the component and provide its partial set of functions are
considered as part of the component. Failure of any of these subcomponents is considered to be part of the component’s primary failure, where as failure of those components that are attached to the components through its interfaces and are subject to change from network to network are considered as secondary failure.

Hence from now on, unless clearly mentioned, by component reliability, the author normally refer to the measure of its primary reliability, i.e.:

\[ R_C(t) = R_P(t) \]

**Equation 8-3**

Note that, if one has the detailed internal design of a module, its primary reliability can be given as a product of the reliability of the algorithm used to implement the module, the reliability of the specification and the reliability of the implementation code. Some problems which do not have correct algorithms that provide the required solutions can have a low primary reliability irrespective of the correctness of the implementation code.

### 8.2.1 Component Failure Rate

Software components normally operate upon demand. Even ‘continuously’ operating software components can be considered to operate upon demand where the demand rate is equal to the execution rate of the components. This demand rate is a function of their operational profile but the maximum demand rate is limited by the execution rate of the component. Since the primary failure of a software component is observed at demand time, component failure rate is a function of the demand rate and the probability of success per demand and may be found as follows. Assume a given component has probability of success per demand given by \( P \), and the rate of demand has exponential distribution with mean \( \eta \). The failure distribution of the component can be obtained from the flow-graph in Figure 8-2 using exponential transform analysis.

**Figure 8-2: Flow Graph of Component State Transition**

Legend:
- \( \bigcirc \) State
- \( \rightarrow \) Transition
  1: component in idle state
  2: component in operation
  ue: component in failed state or undesired event occurred
The probability density function (f(s) in frequency domain and f(t) in time domain) for encountering an undesired event is given by the transfer function from state 1 to ue where the undesired event has occurred:

Based on flow - graph transfer function, we can write

\[ f(s) = \frac{\eta}{s + \eta} (1 - P) = \frac{\eta(1 - P)}{s + \eta(1 - P)} \]

In time domain, we get

\[ f(t) = \eta(1 - P)e^{-\eta(1-P)t} \]

The corresponding distribution function in transform domain is given by

\[ F(s) = \frac{f(s)}{s} = \frac{1}{s + \eta(1 - P)} \]

Solving by factorization, we get

\[ F(s) = \frac{A}{s} + \frac{B}{s + \eta(1 - P)} \]

As + Bs = 0,

\[ A(\eta(1 - R)) = \eta(1 - P) \]

\[ F(s) = \frac{1}{s} - \frac{1}{s + \eta(1 - P)} \]

In time domain, the distribution function is given as

\[ F(t) = 1 - e^{-\eta(1-P)t} \]

Reliability function and failure rate can thus be obtained as

\[ R(t) = 1 - F(t) = e^{-\eta(1-P)t} \]

\[ \lambda(t) = \frac{f(t)}{R(t)} = \frac{\eta(1 - P)e^{-\eta(1-P)t}}{e^{-\eta(1-P)t}} = \eta(1 - P) \]

Hence, the component failure rate may be given by

\[ \lambda_c = \eta(1 - P) \]

Equation 8-4

And the component (primary) reliability function is given by:

\[ R_c(t) = e^{-\eta(1-P)t} \]

Equation 8-5
Equation 8-4 helps us to relate reliability requirements with feasibility of testing conditions as shown in the following example.

Example: Many life critical safety applications require a failure rate as low as \( \lambda = 10^{-9} \) per hour. Consider a machine that can execute a component with mean execution time of 1\( \mu \)s.

- What is the size of the test data that will give you the required failure rate estimation?
- How long will it take to conduct the test?

Probability of failure per demand \( p_f = 1 - P = \frac{\lambda}{\eta} = \frac{10^{-9} \times 10^{-6}}{3600} = 10^{-18} \).  

This means, we have to test for \( 3.6 \times 10^{18} \) test data and get only a maximum of 1 failure.

Number of test data cases required = \( 3.6 \times 10^{18} \)

Testing Time = Test Data Size * Execution time per data = \( 3.6 \times 10^{18} \times 10^{-6} \) sec = \( 3.6 \times 10^{12} \) sec \( = 3600 \times 10^9 / (3600 \times 24 \times 365.25) \) years = \( 1.1408 \times 10^5 \) years.

This is impossible to achieve by conventional testing.

8.2.2 Justification for the Use of Exponential Failure Rate for Software Components:

Equation 8-4 and Equation 8-5 in the previous section suggest exponential failure rate distribution for component failures. Software components normally fail when they are used or their failure is observed at the time of usage. Failure while not active may be assumed to be caused by failure of other hardware resources that store the software components. Such failure causes are considered to be secondary failure and hence their analysis is differed for structural reliability analysis.

Thus, the primary failure rate is directly related to the demand rate and the probability of failure per demand. In the absence of any other known usage distribution, it is quite common to assume exponential distribution of event arrivals, where events can occur at any time with some rate of occurrence. This gives rise to exponentially distributed failure rate.
Also, software components do not wear out. They are not age dependent as such. They may have some failure points created at design time, but there activation to cause system failure may come at any time during their operational life. This implies exponential distribution of failure. There are of course, some failure causes such as depletion of resources/memory fragmentation that may seem to be time dependent. However, this is time dependency in the short range. The system normally undergoes repeated renewal processes through system restarts, garbage collections, scheduled compaction, etc. In the long time range, the system may be considered as not affected by age. This provides another case supporting the exponential failure distribution assumption.

Additionally, the following justification discussed by Barlow (Barlow and Proschan, 1996) where exponential distribution is considered as the failure law of complex equipments suggests its applicability for software components.

Among the situations where the exponential distribution plays a prominent role, include a system consisting of many components, each subjected to an individual pattern of malfunction and replacement and all parts making up the failure pattern of the equipment as a whole. Under some reasonably general conditions, the distribution of the time between equipment failures tends to the exponential as the complexity and time of operation increases.

“Imagine a complex piece of equipment as a large number of sockets into each of which there is inserted a component. After the equipment is put into operation, it remains in operation continuously. We make the following assumptions.

1. Components in different sockets are not necessarily alike and are stochastically independent
2. Every component failure causes equipment failure
3. Each component is replaced immediately at failure
4. The process of failure detection, trouble location, and replacement is assumed to consume no appreciable time (or if it does, that time is taken to be part of the time between failures).

We are interested in the distribution of the time between failures after a long period of time has elapsed and as n (the number of components) becomes very large.”
It is shown in the same reference that a system that is characterized by the above assumptions has exponential failure distribution.

A well structured component based software system seems to satisfy the above assumptions. Consider the various program constructs as components in different sockets. A typical software component will contain many of them. Failure of each building block/construct leads to component failure. In software system, the failure of the component or system may causes damages to its users, but the programs can usually be re-run/restarted as before which may be considered as replacement by a new system. The process of re-running a component may not usually take much time, although the system may still fail again for the same input conditions. Alternatively, consider a component that can be used to instantiate as many instances as necessary (or create as many processes as needed), each of which with separate states. The different instances can provide computing service to different clients and while doing so may encounter failure. In restorative systems, the replacement of failed instances with newly created instances can be made in negligible time compared to the life time of the software system. Such failure and restoration behavior is similar to the complex equipments described above making exponential failure distribution the appropriate choice for modeling software component failure.

8.3 Modeling Interaction Failures (Stochastic Approach)
When a network of components sharing finite resources have to work together to provide a set of services, certain undesired system properties may emerge that may cause failure to a service or system. These undesired events may arise due to interactions among a number of components in a system and may be called interaction failure causes. They include: race conditions, missing of deadlines, depletion of resources and congestions, incompatible input/outputs, occurrence of events in incorrect orders, deadlocks and livelocks.

Some of these failure causes such as the possible occurrence of deadlocks may be avoided through design choices that avoid the simultaneous locking of more than one resource or by including protective components that address such issues. In the later case, the possibility of failing due to these failure causes is measured by the reliability of the
protective component. However, for many of the interaction failures, the design may provide some partial solution, but the failure may be caused by interaction with environments whose behavior may not be completely known at the time of design. For instance, race conditions can arise because of end user’s action or power interruption, while the system is processing some given task. Depletion of resources may occur either due to the partial failure of some of the resources in the system, unexpected surge in demands, or overloading of the computing system by other resource demanding components.

One characteristic of interaction failure causes is that it may usually be difficult to single out one component for the cause of failure. Rather, many components may contribute to their occurrence and those affected may not necessarily be members of the contributors. For instance, if a hard-real-time event occurs when a single processor is processing some un-interruptible task, then the processor may not be able to provide timely response for the new event.

These interaction failures are usually dependent on network structures, and many are related to resource management issues. This gives rise to cases where the same sets of components behave differently (in terms of their reliability) with different network structures.

The possible existence or absence of interaction failures can thus be obtained by making detailed behavioral network analysis along with the behavior of components in the network.

For instance, a service provided by a single sequential process may not have failures due to race-conditions, deadlocks or livelocks. However, possibility of using non-interfering restorative & redundant components is very much restricted, since at any time, the computing resources are assigned to a single component and failure of that component may lead to failure of the service. Failures due to deadlines, if participating components have, may be high with sequential processes.

With concurrent processes, the conditions for existence of race-conditions, deadlocks, livelocks and the effectiveness of protective mechanisms such as monitors may be analyzed.
A convenient way of analyzing interaction failure causes is by representing them as ‘virtual components’ whose reliability functions is derived by considering the interaction among components and resources. An area that may be further explored to handle these failure causes could be latent modeling.

In the sections below, various interaction failure causes are defined and reliability formulas are developed to quantify their possible occurrence in the form of reliability functions. These reliability functions can then represent the reliability of virtual components that are considered to exist in connection with the actual components to obtain the reliability of services and systems.

### 8.3.1 Missing Deadline

Failure due to missing of deadline occurs when the time elapsed between two events exceeds some given deadline time. It may result either from poor performance of components for some input conditions or interaction of various components and threads that result in scheduling problems such as lack of fairness or starvation. The undesired event due to missing of a deadline, whose time limit is given by $t_d$ may be expressed by the following condition.

$$\text{UE}_{d,c} = \{ t(e^-_c) - t(e^+_c) > t_d \}$$

**Equation 8-6**

The above equation can relate any two events that occur at one component or involve many components in the system. When it involves many components, the cause of the undesired event may be due to interactions, i.e., because of existence of many events in the system that cause delays in some of the responses. In some cases, the missing of deadline can occur even when a component is processing a given request without interruption by other processes. This usually happens due to poor performance of the component, but it may also result from interactions with other processes that share same computing environment. The elapsed time in this case is the time between an arrival event (or request event) and a departure event (or response event). The deadline condition can thus be stated as follows:

$$\text{UE}_{d,c} = \{ t(e^-_{c,n}) - t(e^+_{c,n}) > t_d \}$$

**Equation 8-7**
A general approach to this problem is to identify all time sensitive components in a given service or system, represent each possible undesired event by a virtual component and model its occurrence by studying the relevant interaction among components in the environment. The maximum waiting time/timeout properties of the components are assumed to be available in component documentation. The processing time or the production time of the expected outputs may be obtained by identifying those components that are responsible for producing the required output before the deadline. The relation between the two timing properties may be provided by the following general model.

Let \( t_d \) be timeout period (deadline) and let \( T_p = t(e_j) - t(e_i) \) be processing time. For the system to remain operational, the processing time should be less than or equal to timeout period. The reliability of the system (per demand) with respect to not missing deadline may be given by:

\[
R_d = P\{ T_p \leq t_d \}
\]

Equation 8-8

Either one or both of the timing properties \( (t_d \text{ } & \text{ } T_p) \) are assumed to be random variables. (If both are deterministic, then the above expression gives either 0 or 1 implying that either the system does not work at all or does not fail due to deadline).

### 8.3.1.1 Stochastic Processing Time and Deterministic Deadline

If \( t_d \) is deterministic and \( T_p \) is random variable with cumulative function distribution of \( F_p(t) \), then \( R_d \) is the value of the distribution function of the random variable \( T_p \) at \( t_d \), i.e.

\[
R_d = P\{ T_p \leq t_d \} = F_p(t_d)
\]

Equation 8-9

Estimating the deadline related reliability then boils down to finding the cumulative distribution function for the processing time and inserting the deadline time. Finding the cumulative distribution function for the time random variable depends on whether the missing of deadline property is a result of a delayed response by a single component or due to interactions involving many components.
8.3.1.1 Missing Deadline Due to Delayed Response by a Single Component

When missing of deadline is caused by a single component, i.e., \( T_p = t(e^{t^-} - t(e^{t^+}) \), what is required is the component’s time response distribution for representative input distribution to its operational environment.

The random variable \( T_p \) is dependent on input size, the availability of required resources for execution at the time of request and the component performance on the given input. Thus, \( T_p \) can be considered to be the sum of many independent random variables – the input selection, the resource condition at the time of selection, the processor speed, the presence or absence of other processes in the system, caching history and cash memory sizes, process scheduling strategy, priority level of processes, etc.

The distribution of many of these random variables may be difficult to find. In this scenario, the use of normal distribution for the processing time makes sense for the following reasons.

1. For any given component whose processing time can be measured based on a unit of input defined for the component, its response time would generally cluster around a certain mean value where the variations result from the state of the component, the specific input type, the state of resources used by the component etc. making it an additive random variable dependent on many contributory factors.

2. The fact that the response time is a sum of a number of random variables, where all of which are related to time, suggests the applicability of the well-known central limit theorem which states that a random variable that is obtained as the sum of different random variables tends to have normal distribution.

3. The normal distribution can generally be used as an approximation for many other distribution types (Feller, 1971).

Hence, in the absence of any other information about the time distribution of component performance, we may use the normal distribution of time behavior, hence, we have
\[ f(t; \mu, \sigma) = \frac{e^{-r_2 / 2\sigma^2}}{\sigma \sqrt{2\pi}} \]
\[ F(t; \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \int_{u=-\infty}^{t} e^{-r_2 / 2\sigma^2} \, du \]

**Equation 8-10**

If this assumption holds true and the mean response time \( \mu \) and standard deviation \( \sigma \) estimates are obtained through performance testing of the component with the expected usage profile, then the reliability of the component on any demand with respect to not missing a deadline can be estimated by the following.

\[ R_d(t_d) = \frac{1}{\sigma \sqrt{2\pi}} \int_{u=-\infty}^{t_d} e^{-r_2 / 2\sigma^2} \, du \]

**Equation 8-11**

The above reliability estimate provides the probability of providing a response before the deadline in any given demand. It can thus be represented by a virtual component having reliability per demand of the above value that is connected in series with the client and server components.

If a different distribution type that describes the response time better than the normal distribution is known, Equation 8-11 would be replaced by an appropriate expression. One distribution that may also have justifiable mathematical background for representing component response times for some of the components in the system is the exponential distribution.

It is known (Courtois, 1977, p79) that the superposition of independent renewal processes leads to a compound process that are similar to a Poisson process when the number of independent component processes increases. This property has been applied in near-decomposability theory where an aggregate random variable that models the superimposed behavior of several aggregates is shown to be exponentially distributed even though the individual aggregate output processes are non-Poisson. Some components may provide services whose timing behavior is determined by a superposition from different resources with arbitrary service time distributions. The response behavior of such components may then tend to be exponentially distributed.
cases where a component’s response time is known to be exponentially distributed with mean $1/\gamma$, then, the above equations will take the following form.

\[
\begin{align*}
    f(t; \gamma) &= \gamma e^{-\gamma t}, \text{ for } \gamma > 0, t > 0 \\
    F(t; \gamma) &= 1 - e^{-\gamma t} \\
    R_d(t_d) &= 1 - e^{-\lambda t_d}
\end{align*}
\]

Equation 8-12

### 8.3.1.1.1 Loading Factor

Equation 8-10, Equation 8-11 and Equation 8-12 are formulated based on a single demand with some relevant unit of load assumed for the component. In many cases, a component (or different instances of the component) may be used by different clients or in different modes of operation with large variations in loading conditions (e.g. through repeated calls or loops, streaming tasks, etc.). The response time of some of the components may be largely dependent on the size of loading. For instance, a communication component may have a given baud rate for transmitting messages. The response time (representing that total task completion time) of such component for clients that communicate a few bytes of data is quite different from those that may communicate many kilobytes or megabytes of data. Similarly, the response times of computational components may be dependent on the number of jobs being processed at a time.

The variation in loading conditions may result from partial failure of components or resources and may be used for dependency failure analysis. For instance, assume a service is being provided with replicated components that run on two parallel servers sharing incoming loads (say L units of load per unit time) among them. When the two servers are fully operational, the response time for each component would be what can be obtained from L/2. However, if one of the servers is down, the remaining server will have to carry all the L units of load and hence the response time would have to be given based on L units of load.

In order to obtain a more precise response time estimate and hence reliability prediction for applications of different loading requirements and different modes of operations, we may re-write the equations by taking into consideration the loading factor. This may be done through parameterized expression obtained as follows:
Let a component whose unit response time $T$ behavior is given in Equation 8-10 or Equation 8-12 has to process $L$ units of load. We are interested in finding the reliability of such a component not to miss a given deadline.

Let $T_L$ represent the random variable representing the total response time for $L$ units of load. This random variable can be obtained as the sum of the $L$ individual random variables $T$ having the same density functions. i.e.

$$T_L = \sum_{i=1}^{L} T$$

Equation 8-13

Evaluating the density function for $T_L$ from that of $T$ involves convolution of the functions which is a tedious process. An alternative way of finding the density functions is to use one of the transforms such as the moment generating function (MGF) or exponential transform so that the convolution changes to multiplication of the transforms, assuming independence between the $T$ random variables.

For the normal distribution given by Equation 8-10, the MGF transform of the density function is given by:

$$MGF(s) = e^{\mu s + \frac{\sigma^2 s^2}{2}}$$

Equation 8-14

which may be obtained from transform tables or may be derived as follows:
f(t) = \frac{1}{\sigma \sqrt{2\pi}} e^{\frac{(t-\mu)^2}{2\sigma^2}}

\text{MGF}(s) = \int_{-\infty}^{\infty} e^{st} f(t) dt = \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{\frac{(t-\mu)^2}{2\sigma^2}} dt = \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{\frac{(t-\mu)^2-2\sigma^2st}{2\sigma^2}} dt

= \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{\frac{-t^2-2\mu t-2\sigma^2st+2\mu \sigma^2 s+\sigma^4 s^2}{2\sigma^2}} dt = e^{\frac{-t^2}{2\sigma^2}} \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{\frac{(t-\mu+\sigma^2 s)^2}{2\sigma^2}} dt

= e^{\frac{\mu^2 + \sigma^4 s^2}{2}} \int_{-\infty}^{\infty} \frac{1}{\sigma \sqrt{2\pi}} e^{\frac{(t-\mu+\sigma^2 s)^2}{2\sigma^2}} dt

\text{since the integral evaluates to 1 as it is normal cumulative distribution function evaluated at } \infty.

\text{MGF}(s) = e^{\frac{\mu^2 + \sigma^4 s^2}{2}}

Assuming independence among the different random variables, T_L's, the MGF transform of the density function for T_L can thus be obtained as:

\text{MGF}(s_L) = \prod_{L} \text{MGF}(s) = (\text{MGF}(s))^L = (e^{\mu^2 + \sigma^4 s^2})^L = e^{L\mu^2 + L\sigma^4 s^2}

\text{Equation 8-15}

The expression given at the right hand side of the above equation represents MGF transform of a normally distributed random variable. Hence, it is clear that T_L is normally distributed with density function given as:

f(t_L) = \frac{1}{\sigma_L \sqrt{2\pi}} e^{\frac{(t_L-\mu_L)^2}{2\sigma_L^2}}

where
\mu_L = L \mu
\sigma_L^2 = L \sigma^2

Alternatively, we may write the response time density function for load of L units as

f(t_L, L) = \frac{1}{\sigma \sqrt{2\pi L}} e^{\frac{(t_L-\mu)^2}{2L\sigma^2}}

\text{Equation 8-16}
Hence the reliability of the component not to miss a given deadline when working with loading factor \( L \) is given by

\[
R_d(t_d, L) = \frac{1}{\sigma \sqrt{2\pi L}} \int_{t_d=\infty}^{t_d} e^{-\frac{(u-L\mu)^2}{2L\sigma^2}} du
\]

Equation 8-17

For normally distributed random variables, the above result could have been directly obtained from \( f(t) \) since it is well known that a random variable that is obtained as sum of normally distributed random variables is also normally distributed whose mean and variance are obtained as sum of the means and variances of each of the individual random variables, respectively.

However, the result Equation 8-17 can be applicable even if the random variables \( T \)'s are not normally distributed but have some other distribution with finite mean and variance, as could be inferred from the application of the well known central limit theorem in probability theory which is stated below:

8.3.1.1.1.2 Central Limit Theorem (CLT):

The central limit theorem (Feller, 1971) states that the mean of any set of variates with distribution having a finite mean and variance tends to the normal distribution. Let \( X_1, X_2, \ldots, X_L \) be a random sample size \( L \) from a population with a mean \( \mu \) and a variance \( \sigma^2 \) which are finite. Let \( \bar{X} \) be the sample mean. Then the distribution of

\[
\frac{\bar{X} - \mu}{\sigma / \sqrt{L}}
\]

tends to the standard normal distribution as \( L \rightarrow \infty \). That is,

\[
\frac{\bar{X} - \mu}{\sigma / \sqrt{L}} \xrightarrow{d} N(0,1).
\]

8.3.1.1.1.3 Estimating Deadline Reliability using CLT.
Consider the problem of finding the cumulative distribution for a random variable S obtained as the sum of random variables \( X_1, X_2, \ldots, X_L \). Let \( S = \sum X_i \). We would like to know, the probability \( F(s) = P[S \leq s] \).

From the central limit theorem, we know that,

\[
P\left( \frac{\bar{X} - \mu}{\sigma / \sqrt{L}} \leq x \right) \approx \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} \, dt
\]

\[
P\left( \frac{S}{\sigma / \sqrt{L}} \leq x \right) = P\left( \frac{S / L - \mu}{\sigma / \sqrt{L}} \leq x \right) = P\left( \frac{S}{L} \leq x\sigma / \sqrt{L} + \mu \right)
\]

\[
P(S \leq x\sigma / \sqrt{L} + L\mu) = P(S \leq x\sigma / \sqrt{L} + L\mu)
\]

\[
P(S \leq x\sigma / \sqrt{L} + L\mu) \approx \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} \, dt
\]

Let \( s = x\sigma / \sqrt{L} + L\mu \)

\[
P(S \leq s) \approx \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{s} e^{-\frac{t^2}{2}} \, dt = \frac{1}{\sigma \sqrt{2\pi}} \int_{-\infty}^{\frac{s-L\mu}{\sigma \sqrt{L}}} e^{-\frac{t^2}{2}} \, dt
\]

\[
P(S \leq s) \approx \frac{1}{\sigma \sqrt{2\pi L}} \int_{-\infty}^{s} e^{-\frac{(u-L\mu)^2}{2\sigma^2 L}} \, du, \text{ by substituting } t = \frac{u - L\mu}{\sigma \sqrt{L}} \text{ and } dt = \frac{du}{\sigma \sqrt{L}}
\]

From the above, we can see that the distribution of S, which is the same as \( T_L \) in our original notation is a normal distribution with

\[
\mu_S = L\mu \text{ and } \sigma_S^2 = L\sigma^2
\]

**Equation 8-18**

Hence, the reliability of not missing a deadline while working with load \( L \) units is given by the following equation which is the same as what was obtained in Equation 8-17:

\[
R_d(t_d, L) = F(t_d) \approx \frac{1}{\sigma \sqrt{2\pi L}} \int_{-\infty}^{t_d} e^{-\frac{(u-L\mu)^2}{2\sigma^2 L}} \, du
\]

In some cases, the above equation may be replaced by an appropriate one if the response time of the component for a unit load is known to have a specific distribution type and a closed form of the cumulative distribution function for the response time of the component with load \( L \) is known.
For instance, in case of exponentially distributed unit load responses, we can obtain the cumulative distribution function as gamma distribution as follows:

\[ f(t; \gamma) = \lambda e^{-\lambda t}, \text{ for } \lambda > 0, t > 0 \]

\[ MGF(s) = \int_0^\infty e^{st} \lambda e^{-\lambda t} dt = \int_0^\infty \lambda e^{-(\lambda-s)t} dt = \frac{\lambda}{\lambda-s} \]

Equation 8-19

\[ MGF(s_L) = \prod_L MGF(s) = (MGF(s))^L = \frac{\lambda^L}{(\lambda-s)^L} = \frac{1}{(1 - s/\lambda)^L} \]

Equation 8-20

Equation 8-20 is an MGF function for the gamma distribution function with scale parameter \(1/\lambda\) and shape parameter \(L\). Thus, the density and distribution functions are given by their standard definition given below:

\[ f(t_L) = \frac{\lambda^L t_L^{L-1}}{\Gamma(L)} e^{-\lambda t_L}, \text{ for } \lambda > 0, t_L > 0 \]

\[ F(t_L) = \int_0^{t_L} \frac{\lambda^L x^{L-1}}{\Gamma(L)} e^{-\lambda x} dx \]

with

mean, \( \mu_L = \frac{L}{\lambda} \) and variance, \( \sigma_L^2 = \frac{L}{\lambda^2} \)

Equation 8-21

Hence, the reliability with respect to not missing deadline is given by:

\[ R_d(t_d, L) = F(t_d) = \int_0^{t_d} \frac{\lambda^L x^{L-1}}{\Gamma(L)} e^{-\lambda x} dx \]

Equation 8-22

Note that relationships between the mean and variance of the cumulated response time for \(L\) units with the per unit response time obtained in Equation 8-21 is the same as what was obtained in Equation 8-16 and Equation 8-18, i.e.,

\[ \mu_L = L\mu \]

\[ \sigma_L^2 = L\sigma^2 \]
Since the mean and variance of an exponential distribution is given by
\[ \mu = \frac{1}{\lambda} \] and variance, \( \sigma^2 = \frac{1}{\lambda^2} \):

This also shows that Equation 8-17 is an approximation of Equation 8-22 since the approximation of a gamma distribution function by normal distribution function is done by using the same mean and variance in both functions. Alternatively, if \( X \) is a gamma distributed variable with scale parameter \( 1/\lambda \) and shape parameter \( L \), then a random variable \( Y \) given as follows tends to standard normal distribution:

\[ Y = \frac{X - L/\lambda}{\sqrt{L/\lambda}} = \frac{X - L\mu}{\sigma\sqrt{L}} = \frac{X - \mu_L}{\sigma_L} \sim N(0,1) \]

Hence, the approximation of the deadline reliability in normal distribution may be given as:

\[ R_d(t_d, L) = \frac{1}{\sigma\sqrt{2\pi L}} \int_{-\infty}^{t_d} e^{-\frac{(u-\mu_L)^2}{2\sigma^2L}} du = \frac{1}{\sigma_L\sqrt{2\pi}} \int_{-\infty}^{t_d} e^{-\frac{(u-\mu_L)^2}{2\sigma^2}} du = \frac{\lambda}{\sqrt{2\pi}} \int_{-\infty}^{t_d} e^{-\frac{(u-L/\lambda)^2}{2L}} du \]

Equation 8-23

Observe that in the above equations; if the mean processing time of components is smaller, then the reliability with respect to not missing deadline becomes higher, implying faster components have less chance of missing deadlines. Reliability with respect to not missing deadlines can be improved by making the components faster or increasing the deadline times.

8.3.1.1.2 Missing Deadline Due to Interaction between Multiple Components

When deadlines are missed mainly due to the performance of many components involved in the system, the interaction among the various components that affect the time need to be considered. In this case, the deadline condition is given by: \( UE_{d,c} = \{ t(e_j) - t(e_i) > t_d \} \)

With \( T_p = t(e_j) - t(e_i) \), the reliability of the system with respect to not missing a deadline is given by:

\[ R_d = P\{ T_p \leq t_d \} = F_p(t_d) \]

Equation 8-24
The presence of a number of components and/or processes that contribute to missing deadlines suggests that there are some sets of events that must occur between the two events of interest $e_j$ and $e_i$. The two events or those that must come between them may not necessarily be in one ‘sequential process’ as is known in operating system literature but they could also be from different concurrent processes or interactions with the environment.

Thus, a convenient way of finding the distribution time for such undesired events is to identify all the processes that contribute to the time distribution, define the relations among them with the help of flow graphs, and then obtain the density function as transfer function between the events being considered.

### 8.3.1.1.2.1 Analysis of Timing Behavior Using Event Flow Graph

Consider a simple (the approach is extendable to more complex networks through the well known flow-graph reduction techniques) producer-consumer problem where the producer is sensitive to encountering a timeout unless the consumer consumes what is produced within a given timeline. Assume also the case where the producer’s production capacity exceeds that of the consumer’s consumption capacity in that the consumer has to typically repeat its consumption a number of times for each production cycle.

Let us represent the various events involved in the producer-consumer process, as follows.

- $in =$ start of the process
- $p =$ production by producer
- $c =$ consumption by consumer
- $\tau_p =$ mean-production time
- $\tau_c =$ mean-consumption time
- $p_r =$ probability of repeating the consumption process. For a consumer of buffer size $b_c$ and producer of buffer size $b_p$,

\[
p_r = \begin{cases} b_c/b_p, & \text{For } b_c < b_p \\ 1, & \text{otherwise.} \end{cases}
\]

- $p_d =$ probability of the producer’s deadline not to have passed, $p_d = P\{T_c \leq t_{dp}\}$
ue = occurrence of undesired event in this case missing of deadline.
out = successful completion of the process
The event flow equations can be written as follows.

\[ c_{n+1} = p_n \lor p_d (1-p_r) c_n \]
\[ ue_{n+1} = (1-p_d)(1-p_r)c_n \]
\[ out_{n+1} = (1-p_r)c_n \]

When represented in flow graph form, they look like as follows.

![Flow Graph](image)

**Figure 8-3: Event Flow Graph for a producer-consumer process**

We are considering the primary failure causes separately. As a result, the edges representing the producer and consumer processes have a value of 1 for the transmission probability and the flow graph is used to obtain an estimate of reliability with respect to not missing the producer’s deadline. This can be obtained as the transfer function in terms of probability of the graph from in-to-out, giving rise to:

\[ R_d = \frac{p_r}{1-(1-p_r)p_d} \]

**Equation 8-25**

To evaluate the above, we need to know \( p_d \), the probability of not missing the deadline. Deadline is missed if the total time spent during the consumption cycle, that is before the process completes (moves to ‘out’ state) exceeds the producer’s deadline. Thus, if we have the distribution of the time for the duration when the process first enters to state 1 until it goes to state ‘out’, then it can be used to estimate the desired probability.

The distribution may be obtained from the same flow graph, but this time, we only consider the system that is not affected by the interruption due to missing of the deadline. Hence, we redraw the modified flow graph without the effect of the deadline as follows.
The interest here is to evaluate the time distribution. Unlike probability values which have multiplicative property when traversing from one edge to the next edge in a flow graph, the time values have additive properties. For instance, the probability of successfully reaching state 2 from ‘in’ can be obtained as a simple product of the two edge probabilities (‘in’-1 & 1-2), where as the time spent in reaching state 2 from ‘in’ is obtained as the sum of the time spent in the two branches. However, flow graph equation formulations are based on multiplication of edge values when traversing across paths (and sum of incoming edges at each node). Hence, we need to apply one of the transformations, such as Laplace transforms or Moment Generating Functions, on time values that will convert the additive property to multiplicative property. Such approaches are well-known in probability theory (Jenab and Dhillon, 2005).

If the consumption process is expected to have exponentially distributed unit consumption time, the total time elapsed from states 1 to ‘out’ will also be exponentially distributed as can be seen from the following:

$$f^c(s) = \int_0^\infty e^{-st} f(t) dt, \text{ Laplace/exponential transform}$$

For exponentially distributed density function with mean $\lambda = \frac{1}{\tau}$

$$f(t) = \lambda e^{-\lambda t}$$

$$f^c(s) = \int_0^\infty e^{-st} \lambda e^{-\lambda t} dt = \frac{\lambda}{s + \lambda}$$

Equation 8-26

For the flow graph given in Figure 8-4, we can obtain the exponential transform of the elapsed time as a transfer function from 1 to out.

$$f_{1o}^c(s) = \frac{\lambda_c p_r}{s + \lambda_c} = \frac{\lambda_c p_r}{s + \lambda_c - (1-p_c)\lambda_c} = \frac{\lambda_c p_r}{s + p_r\lambda_c}$$
Equation 8-27
One can see that Equation 8-27 is in the same form as Equation 8-26 indicating the elapsed time to be exponentially distributed. Hence, the probability of not missing the deadline can be obtained as:

\[ p_d = P\{ T_c \leq t_{d,p} \} = F_{10}(t_{d,p}) = 1 - e^{-\lambda c_{d,p}} = 1 - e^{-\lambda_{d,p}/\tau_c} \]

Equation 8-28
Substituting Equation 8-28 in Equation 8-25, we get:

\[ R_d(t_{d,p}) = \frac{p_r}{1 - (1 - p_r)(1 - e^{-t_{d,p}/\tau_c})} \]

Equation 8-29
As can be seen from the above, as the timeline by the producer approaches zero, the reliability of the system with respect to deadline approaches \( p_r \), whereas as if the timeline is made to be very large, \( R_d \) approaches to 1. Also, we can see that increasing \( p_r \) increases \( R_d \) and if \( p_r = 1 \) then there is no risk of missing a deadline.

Equation 8-29 is obtained based on the assumption of exponentially distributed consumption process. Nevertheless, from the discussion in the previous section, the consumer component may actually involve a number of components and its processing time may be normally distributed. In this case, the distribution function may have a different form. Below we attempt to find out this distribution function through the use of the Moment Generating Function (MGF) for its simpler expression than the Laplace transform when considering normally distributed functions. It is known that the MGF of the sum of independent random variables is equal to the product of the MGF of the individual random variables (Jenab and Dhillon, 2005).

For a consumer process that has normal distribution of consumption unit time with mean \( \mu_c \) and standard deviation \( \sigma_c \), the time distribution from the moment the process enters state 1 until it reaches state ‘out’ may be obtained first by evaluating the MGF for the density function as follows.
\[ f_{lo}(s) = \frac{e^{s \tau_c \sigma_c^2}}{2} \frac{p_r}{1-(1-p_r)e^{s \sigma_c^2}} = \frac{e^y p_r}{1-(1-p_r)e^{y}} \]

where

\[ y = s \tau_c + \frac{s^2 \sigma_c^2}{2} \]

**Equation 8-30**

The above function is the MGF transform of the density function for the time spent between the two states of interest. While obtaining the distribution function directly from the above is not obvious, one can easily evaluate the mean and variance for the random variable using the well known property of MGF derivatives.

Since the \( k \)th moment of a random variable can be obtained from its MGF as the \( k \)th derivative of the MGF evaluated at \( s=0 \), as given,

\[ M^k = \frac{d^k f(s)}{ds^k} \bigg|_{s=0} \]

One can obtain the mean and variance of the time spent between state 1 and ‘out’ as:
\[
\tau^{(1)}_{10} = \left. \frac{df_{10}(s)}{ds} \right|_{s=0} = \frac{(\tau_c + s\sigma_c^2) e^{\frac{s^2\sigma_c^2}{2}} p_r \left[ 1 - (1 - p_r) e^{\frac{s^2\sigma_c^2}{2}} \right]}{1 - (1 - p_r) e^{\frac{s^2\sigma_c^2}{2}}} \left[ 1 - (1 - p_r) e^{\frac{s^2\sigma_c^2}{2}} \right]}
\]

\[
\tau^{(2)}_{10} = \left. \frac{d^2f_{10}(s)}{ds^2} \right|_{s=0} = \frac{\left( \tau_c + s\sigma_c^2 \right) e^{\frac{s^2\sigma_c^2}{2}} p_r}{1 - (1 - p_r) e^{\frac{s^2\sigma_c^2}{2}}} \left[ 1 - (1 - p_r) e^{\frac{s^2\sigma_c^2}{2}} \right]}
\]

\[
\sigma^{2}_{10} = \left. \frac{M^{2}_{10}}{1 - (1 - p_r) e^{\frac{s^2\sigma_c^2}{2}}} \right|_{s=0} = \frac{p_r (\tau_c^2 + \sigma_c^2) p_r + 2p_r (1 - p_r)}{(1 - (1 - p_r))} - \frac{\tau_c^2 p_r}{p_r} \left( \frac{\sigma_c^2 - \tau_c^2 p_r + 2\tau_c^2}{p_r} \right) = \frac{\sigma_c^2}{p_r} - \frac{\tau_c^2}{p_r} \left( \frac{\sigma_c^2}{p_r} + \frac{(1 - p_r)\tau_c^2}{p_r} \right)
\]

Equation 8.31

8.3.1.2.2 Approximation by Normal Distribution Function:

If one was to use a simple approximation of the distribution function by a normal distribution having the same mean and variance, one will obtain the following function.

\[
F_{10}(t) \approx \frac{1}{\sigma_{10}\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{(x-\tau_{10})^2}{2\sigma_{10}^2}} \, dx
\]

\[
\approx \frac{p_r}{\sqrt{2\pi(\sigma_c^2 p_r + (1 - p_r)\tau_c^2)}} \int_{-\infty}^{\infty} e^{-\frac{(x+p_{10}-\tau_c)^2}{2(\sigma_c^2 p_r + (1 - p_r)\tau_c^2)}} \, dx,
\]

Equation 8.32
Both Equation 8-30 and Equation 8-28 show that the effect of the feedback loop is to increase the mean by a factor of $1/p_r$. This is similar to the effect of the loading factor $L$ in the previous section where $L=1/p_r$.

A more accurate but more involved form of approximation can use saddle point approximation (Butler, 2000) as follows.

### 8.3.1.2.3 Saddle Point Approximation of Distribution Function:

$$
\begin{align*}
    f_{10}(s) &= \frac{e^{\tau_e + s^2 \sigma_e^2}}{1 - (1 - p_r)e^{\tau_e + s^2 \sigma_e^2}} p_r \\
    K(s) &= \ln(f_{10}(s)) = \ln(e^{\tau_e + s^2 \sigma_e^2} p_r) - \ln(1 - \bar{p}_e e^{\tau_e + s^2 \sigma_e^2}), \text{ for } \bar{p}_r = 1 - p_r \\
    &= \ln(p_r) + s\tau_e + \frac{s^2 \sigma_e^2}{2} - \ln(1 - \bar{p}_e e^{\tau_e + s^2 \sigma_e^2}) \\
    K'(s) &= \tau_e + s\sigma_e^2 + \frac{1}{1 - \bar{p}_e e^{\tau_e + s^2 \sigma_e^2}} (-\bar{p}_e e^{\tau_e + s^2 \sigma_e^2})(\tau_e + s\sigma_e^2) \\
    &= (\tau_e + s\sigma_e^2) \left( 1 + \frac{\bar{p}_e e^{\tau_e + s^2 \sigma_e^2}}{1 - \bar{p}_e e^{\tau_e + s^2 \sigma_e^2}} \right) \\
    K'(s_0) &= t
\end{align*}
$$

Solving for $s_0$ from the above equation is a bit complicated and the solution is not unique (but can be solved with the help of mathematical tools such as MathLab). Alternatively, we can simplify the problem by considering only the mean-holding time and applying the MGF for a system whose mean-holding time is known following the derivation given below:
To find the saddle point approximation for the density function

\[ f_{10}(s) = \frac{e^{sr_c} p_r}{1 - (1 - p_r)e^{sr_c}} = \frac{p_r}{e^{-sr_c} - (1 - p_r)} \]

\[ K(s) = \ln(f_{10}(s)) = \ln(p_r) - \ln(e^{-sr_c} - p_r) \]

\[ K'(s) = \frac{\tau_c e^{-sr_c}}{e^{-sr_c} - p_r} = \frac{\tau_c}{1 - p_r e^{sr_c}} \]

\[ K'(s_0) = t \]

\[ 1 - p_r e^{sr_c} = \frac{\tau_c}{t} \]

\[ s_0 = \frac{1}{\tau_c} \ln \left( \frac{t - \tau_c}{\tau_c} \right) \]

\[ K''(s) = \frac{d}{ds} \frac{\tau_c e^{-sr_c}}{1 - p_r e^{sr_c}} = \frac{\tau_c^2 p_r e^{sr_c}}{(1 - p_r e^{sr_c})^2} \]

\[ K''(s_0) = \left( \frac{\tau_c^2 p_r e^{sr_c}}{1 - p_r e^{sr_c}} \right)^\frac{1}{2} \left( \frac{t - \tau_c}{\tau_c} \right)^\frac{1}{2} = t(t - \tau_c) \]

To find the saddle point approximation for the density function

\[ K(s_0) = \ln(p_r) - \ln(e^{-sr_c} - p_r) \]

\[ = \ln(p_r) - \ln \left( e^{-\ln \left( \frac{t - \tau_c}{\tau_c} \right)} - p_r \right) = \ln(p_r) - \ln \left( \frac{\tau_c}{t - \tau_c} \right) = \ln(p_r) - \ln \left( \frac{\tau_c}{t - \tau_c} \right) \]

\[ = \ln \left( \frac{p_r (t - \tau_c)}{\tau_c} \right) \]

\[ f_{10}(t) = \frac{1}{\sqrt{2\pi K''(s_0)}} e^{(K(s_0)-s_0)t} = \frac{1}{\sqrt{2\pi (t - \tau_c)}} e^{\ln \left( \frac{\tau_c}{t - \tau_c} \right) \frac{t - \tau_c}{\tau_c}} = \frac{1}{\sqrt{2\pi (t - \tau_c)}} e^{\ln \left( \frac{\tau_c}{t - \tau_c} \right) \frac{t - \tau_c}{\tau_c}} \]

\[ = \frac{1}{\sqrt{2\pi (t - \tau_c)}} \frac{p_r (t - \tau_c)}{\tau_c} = \frac{1}{\sqrt{2\pi (t - \tau_c)}} \frac{p (t - \tau_c)}{\tau_c} \left( \frac{t}{\tau_c} \right) \]

The distribution function may be obtained as follows (Butler, 2000):
\[ P(T \leq t) = F_{1o}(t) = \Phi(\omega) + \phi(\omega) \left( \frac{1}{\omega} - \frac{1}{u} \right) \]

where \( \Phi \) and \( \phi \) are the standard normal distributions and density functions, and

\[
\omega = \text{sgn}(s_0) \sqrt{2(s_0 t - K(s_0))} = \text{sgn}\left( \ln \left( \frac{t - \tau_c}{\bar{p} t} \right) \right) \sqrt{2 \left( \frac{t}{\tau_c} \ln \left( \frac{t - \tau_c}{\bar{p} t} \right) - \ln \left( \frac{p_\tau(t - \tau_c)}{\bar{p} \tau_c} \right) \right)}.
\]

\[ u = s_0 \sqrt{K''(s_0)} = \frac{1}{\tau_c} \ln \left( \frac{t - \tau_c}{\bar{p} t} \right) \sqrt{t(t - \tau_c)} \]

\[ p_d = P(T \leq t) = F_{1o}(t) = \Phi(\omega) + \phi(\omega) \left( \frac{1}{\omega} - \frac{1}{u} \right) \]

Equation 8-33

Equation 8-33 can be substituted to Equation 8-24 to obtain an estimate of reliability with respect to missing a deadline. Although these equations seem to be complicated, the parameters used are obtainable from design or test documentation. Specifically, \( p_r \) is a design property as it is a function of buffer sizes for producers and consumers, \( \tau_c \) is the unit consumption time obtainable from test data and \( t_{d,p} \) is a timeline set by producer, which must be available on producer’s component documentation.

Summary of Approach for Estimating Reliability with respect to not Missing Deadline

- Write out the event flow equations that directly or indirectly affect the timing behavior of a system of interest
- Draw the corresponding flow graph. One can start directly from this step if convenient.
- Identify the starting and terminating states for which the time distribution is required.
- Using one of the transformation techniques, Laplace or MGF, obtain the expressions for the transforms of the probability density functions
- Obtain the probability distribution functions either in closed form whenever possible or in their approximate form.

8.3.1.2 Stochastic Processing Time and Stochastic Deadline Time

In some cases, both the processing time as well as the deadline time can be random variables. This can happen if the deadline time is determined by some stochastic process,
for instance in the previous example on the production process that is dependent on input data. It can also happen if a deterministic timeline is superimposed on a random variable time and the cumulative time affects deadline conditions since a random variable added with a deterministic value gives random variable. The assessment of such values can be obtained as follows.

Let $T_d$ represent the random variable related to deadline time, $T_c$ represent the random variable related to the processing time (consumption time in the previous example). We can define another random variable $T$ as: $T = T_c - T_d = T_c + (-T_d)$ whose density function may be obtained from (Wallace R. Blischke and Murthy, 2000)

$$f(t) = \int_{-\infty}^{\infty} f_c(t_d) f_d(t + t_d) dt_d$$

**Equation 8-34**

$$R_d = P\{T_c \leq T_d\} = P\{T_c - T_d \leq 0\} = P\{T \leq 0\} = F(0).$$

**Equation 8-35**

For normally distributed deadline and processing times, with respective mean and standard deviation parameters given by, $(\mu_d, \sigma_d)$ & $(\mu_c, \sigma_c)$, $T$ is also a normal distribution with

$$\mu = \mu_c - \mu_d \quad \text{and} \quad \sigma = \sqrt{\sigma_c^2 + \sigma_d^2}$$

Hence, the reliability with respect to not missing deadline can be given as:

$$R_d = \Phi_T(0),$$

where

**Equation 8-36**

If both random variables are exponentially distributed, with mean values $\gamma_c$ and $\gamma_d$, we have,

$$R_d = \frac{\gamma_c}{\gamma_c + \gamma_d}$$

**Equation 8-37**

The derivation of this is given below.

For exponentially distributed times,

$$R_d = P\{T_c \leq T_d\}$$
Using a conditional approach (Wallace R. Blischke and Murthy, 2000, p176), we have

\[ P(T_c \leq T_d \mid T_d = t_d) = \int_0^{t_d} f_c(t_c)dt_c = \int_0^{t_d} \gamma_c e^{-\gamma_c t_c}dt_c \]

Removing the condition we get,

\[ P(T_c \leq T_d) = \int_0^\infty f_d(t_d) \left( \int_0^{t_d} f_c(t_c)dt_c \right) dt_d = \int_0^\infty \gamma_d e^{-\gamma_d t_d} \left( \int_0^{t_d} \gamma_c e^{-\gamma_c t_c}dt_c \right) dt_d \]

\[ = \int_0^\infty \gamma_d e^{-\gamma_d t_d} \left( 1 - e^{-\gamma_c t_d} \right) dt_d = \int_0^\infty \gamma_d e^{-\gamma_d t_d} dt_d - \int_0^\infty \gamma_d e^{-(\gamma_d + \gamma_c) t_d} dt_d \]

\[ = -e^{-\gamma_d t_d} \bigg|_0^\infty + \frac{\gamma_d}{\gamma_d + \gamma_c} e^{-(\gamma_d + \gamma_c) t_d} \bigg|_0^\infty \]

\[ = 1 - \frac{\gamma_d}{\gamma_d + \gamma_c} = \frac{\gamma_c}{\gamma_d + \gamma_c} \]

**8.3.2 Race Conditions**

Failure due to race conditions normally occurs if an event enters into a component (that has no queuing facility) before a previously entered event leaves the component. This condition may be expressed as follows. For a component that has no queuing facility for its clients,

\[ UE_{rc,c} = \{ t(e_{c,n+1}^+) < t(e_{c,n}^-) \} = \{ e_{c,n+1}^+ < e_{c,n}^- \} , \]

**Equation 8-38**

Since a new incoming event to a component needs to occur after the component responds to a previous event.

For components that have queuing facility, the above condition may not be considered as undesired event so long as the queue is not full. However, the arrival of incoming events at a faster rate than their departure rate from the component can eventually lead to component buffer overflow and hence occurrence of undesired events.

It can be shown that a system with no race condition with respect to overlap between its demand rate and its service rate, can be considered to be a *stable* system (Bolch et al.,
1998), which is characterized by the demand rate that is less than the service rate for each component in the system.

No race condition between request and response \( \Rightarrow \ t(e_{c,n+1}^{+}) > t(e_{c,n}^{-}) \) \( \Rightarrow \ t(e_{c,n+1}^{+}) - t(e_{c,n}^{+}) > t(e_{c,n}^{-}) - t(e_{c,n}^{+}) \)

\[ \Rightarrow E(t(e_{c,n+1}^{+}) - t(e_{c,n}^{+})) > E(t(e_{c,n}^{-}) - t(e_{c,n}^{+})) \Rightarrow 1/\eta_{c} > 1/\gamma_{c} \Rightarrow \eta_{c} < \gamma_{c} \Rightarrow \text{stable system.} \]

Failure due to race conditions can be generalized to cover all interaction failures caused by the occurrence of some events in between two other dependent events that invalidate the assumptions shared by the dependent events. For instance, a communication component may read the limits of maximum chunk size to be streamed across a communication channel and prepare the data to be streamed within this limit. However, if an event that lowers the limit occurs during communication, that is, before streaming ends, then the streamer may be unable to perform its task correctly.

If event \( e_{2} \) is dependent on event \( e_{1} \) in that it uses the outputs of \( e_{1} \), and event \( e_{c} \) is any event that can possibly change some of the assumptions communicated between \( e_{1} \) and \( e_{2} \) that makes \( e_{2} \) unable to give a proper result, then an undesired event occurs when \( e_{c} \) comes in between the two events. This condition may be written as:

\[ UE_{\eta_{c},c} = \{ t(e_{1}^{+}) < t(e_{c}^{+}) < t(e_{2}^{+}) \mid t(e_{1}) < t(e_{2}) \} = \{ e_{1} < e_{c} < e_{2} \mid e_{1} < e_{2} \} \]

**Equation 8-39**

In the above, it is expected that the occurrence of \( e_{2} \) is after \( e_{1} \) by design. What is not desired but can happen is the occurrence of \( e_{c} \) before \( e_{2} \). To model this, let us consider the occurrence of \( e_{1} \) as the process start time, \( t=0 \).

Let \( T_{c} \) be the random variable representing the next occurrence time of event \( e_{c} \) on the condition that \( e_{1} \) occurs at \( t=0 \), i.e. \( T_{c} = t(e_{c}) - t(e_{1}) \).

Let \( T_{e2} \) be the random variable representing the next occurrence time of event \( e_{c} \) on the condition that \( e_{1} \) has occurred at \( t=0 \), i.e. \( T_{e2} = t(e_{2}) - t(e_{1}) \).

The system will be able to work without problem if \( e_{c2} \) arrives before \( e_{c} \), i.e. \( T_{e2} \leq T_{c} \). To model this condition, we can define a random variable \( T \) as \( T = T_{e2} - T_{c} = T_{c} + (-T_{d}) \)
\[ R_{rc} = P\{ T_{e2} - T_c \leq 0 \} = P\{ T \leq 0 \} = F(0). \]

**Equation 8-40**

Observe that this equation is the same as the one obtained in Equation 8-35. Hence, the results of the previous section, stochastic processing time and stochastic deadline, are directly applicable to race conditions provided that we consider the appropriate conditional probability density functions, the condition being the occurrence of event \( e_1 \) at \( t=0 \).

For instance, for normally distributed event arrival times, with respective mean and standard deviation parameters given by, \((\mu_{e2}, \sigma_{e2}) \) & \((\mu_c, \sigma_c)\), \( T \) is also a normal distribution with

\[ \mu = \mu_{e2} - \mu_c \quad \text{and} \quad \sigma = \sqrt{\sigma_c^2 + \sigma_{e2}^2} \]

Hence, the reliability with respect to race conditions can be given as:

\[ R_{rc} = \phi_T(0) \]

**Equation 8-41**

For exponentially distributed events, the reliability of the system with respect to race conditions is given by:

\[ R_{rc} = \frac{\gamma_{e2}}{\gamma_{e2} + \gamma_c} \]

**Equation 8-42**

For components that have no queuing facility for incoming events, then race conditions may occur if the following condition is satisfied.

\[ UE_{rc,c} = \{ t(e_{c,n+1}^+) < t(e_{c,n}^-) \} = \{ e_{c,n+1}^+ < e_{c,n}^- \} \]

Implicitly in the above is that a demand \( e_{c,n}^+ \) corresponding to \( e_1 \) has occurred at time \( t=0 \).

Hence, the reliability with respect to race conditions is given by:

\[ R_{rc,c} = P\{ t(e_{c,n}^-) \leq t(e_{c,n+1}^+) \} = P\{ t(e_{c,n}^-) - t(e_{c,n}^+ \leq t(e_{c,n+1}^+) - t(e_{c,n}^+) \} \]

**Equation 8-43**
Note that from Equation 7-38 and Equation 7-40, the mean value of the random variables used in Equation 8-43 are given as the multiplicative inverse of component service rate $\gamma_c$ and the component demand rate $\eta_c$:

$$E(t(e_{c,n}^-) - t(e_{c,n}^+)) = 1/\gamma_c$$
$$E(t(e_{c,n+1}^-) - t(e_{c,n}^+)) = 1/\eta_c$$

For exponentially distributed demand and service rates, the reliability of each component that has no protective facility in a system with respect to race conditions is given by:

$$R_{rc,c} = \frac{\gamma_c}{\gamma_c + \eta_c} = \frac{\tau_{d,c}}{\tau_{d,c} + \tau_{p,c}}$$

where

$$\tau_{d,c} = 1/\eta_c$$
$$\tau_{p,c} = 1/\gamma_c$$

**Equation 8-44**

From the above, one can see that the slower the component demand rate and/or the faster the component service rate, the higher the reliability with respect to not encountering race conditions.

There are cases where race conditions may not occur at all in any of the components, among which include:

- If each incoming demand to a component is provided with its own resources (or objects) that are independent from other incoming demands to the component
- If there are protective components such as monitors that provide queuing service to incoming demands at each component to avoid race conditions
- In strictly sequential process configurations where no new event is allowed to enter into the network system unless a currently active event completes its departure.

The choice of any of these design solutions to reduce risk of race conditions can have its own reliability related risks that include both primary and interaction failure causes. For instance, forcing all components to be executed sequentially can lead to unnecessary interdependency among components and/or increased deadline failures. Protection through queuing may increase deadline failure causes. Providing each client with its own independent resource may increase the chance to depletion of resources. Thus, design
decisions may need to be analyzed based on the overall reliability effect on the service to be provided.

### 8.3.3 Resource Depletion and Congestion Conditions

Another class of failure causes that are dependent on the structuring of events and input/output variables are those events that may be caused by depletion of resource or congestions. Any machine has finite set of resources. The processing speed of processors is limited to some maximum value. The memory or storage size has limits however large it may be. Communication buses or channels are limited by their maximum channel capacity.

Hence, the amount of computing resources every processing component can be provided at any time is limited by the underlying machine capacity and the amount of resources being used by other components in the system. As a result, a request for processing at some point in time can even fail to start due to an unavailability of one or more of computing resources.

Most resources are shared among a number of components in different ways. The commonly used ways of resource sharing strategies include:

- **Time partitioning** – e.g. processors, busses
- **Space partitioning** – e.g. memory and storage devices
- **Frequency partitioning** – e.g. communication channels

Such partitioning allows counting or measuring the size of resources in a given system and making relevant risk analysis. For a resource of total capacity $n_r$, let $R_r$ be a random variable that represents the available resource size at any time $t$, i.e.

$$R_r = n_r - \text{size_of (resource being consumed)}.$$

Then, an arrival of an event at a component that requires resource size of $m$, can fail to start, if the available resource size is less than the required size. This condition may be given as follows:

$$UE_{rd,c,m} = \{m_{r,c} > R_r\}$$

*Equation 8-45*
The significance of the event structure in the above condition may not be clearly seen as it is hidden behind the random number, \( R_r \). This number is a function of the component demand rates along with each component's resource requirements, and the component response rates which provides the resource release rates. We can see this through an example.

Consider a simple system of one component whose primary reliability is 1, i.e., given all the necessary inputs and resources it requires, it always produces the correct output. As is common in software systems, the single component can be used to create a number of instances each of which providing the same service to the different clients at the same time. The maximum number of clients that can get the service of this system at the same time is limited by the maximum size of the available resources in the computing system and the size of resources required by each instance. As a result, the observed reliability of the simple one component system by any client can be less than 1.

Note that this failure cause is different from failure of the resources themselves. Basically, all resources are components that are subject to failure and their failure usually results in system failure if they are relevant and there are no redundant resources to take-over.

In environments where there is scarcity of resources, i.e., the available resources in the operational environment may not always exceed the resource requirements of the components in the system, interaction failures due to depletion (even temporarily) of shared resource can arise with non-negligible failure probabilities.

Resource depletions and congestions can also occur in cases where there is significant difference in the speed of communicating components, especially when the demand rate for a given component exceeds that of the execution rate of the component. In such cases, each demand for a component will have to queue up increasing the size of the component-input queue with time. However, since any machine will have some finite limit beyond which the queue length cannot increase, there may come a point where other incoming demands may not be served, initiating undesired event.

Thus, the reliability analysis must include the effect of each component on each shared resource and vise versa in order to model failure caused by depletion of resources.
8.3.3.1 Modeling Resource Depletion using Birth-Death Process

For a system that requires $m_{c,r}$ number of resource units to operate, the undesired event conditions for the occurrence of depletion of resources or congestion in a system was given earlier as:

$$UE_{rd,c,m} = \{m_{c,r} > R_r\}$$

And the probability of getting sufficient resource to undertake its task, and hence the reliably per demand with respect to resource availability is given by:

$$R_{rd} = P\{m_{c,r} \leq R_r\} = P\{m_{c,r} \leq n_r - K_r\} = P\{K_r \leq n_r - m_{c,r}\}$$

Equation 8-46

Where,

- $n_r$ = total capacity of resource $r$,
- $K_r$ is a random variable that represents the size of resources being used by components in the system at any time $t$.
- $R_r$ is a random variable that represents the available resource size at any time $t$, i.e, $R_r = n_r - K_r$.
- $m_{c,r}$ = the minimum size of resource $r$ that is required by component $c$ for it to provide its service.

To evaluate this probability, we consider a stochastic system having $n+1$ states numbered from 0 to $n$ where the state numbering corresponds to the size of resources that are being consumed by components in the system and hence not available for use. By considering infinitesimally small time duration, we can assume the state of the system to change by either a removal of a unit of resource as a component consumes a resource or the addition of a unit of resource as a component releases a resource.

This process gives rise to the well known birth-death process in probability theory. The transition between states is governed by the resource demand and resource release rates. Let the resource demand rate at the $i$th state be $\lambda_i$, and let the resource release rate be given by $\mu_i$. Figure 8-5 shows the flow graph for state transition of the available resource.
Figure 8-5: Flow Graph of Birth Death Process

The probability of getting sufficient resources can then be obtained using Continuous Time Markov Chain (CTMC) analysis. The infinitesimal generator matrix for the above looks like:

\[
Q = \begin{bmatrix}
-\lambda_0 & \lambda_0 & 0 & 0 & \ldots & 0 & 0 & 0 \\
\mu_1 & - (\lambda_1 + \mu_1) & \lambda_1 & 0 & \ldots & 0 & 0 & 0 \\
0 & \mu_2 & - (\lambda_2 + \mu_2) & \lambda_2 & \ldots & 0 & 0 & 0 \\
& & & & & & & \\
0 & 0 & 0 & 0 & \ldots & \mu_{n-1} & - (\lambda_{n-1} + \mu_{n-1}) & \lambda_{n-1} \\
0 & 0 & 0 & 0 & \ldots & 0 & \mu_n & - \mu_n
\end{bmatrix}
\]

The steady state solution is obtained by solving the equation

\[
\pi Q = 0,
\]

where \( \pi = [\pi_0 \ \pi_1 \ \pi_2 \ \ldots \ \pi_n] \) is the state vector, which gives:

\[
\pi_k = \pi_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}, \text{ for } 0 < k \leq n
\]

Since sum of all stationary probabilities has to be 1, we have

\[
\pi_0 + \sum_{k=1}^{n} \pi_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} = 1
\]

\[
\pi_0 = \frac{1}{1 + \sum_{k=1}^{n} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}
\]

\[
\pi_k = \frac{\prod_{i=0}^{k-1} \lambda_i}{1 + \sum_{k=1}^{n} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}
\]

Equation 8-47
Using these state probabilities, one can find the probability of the availability of enough resource to undertake a given task. For a service that requires a minimum of \( m \) resources (out of \( n \)), the probability of getting its demand at any time (on steady state conditions) can be given as the sum of the probability of being in the first \( n-m \) states, i.e.:

\[
R_{rd,r} = P(K \leq n-m) = \sum_{k=0}^{n-m} \pi_k = \frac{\sum_{k=0}^{n-m} \prod_{i=0}^{k-1} \lambda_i}{\left(1 + \sum_{k=1}^{n} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}\right)}
\]

Equation 8-48

Alternatively, the asymptotic failure rate \((\tilde{\Lambda}(\infty))\) may be estimated as the rate of transition from the minimal operating state to failed state as follows (Villemeur, 1992b p296):

\[
\tilde{\Lambda}(\infty) = \frac{\sum_{i \in \mathcal{E}_M} \lambda_i \pi_i}{\sum_{i \in \mathcal{E}^F} \pi_i}
\]

\[
\lambda_i = \sum_{j \in \mathcal{E}^O} q_{ij}
\]

where\n
\( \mathcal{E}_M \) is the set of minimal operating states (states with at least one transition to the failed states)\n
\( \mathcal{E}^O \) is the set of operating states and\n
\( \mathcal{E}^F \) is the set of failed states.

\[
\tilde{\Lambda}(\infty) = \frac{\lambda_{n-m} \pi_{n-m}}{\sum_{i=0}^{n-m} \pi_i} = \frac{\left(1 + \sum_{k=1}^{n-1} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}\right) \lambda_{n-m} \prod_{i=0}^{n-m-1} \frac{\lambda_i}{\mu_{i+1}}}{\sum_{i=0}^{n-m} \pi_i \sum_{k=1}^{n-1} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}} \left(1 + \sum_{k=1}^{n} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}\right)
\]

Equation 8-49
The actual values of reliability per demand or failure rate with respect to resource depletion rates is dependent on the relation between the resource demand and release rates, which may in turn depend on the mode of resource sharing. Two typical cases are discussed below.

8.3.3.1.1 Resource Depletion for the Case of Consumption and Release by Independent Processes

In many cases, various services may be running on a system sharing a set of resources. These services may be unrelated to each other in terms of their states, even if they share program code, in that their processes may not be communicating directly to each other, other than the indirect influence that came due to sharing of the resources. In such cases, each process contributes to the resource consumption and depletion rate independently, resulting in some aggregate demand and release rates. Let these aggregate demand and release rates by represented by \( \lambda_r \) and \( \mu_r \) respectively. In the state transition flow graph Figure 8-5, the resource demand rate can be considered to be independent of the state of the system as it is driven by the execution rate of processes, while the resource release rate is state dependent. The less the number of available resources in the system implies that many processes have consumed more resources and they are likely to return it. Therefore, the state diagram can be modified by making \( \lambda_i = \lambda_r \) and \( \mu_i = i \mu_r \), for each \( i \).

Substituting these values in, gives rise to:

\[
R_{rd,r} = P(K \leq n - m) = \frac{\sum_{k=0}^{n-m} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}{1 + \sum_{k=0}^{n} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}} = \frac{\sum_{k=0}^{n-m} \prod_{i=0}^{k-1} \frac{\lambda_r}{(i + 1)\mu_r}}{1 + \sum_{k=0}^{n} \prod_{i=0}^{k-1} \frac{\lambda_r}{(i + 1)\mu_r}} = \frac{\sum_{k=0}^{n-m} \left( \frac{\lambda_r}{\mu_r} \right)^k}{1 + \sum_{k=0}^{n} \frac{1}{k!} \left( \frac{\lambda_r}{\mu_r} \right)^k}
\]

Equation 8-50
Equation 8-51

\[ \tilde{\Lambda}(\infty) = \frac{\lambda_{n-m} \prod_{i=0}^{n-m-1} \lambda_{i}}{\sum_{k=0}^{n-m} \prod_{i=0}^{k} \frac{\lambda_{i}}{\mu_{i+1}}} = \frac{\lambda_{r} \left( \frac{\lambda_{r}}{\mu_{r}} \right)^{n-m}}{(n-m)! \left( \frac{\lambda_{r}}{\mu_{r}} \right)^{k}} \]

In some cases, an approximate solution may suffice to see whether the risk of resource depletion is significant or not. In such cases, the Taylor series approximation may be used to simplify the above equations. Observe that the summations in the above equations is the first n-m+1 terms of the Taylor series representation of exponential function. From this, we can obtain:

\[ e^{\frac{\lambda_{r}}{\mu_{r}}} = \sum_{k=0}^{\infty} \frac{1}{k!} \left( \frac{\lambda_{r}}{\mu_{r}} \right)^{k} = \sum_{i=0}^{n-m} \frac{1}{k!} \left( \frac{\lambda_{r}}{\mu_{r}} \right)^{k} + \sum_{k=n-m+1}^{\infty} \frac{1}{k!} \left( \frac{\lambda_{r}}{\mu_{r}} \right)^{k} \]

where

\[ r_{n-m+1} \left( \frac{\lambda_{r}}{\mu_{r}} \right) = e^{c} \left( \frac{\lambda_{r}}{\mu_{r}} \right)^{n-m+1} \frac{1}{(n-m+1)!} \text{ for some } c, 0 \leq c \leq \frac{\lambda_{r}}{\mu_{r}}. \]

Substituting the above in Equation 8-50 and Equation 8-51, we get approximating equations for reliability per demand and failure rate as:

Equation 8-52
Equation 8-53

\[ R_{rd,r} = \frac{\sum_{k=0}^{n-m} \frac{1}{k!} \left( \frac{\lambda_r}{\mu_r} \right)^k}{1 + \sum_{k=0}^{n} \frac{1}{k!} \left( \frac{\lambda_r}{\mu_r} \right)^k} = e^{\frac{\lambda_r}{\mu_r}} - e^{c} \left( \frac{\lambda_r/\mu_r}{(n-m+1)!} \right) \]

for some \( c, 0 \leq c \leq \frac{\lambda_r}{\mu_r} \)

Equation 8-54

\[ \tilde{\Lambda}(\infty) = \frac{\lambda_r}{(n-m)!} \left( \frac{\lambda_r}{\mu_r} \right)^{n-m} \]

\[ \times \left( e^{\frac{\lambda_r}{\mu_r}} - e^{c} \left( \frac{\lambda_r/\mu_r}{(n-m+1)!} \right) \right) \]

8.3.3.1.2 Resource Depletion for the Case of Consumption and Release by Dependent Processes (The Case of Producer-Consumer Processes)

Many problems require designs that would involve the use of producer-consumer processes either in single pair or in group pairs. Such processes do not occur independently but rather jointly to undertake some required task. The solution would require having a common store or buffer where producers store their outputs and consumers consume from.

In cases where the producer and consumer speeds are not synchronized, different types of undesired events can occur. For instance, starvation may affect fast consumers if the producers are slow. On the other hand, when the producers are faster than the consumers, the producers may not get any more room to place their output due to the depletion of the storage resources.

Resource depletions caused by such type of process can also be analyzed using Equation 8-48 and Equation 8-49, but in this case, the resource consumption rate is directly equal to the production rate and the resource release rate is equal to the consumption rate. Both
rates are not resource state dependent, i.e. \( \lambda_i = \lambda_r \) and \( \mu_i = \mu_r \). Substituting these values into Equation 8-48 we get:

\[
R_{rd, r} = P(K \leq n - m) = \sum_{k=0}^{n-m} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \prod_{i=k}^{n-m-1} \frac{\lambda_i}{\mu_{i+1}} = \frac{\sum_{k=0}^{n-m} \left( \frac{\lambda_r}{\mu_r} \right)^k}{\sum_{k=0}^{n} \left( \frac{\lambda_r}{\mu_r} \right)^k} = \frac{\sum_{k=0}^{n-m} \left( \frac{\lambda_r}{\mu_r} \right)^k}{\sum_{k=0}^{n} \left( \frac{\lambda_r}{\mu_r} \right)^k}
\]

Equation 8-55

Simplifying the geometric series in the above gives:

\[
R_{rd, m} = \left[ \frac{1 - \left( \frac{\lambda_r}{\mu_r} \right)^{n+1}}{1 - \left( \frac{\lambda_r}{\mu_r} \right)^{n+1}} \right] = 1 - \left( \frac{\lambda_r}{\mu_r} \right)^{n+1}, \quad \text{for } \lambda_r \neq \mu_r
\]

\[
= \frac{n-m+1}{n+1} = 1 - \frac{m}{n+1}, \quad \text{for } \lambda_r = \mu_r
\]

Equation 8-56

Similarly, corresponding to Equation 8-49, the failure rate may be given as:

\[
\tilde{\Lambda}(\infty) = \frac{\lambda_{n-m} \prod_{i=0}^{n-m-1} \lambda_i}{\sum_{k=0}^{n-m} \prod_{i=0}^{k-1} \lambda_i / \mu_{i+1}} = \lambda_r \left( \frac{\lambda_r}{\mu_r} \right)^{n-m} \left( 1 - \left( \frac{\lambda_r}{\mu_r} \right)^{n+1} \right) / \left( 1 - \frac{\lambda_r}{\mu_r} \right)
\]

\[
\tilde{\Lambda}(\infty) = \lambda_r \left( \frac{\lambda_r}{\mu_r} \right)^{n-m} \left( 1 - \left( \frac{\lambda_r}{\mu_r} \right)^{n+1} \right) / \left( 1 - \frac{\lambda_r}{\mu_r} \right) = \lambda_r \left( \frac{\lambda_r}{\mu_r} \right)^{n-m+1} (\mu_r - \lambda_r) / (\mu_r^{n-m+1} - \lambda_r^{n-m+1})
\]

Equation 8-57

Example: If a producer-consumer process runs on a handheld device having 32MB (i.e. n=32) memory, where the producer and consumer process produce or consume 1MB (m=1) of data at a time, the following are estimates of the reliability with respect to depletion of memory resource for different relative producer and consumer speeds.
1. For a system where the rate of production is twice that of the rate of consumption, i.e. 
\( \lambda_r = 2\mu_r \), we get 
\[
R_{rd,m} = 1 - \left( \frac{\lambda_r}{\mu_r} \right)^{n+1} = \frac{1 - 2^{32}}{1 - 2^{33}} = 0.49999999942 \approx 0.5
\]

2. For a system where the rate of consumption is twice that of the rate of production, 
i.e., \( 2\lambda_r = \mu_r \) 
\[
R_{rd,m} = 1 - \frac{(1/2)^{32}}{(1/2)^{33}} = 0.99999999883 \approx 1
\]

3. For a system where the rate of production and the rate of consumption are equal, i.e. 
\[
R_{rd,m} = 1 - \frac{m}{n+1} = 1 - \frac{1}{32} = 0.96875
\]

### 8.3.4 Deadlocks and Livelocks

Deadlocks and livelocks are also possible undesired events that may or may not occur depending on the structuring of events and resource sharing mechanism among competing events. The conditions for deadlocks and livelocks may exist if there are two or more processes that may have to share two or more resources/objects in mutual exclusion with each other and the processes are allowed to hold for a resource and wait for other resources to be released. These conditions normally occur in designs involving concurrent processes and attempts to avoid race conditions. Since, many processes and many variables can be created from a single module or component; the conditions can occur even in cases that contain only one module.

Deadlock is an undesired event that occurs if different threads wait for each other in which none of the threads would be able to continue their work. Livelock on the other hand occurs when threads continue to execute (not in a blocked state) but they are not able to progress towards completion of their tasks.

Various approaches of avoiding the occurrence of these undesired events or ways of detecting and resolving such problems exist. For instance, design approaches that share nothing but only communicate through messaging and approaches that avoid locking of more than one shared resource avoid the possible occurrence of these undesired events. In
cases where sharing of resources is unavoidable, it is generally better to allocate the sharing responsibility to one or a few specific components where the non-occurrence of these undesired events is made to be the reliability of the specific components. When avoidance is not possible for some reason, deadlock detection and prevention mechanism could be put in place.

Despite the above, it may be possible for deadlocks to occur. Hence, this author models the possibility of their occurrence through a stochastic process, formulated as a resource allocation problem. Assume there are \( n \) processes that compete for \( m \) resources. Let the demand rate for these resources from the \( n \)-processes be given by \( \eta \). Let \( R_{dp} (=1-F_{dp}) \) be the reliability of deadlock protector component, i.e., its ability to prevent a deadlock from occurring or resolve it without problem if it occurs. A system that has no deadlock protector has \( R_{dp}=0 \) and a system which can prevent all deadlock occurrences has \( R_{dp}=1 \).

Let \( \gamma_c \) be the collection rate of resources by a process, i.e., a process may take a mean time of \( 1/\gamma_c \) to collect all the resources it needs to complete its task. If a deadlock detector fails to prevent or there is no protection mechanisms, other processes may start collecting resources before a process is granted access. The state of the system may be modeled using the following state transition diagram.

![State Transition Diagram modeling Possible Deadlock Occurrence.](image)

**Figure 8-6: State Transition Diagram modeling Possible Deadlock Occurrence.**

Using a semi-Markov modeling (Villemeur, 1992b, pp 323-326), the occurrence rate of undesired condition may be estimated as follows.
\[
B = \begin{bmatrix}
0 & \gamma_c & 0 \\
\frac{\gamma_c}{\gamma_c + F_{dp}\eta} & 0 & \frac{F_{dp}\eta}{\gamma_c + F_{dp}\eta} \\
0 & 0 & 0 
\end{bmatrix}
\]

\[
B_2' = B - \begin{bmatrix}
1 \\
0 \\
0 
\end{bmatrix}
\]

\[
[1 - B_2']^{-1} = \begin{bmatrix}
\frac{\gamma_c + F_{dp}\eta}{F_{dp}\eta} & 1 \\
\frac{\gamma_c}{\gamma_c + F_{dp}\eta} & 1 
\end{bmatrix}
\]

\[
D = \begin{bmatrix}
\frac{1}{\eta} \\
1 \\
\frac{1}{\gamma_c + F_{dp}\eta} 
\end{bmatrix}
\]

**Equation 8-58**

\[
MTTF = [1 - B_2']^{-1}D = \frac{F_{dp}\eta}{\gamma_c + F_{dp}\eta} \begin{bmatrix}
1 \\
\frac{\gamma_c}{\gamma_c + F_{dp}\eta} \\
\frac{1}{\gamma_c + F_{dp}\eta} 
\end{bmatrix}
\]

\[
\Lambda_1(\infty) = \frac{F_{dp}\eta^2}{\gamma_c + \eta(1 + F_{dp})}
\]

**Equation 8-59**

Note that from the above equation, it is clear that failure rate due to deadlock can be zero if the deadlock preventive component has reliability of 1. Another condition that can avoid deadlock is to make the collection rate of resources by a process infinity, i.e., \(\gamma_c = \infty\). This is equivalent to making the mean collection time zero. This may seem impractical since any task will take some calendar time. Nevertheless, it is one of the
approaches of prevention, whereby resource allocation is made an atomic event that cannot be subdivided.

For a system that has no deadlock preventive component while the conditions for its occurrence exist, the failure rate is close to what would be obtained by considering the problem similar to race condition. That is, for a deadlock not to occur, a process with resource consumption rate \( \gamma \) must complete its consumption before other processes arriving at the rate of \( \eta \) obtain some of the required resources. Thus, Equation 8-44 can be used to estimate the reliability of the system with respect to occurrence of deadlock:

\[
R_{di} = \frac{\gamma}{\gamma + \eta}
\]

Equation 8-60

Since the above probability can occur at the rate of \( \eta \), the failure rate due to deadlock can be obtained by inserting the above into Equation 8-4, which gives us:

\[
\Lambda = \eta(1 - R_{di}) = \eta \left(1 - \frac{\gamma}{\gamma + \eta}\right) = \eta \frac{\eta}{\gamma + \eta} = \frac{\eta^2}{\gamma + \eta}
\]

Equation 8-61

For a system that has no deadlock protective component, the failure rate from Equation 8-59 is given by:

\[
\Lambda_f(\infty) = \frac{1}{\gamma + \eta(1+1)} = \frac{\eta^2}{\gamma + 2\eta}
\]

When the resource collection rate is significantly greater than the resource demand rate, then both of the above estimates become close to each other.

\[
\Lambda_f(\infty) = \frac{\eta^2}{\gamma + 2\eta} = \frac{\eta^2}{\gamma} \approx \Lambda
\]

Equation 8-62

### 8.3.5 Incompatible Input-Outputs

In a composite structure, the inputs of one component are normally outputs of other components both user and server components. There may appear cases where some inputs provided to components (to which they are required to respond to) lie outside their
domain. This may result from failure of some of the components in the network or environment or due to incompatible interfaces resulting from design errors.

One objective of module-interface relations and connection matrices is ensuring compatibility of input-output types of components in a system. Given, a connection matrix $Q$, we can obtain the set of all possible inputs to each component using Equation 7-29, given as: 

$$T_n = T_i \otimes Q = U_n,$$

where $C$ represents component, $I$ represents input, and $Q$ represents the connection matrix.

The requirement for providing consistent inputs to each component is described by the following condition:

$$I^0_c \subseteq \text{Dom}(C),$$

where $\text{Dom}$ represents the domain of the desired output functions of component $C$.

For a set of components developed as unit, checking this requirement is normally one of the main objectives of syntax analysis. However, when heterogeneous components that do not have common specifications have to be composed from different suppliers, satisfying the relation can be difficult if not impossible. Additionally, for components that interface with the external operational environment, input/output incompatibilities can occur due to lack of complete information about all possible inputs from the operational environment.

Generally, ensuring input/output compatibility requires complete assessment of the operational environment of the system under design- the possible devices it can interface with, the range of signals and inputs the system can receive and the possible size of the data to be generated. In many cases, different modules may be assigned for hiding input/output details and complexities. By doing so, the assessment of undesired events due to input/output incompatibilities can be converted into the reliability of the respective modules.

When the occurrence of certain inputs to a component outside the domain of the component is possible and may result in component failure, the condition may be considered as undesired event that affects system reliability.
\[ UE_{UI} = P\{I_C^0 \not\in \text{Dom}(C)\} \]

Equation 8-63

8.3.6 Incompleteness of Requirements

Some failures may result from incompleteness of system requirement specifications or incomplete coverage of all possible inputs. One can still define reliability functions for uncovered inputs by defining the probability of success per arrival of ‘uncovered inputs’ as 0 and the arrival rate of these uncovered inputs to be \( \eta_{UI} \).

\[ \lambda_{UI} = \eta_{UI} (1 - P) = \eta_{UI} (1 - 0) = \eta_{UI} \]

Equation 8-64

The system reliability function with respect to uncovered inputs is given by:

\[ R_{ei} (t) = e^{-\eta_{UI} t} \]

Equation 8-65

8.3.7 Mis-ordered Events

Component instances are finite state machines that normally require receiving a specific sequence of events that enable them to produce certain required outputs. Most programming language type system and syntax analyses are normally limited to checking input-output data types and they do not guarantee that the clients reaching a server component would be of the required state to communicate with the service provider. For instance, a component may expect its input objects to be in certain specified state when they require its service. This implies that all objects reaching that component must have already received the necessary events that will enable them to undergo state transitions to enable them to respond to component request. If any of the paths such objects take does not provide the necessary events to put the objects in a given state, then an undesired event may occur.

The arrival of objects that have not received the necessary event sequences to a service provider component can thus be considered as undesired events that may cause failure to the service that was supposed to be provided. The failure may only affect the object that was not on the required state, but if the design allows transfer of control from service
provider to client, it may also propagate to the service provider as well as other components that may be waiting for service provision from the same server component.

In sequential systems where all occurrences of events are pre-ordered by design, the ordering of events is deterministic and the system either works from the very beginning or does not work at all. However, in asynchronous concurrent systems, events may occur at an arbitrary order and may need to be synchronized at certain time points. In such systems, the ordering of events can at least partially be determined by some non-deterministic processes such as the relative speed of the various processes. The possible occurrence of mis-ordered events may be described by the following condition.

\[ UE_{u,e,c} = \{ t(e_2) > t(e_1) \} = \{ e_2 > e_1 \} \]

**Equation 8-66**

Reliability analysis of undesired events due to incorrect event orderings requires studying the possible event sequences of the system and identifying the possible undesired sequences. The event-structure introduced earlier can be used for this purpose.

For instance, consider the simplified MSS event flow graph shown in Figure 7-8. From the flow-graph, one can see that objects flowing through the event structure must first pass through event u and e in any order before reaching x. Both u and e represent input resources to a job instance – where e is the object representing the exec component and u the urls of the data to be streamed. x is an event which puts the job order into the exec job queue and can only be fired from job through the statement e.x. Thus, it is logical to assume that event e normally occurs before x. However, event x should not occur unless event u has also occurred prior to e. If this not the case, the next event f that follows x may not be able to get the url from the job object, creating undesired condition for both job and exec. Such undesired events must normally be avoided through design that first checks the state of the object before making transition.

What happens if such restrictions are violated? The ‘bug’ may not be detected through testing. This is because, events e, u, x, and f may come in the expected order for some time and may appear in incorrect order only occasionally. Note that both u and e are events in parallel processes that started jointly at event j. The two processes may have different relative speeds. Let the arrival time of u and f be exponentially distributed with
rates with mean values $\gamma_u$ and $\gamma_f$ both measured in relative to event j. The undesired event condition now becomes a race condition between the two events and the probability of successful operation of the system is:

$$R_{eo} = \Pr\{t(u)-t(j) \leq t(f)-t(j)\}.$$  

Equation 8-67

Equation 8-67 is equivalent to Equation 8-43 and hence, for exponentially distributed arrival times, the reliability of the system with respect to being in the correct event order at event f is:

$$R_{eo} = \frac{\gamma_u}{\gamma_u + \gamma_f}.$$  

Equation 8-68

Of course, the MSS system described in the event flow graph does not have the problem of incorrect event order at f, since every path from start j to f always contains e and u before f. The occurrence of undesired event would have been possible, if the link e-x was not conditioned by event u.

Generally, the event structure can be used to list out all possible externally observable event sequences in a system. If there are paths that have incorrect order with respect to the specification of some of the components in the system, then the probability of encountering these paths can be added to the unreliability of the system.

### 8.4 Service Reliability Structure

Services represent a unit of interaction with the system. A service is normally realized by a composite and there may be more than one way of creating the required component networks.

The need for predicting the reliability of each distinct service was established in section 7.2.5. In designs that allow fault tolerance and degradation by separating different outputs, there can be more than one service reliability function. The required service outputs may be classified into three disjoint sets:

- Primary Outputs: outputs whose incorrectness is considered as service failure
• Secondary Outputs: outputs whose incorrectness is not considered as service failure but rather service degradation
• Irrelevant Outputs: outputs whose state is not considered as part of the service state

The reliability of services is a function of the reliability of the various components that comprise the composite structure, the interaction failure causes that may affect any of the components and their interconnection.

The reliability contribution of the various components to services in itself is dependent on the reliability role each component plays in the service, the specific outputs required from each component in case of multiple-output components and possibly the duty cycle of the component in the service.

### 8.4.1 Component Reliability Roles in Services

Table 8-1 shows possible reliability role of components in a service.

<table>
<thead>
<tr>
<th>Component Role</th>
<th>Description of its Effect on Service Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>A component is said to be critical to a service, if the service is considered to have failed when the component has failed. In an input/output connection graph, a critical component is a component that lies in a path that connects inputs to at least one of the primary outputs and whose fault cannot be tolerated.</td>
</tr>
<tr>
<td>Protective</td>
<td>A component is said to be protective if its role is to protect the service or other components in the network from being exposed to some undesired events that may or may not happen as in the case of synchronizers, deadlock detectors, cryptographic components, antivirus, firewalls, input adaptors, etc.</td>
</tr>
<tr>
<td>Restorative</td>
<td>A component is said to be restorative, if its role is to maintain or restore the states of some of the resources or other failed components in the system.</td>
</tr>
<tr>
<td>Redundant</td>
<td>A component is redundant in a service if its role is to provide its service in parallel to other components either in stand-by or active redundancy mode.</td>
</tr>
<tr>
<td>Supportive</td>
<td>A component is said to be supportive if its failure is considered as Service degradation but not service failure. In a connection graph, a supportive component is a component that does not lie in the path between inputs and primary outputs but between inputs and secondary outputs.</td>
</tr>
<tr>
<td>Irrelevant</td>
<td>A component may be irrelevant for a service if failure of the component is not considered as failure or degradation of the service being analyzed.</td>
</tr>
</tbody>
</table>
The role that components play in a service/system affects their reliability contribution in different ways. The failure rate and reliability functions of critical and redundant components in a service remain unchanged by their role and only the structure function for the service makes the distinction between redundancy and criticality. However, the case is different with components playing other roles – such as protective, restorative or supportive. Thus, one needs to first obtain the effective failure rate and reliability function of these components in any given service. These are obtained below.

### 8.4.1.1 Protective Components:
Protective components play an important role in improving system reliability by reducing the risk of failure. Many of the protective components are included to avoid interaction failures. An example of such protective components is a synchronizer included to avoid race conditions. When these interaction failure causes are considered as virtual components, the mathematical model for the system reliability having a protective component is similar to that of component redundancy. This was shown earlier in Equation 6-9 by:

\[
R_S = R_A + (1 - R_A)R_B = R_A + R_B - R_A R_B
\]

Sometimes, protective components may be added to the system to protect it from external hazards that may or may not happen. For instance, the use of cryptographic components during communication helps to protect the message from being accessed by unauthorized persons with some degree of effectiveness. The reliability model for such components may be obtained as follows.

Let the failure rate of a protective component be given by \( \lambda \), the restoration rate \( \mu \) and the arrival rate for system threats be \( \eta \). Let \( p \) be the probability of failing to protect the occurrence of a threat when a protective component is in operational mode. The service failure rate due to occurrence of system threats can be obtained from the following state model.

![State Transition Diagram for a protective component](image)

**Figure 8-7: State Transition Diagram for a protective component**
Using Semi-Markov analysis (Villemeur, 1992b), we have:

\[
B = \begin{bmatrix}
0 & \frac{\lambda}{\eta p + \lambda} & \frac{\eta p}{\eta p + \lambda} \\
\frac{\mu}{\mu + \eta} & 0 & \frac{\eta}{\eta + \mu} \\
0 & 0 & 0
\end{bmatrix}
\]

\[
B_2' = \begin{bmatrix}
0 & \frac{\lambda}{\eta p + \lambda} \\
\frac{\mu}{\mu + \eta} & 0 \\
\frac{\mu}{\mu + \eta} & 0
\end{bmatrix}
\]

\[
I - B_2' = \begin{bmatrix}
1 & -\frac{\lambda}{\eta p + \lambda} \\
-\frac{\mu}{\mu + \eta} & 1
\end{bmatrix}
\]

\[
[1 - B_2']^{-1} = \frac{(\mu + \eta)(\eta p + \lambda)}{\eta(\lambda + p(\mu + \eta))} \begin{bmatrix}
1 & \frac{\lambda}{\eta p + \lambda} \\
\frac{\mu}{\mu + \eta} & 1
\end{bmatrix}
\]

\[
D = \begin{bmatrix}
\frac{1}{\eta p + \lambda} \\
\frac{1}{\mu + \eta}
\end{bmatrix}
\]

\[
MTTF = [1 - B_2']^{-1} D = \frac{(\mu + \eta)(\eta p + \lambda)}{\eta(\lambda + p(\mu + \eta))} \begin{bmatrix}
1 & \frac{\lambda}{\eta p + \lambda} \\
\frac{\mu}{\mu + \eta} & 1
\end{bmatrix}
\]

\[
\Lambda_1(\infty) = \frac{\eta(\lambda + p(\mu + \eta))}{\mu + \eta + \lambda}
\]

\[
\Lambda_2(\infty) = \frac{\eta(\lambda + p(\mu + \eta))}{\mu + \eta p + \lambda}
\]

For systems starting with state 1.
\[ \Lambda_1(\infty) = \frac{\eta(\lambda + p(\mu + \eta))}{\mu + \eta + \lambda} \]

**Equation 8-69**

The above equation assumes that the protective component may fail to protect the system against an incoming threat with probability \( p \) when it is operational and the system is fully exposed to the threat if it is not used. It is implicitly assumed that the system tolerates any failure originating from the failure of the protective component itself in the absence of external threat. If the connection of the system allows the possibility of system failure due to the failure of the protective component, then the above model must be modified to that effect.

A simple way of taking such critical failure modes into consideration is to represent the failure rate that affects system reliability with \( \lambda_c \) while \( \lambda \) represents the non-critical failure rate. The effect of this in the above model is the transition rate from state 1 to state 3 is \( \eta p + \lambda_c \) in stead of \( \eta p \). Solving the previous sets of equations, the effective failure rate of the protective component is given by:

\[ \Lambda_1(\infty) = \frac{\eta(\lambda + p(\mu + \eta)) + \lambda_c(\mu + \eta)}{\mu + \eta + \lambda} = \frac{\eta \lambda + (\eta p + \lambda_c)(\mu + \eta)}{\mu + \eta + \lambda} \]

**Equation 8-70**

### 8.4.1.2 Monitoring, Backup and Restorative Components

These are components that are configured to monitor, take backups of component states or resource states based on a certain schedule or strategy and restore the states of those components or resources that are not in operating state. Some of these restorative components may be assigned for managing computing resources. An example of such a component is a garbage collector, which restores the primary memory resource, which itself may be considered as a component that is subject to failure due to depletion during system execution. The restoration may be either partial or full. A component that monitors a wireless communication channel and re-connects it upon random disconnection may be considered as a restorative component.
For restorative components that are assigned to restore ‘failed’ components or to replace with ‘new’ components, their execution rate multiplied by the probability of successful restoration may be considered as the restoration rate of the target components.

\[ \gamma_r \cdot p = \mu_c \]  

**Equation 8-71**

### 8.4.1.3 Supportive Components and Derated States

The failure of these components may not be considered as service failure in general. However, the components may interact or interfere with other critical or redundant components. In this case, their possible interference may be included for the reliability assessment. This requires analysis of the design by identifying the various types of interactions and interferences such components may have with other components in the system.

For a supportive component, let us assume three states of operation. State 1 is where the component is fully operational. From this state, it may fail either in its critical mode with probability of \( p \) – going to state two, or it may fail in its derated mode with probability \( (1-p) \) going to state 3. Assume, the restoration from state two to be either to state 1 with probability of success \( q \) or to the derated state with probability of success \( (1-q) \) both with the same restoration rate. Figure 8-8 shows the possible state transition for such a component.

![State transition diagram for a supportive component](image)

Figure 8-8: State transition diagram for a supportive component

Solving the CTMC model obtained from the above state-diagram, the following steady state probabilities can be obtained.
\[
\pi_1 = \frac{\mu \mu_d}{\mu_d + \lambda p \mu_d + \lambda \mu (1 - pq)} \\
\pi_2 = \frac{\lambda p \mu_d}{\mu_d + \lambda p \mu_d + \lambda \mu (1 - pq)} \\
\pi_3 = \frac{\lambda \mu (1 - pq)}{\mu_d + \lambda p \mu_d + \lambda \mu (1 - pq)}
\]

From this, the failure rate of that cause loss of service may be given as:

\[
\Lambda(\infty) = \frac{\lambda p \mu_d}{\mu_d + \lambda (1 - pq)}
\]

**Equation 8-72**

This failure rate comes in series with other critical components in the service. An irrelevant component may be considered as a special case of a supportive component with \( p=0 \).

In case of the degraded mode of operation, the effect may be reflected either by the loss of some of the outputs in the service that are not considered to be critical or by the increase in failure rate of other components. For instance, if the degradation affects timing properties or duty cycles, the effect may be obtained by taking the proportion of time the system is in degraded or full operational mode, which may be obtained as follows

\[
p_o = \frac{\pi_1}{\pi_1 + \pi_3} = \frac{\mu_d}{\mu_d + \lambda (1 - pq)}
\]

\[
p_d = \frac{\pi_3}{\pi_1 + \pi_3} = \frac{\lambda (1 - pq)}{\mu_d + \lambda (1 - pq)} = 1 - p_o
\]

**Equation 8-73**

These probabilities can then be used in other system components whose behavior are affected by the degradation in some way—as in through changes in duty cycles, or increased/decreased failure rates due to missing of deadlines.

**8.5 The Structure Function**

The structure function (Barlow and Proschan, 1996) is a binary function which gives the operational state of a given structure as a function of the operational state of its
components that are assumed to be given in binary form. This function is defined in (Barlow and Proschan, 1996) as follows.

A system consisting of n components is represented by an n-tuple \( x=(x_1, x_2, \ldots, x_n) \). At any time, the performance of each component may be represented by a binary number with value 1 if it is operating and 0 if it is in failed state. The performance of the system may be represented by a binary function \( \Phi(x) \) which is called the \textit{structure function} of the system.

\textit{Monotonic or coherent} structures are structures that satisfy the following conditions.

1. \( \Phi(\mathbf{1})=1 \), where \( \mathbf{1}=(1, 1, \ldots, 1) \)
2. \( \Phi(\mathbf{0})=0 \), where \( \mathbf{0}=(0, 0, \ldots, 0) \)
3. \( \Phi(x) \geq \Phi(y) \), whenever \( x_i \geq y_i \), \( i=1,2,\ldots,n \)

In monotonic structure, the functioning of each component contributes to the functioning of the system.

In its original form, the structure function may not be used to represent the reliability of software components, due to the presence of interaction failures and possible fault-tolerances that make some of the components irrelevant. But, in the sections below, this author adapts the various software structure functions to be used as reliability predictors in a modified form of the structure function.

Specifically, the service-module dependency matrix introduced in 7.2.5 along with the interaction failure causes discussed in the previous section can be used to derive the structure function for each of the possible paths as a function of the state of the modules whose corresponding values in the path are 1.

The derivation process starts by first obtaining a modified service-module dependency matrix, \( S_r(P,M) \) from \( S(P,M) \) based on failure dependency assessment discussed below. Initially, \( S_r(P,M) \) is made the same as \( S(P,M) \).

\textbf{8.5.1 Interaction Concerns as Virtual Components}

As discussed earlier, systems and services may fail due to interaction failure causes besides module primary failures. The interactions may affect one or more components while their occurrence is a result of cumulative effect of almost all components in a
system. A convenient approach in dealing with interaction failures is to represent them as virtual components whose failure distribution function is provided by appropriate models derived based on analysis of the dynamic structure of the system.

The virtual components are appended as new columns on \( S_r(P, M) \) matrix and every path which can be interrupted by the failure of these virtual components will have an entry of value 1 in the respective column. The number of virtual components that are appended in the matrix depends on the possible interactions that may exist among components in each path and their interdependencies. Generally, this number needs to be determined by the number of distinct causes of failure but not on the number of affected components.

For instance, any resource that may be required by one or more component in a path but that is shared among various components in a system in a way that may lead to interaction failures can be represented by a virtual component. Similarly, the presence of a time-sensitive component in a path suggests appending a corresponding virtual component that represents failures due to missing of deadlines. If a component is subjected to more than one source of deadline failure, then all the distinct sources of failure will have to be represented by distinct virtual components. On the other hand, if multiple (two or more) components are subjected to interaction failure due to the same underlying cause, only one virtual component representing the common cause needs to be added to the matrix.

The reliability functions for the virtual component may differ from path to path, due to differences in dynamic properties such as timings and components’ requirement of resources.

For instance, if a path consists of \( n \) components each of which requiring \( m_{i,r} \) units of resource \( r \) and all have to be active at the same time to process a given task, then only one virtual component representing \( r \) needs to appear in the path but its resource requirement is given by the sum of the requirements of each component in the path

\[
R_{rd} = P\left[K_r \leq n_r - \sum_{i=1}^{n} m_{i,r}\right]
\]

Equation 8-74
However, if the usage model of the resource r by the n components is itself in scheduled way in which each of the components are provided the required resources at different times, then, the effective resource requirement is the maximum of the individual requirements since the component that requests the maximum is the first to fail.

$$R_{rd} = P\left\{ K_r \leq n_r - \max_{i=1}^{n} (m_{i,r}) \right\}$$

Equation 8-75

Similarly, if in any given path, there are n-processes that may be waiting for the occurrence of a specific event (for instance completion of a commonly shared server process), but each of them have their own timeline given by $t_{di}$, 0<i≤n, then the combined reliability of not missing the deadline is determined by the process that has the smallest deadline time since it would be the first to fail.

For deterministic deadline times, the deadline reliability may be given as:

$$R_d = P\left\{ T_p \leq \min_{i=1}^{n} (t_{di}) \right\} = F_p\left( \min_{i=1}^{n} (t_{di}) \right)$$

Equation 8-76

For stochastic deadline times, the same min function is used but this time on random variables $T_{di}$ instead of the deterministic values, $t_{di}$

$$R_d = P\left\{ T_p \leq \min_{i=1}^{n} (T_{di}) \right\}$$

Equation 8-77

The above equation can be solved by introducing a random variable $T_d$ defined as:

$$T_d = \min_{i=1}^{n} (T_{di})$$

Equation 8-78

Let $F_d$ represent a cumulative distribution function and $G_d$ represent the corresponding right tail distribution function ($G_d(t)=1-F_d(t)$). Then, the distribution function for $T_d$ may be obtained as:
If the deadline times are known to have exponential distribution with mean $\gamma_{di}$, then the distribution of $T$ will also be exponentially distributed with mean $\gamma_d$ given as the sum of $\gamma_{di}$ as shown below.

$$G_d(t) = 1 - F_d(t) = P[T_d > t] = P[T_{d1} > t \land T_{d2} > t \land ... T_{dn} > t] = \prod_{i=1}^{n} P[T_{di} > t] = \prod_{i=1}^{n} G_{di}(t)$$

**Equation 8-79**

Hence, if $T_p$ is also exponentially distributed with mean $\gamma_p$, then from Equation 8-37 and Equation 8-74, the reliability with respect to not missing a deadline by any one of the involved processes can be obtained as

$$R_d = \frac{\gamma_p}{\gamma_p + \gamma_d} = \frac{\gamma_p}{\gamma_p + \sum_{i=1}^{n} \gamma_{di}}$$

**Equation 8-80**

Equation 8-77 and Equation 8-81 are applicable in cases where the $n$-independent processes wait at the same time for a commonly shared event. An example of this would be $n$-consumer or reader processes waiting for the completion of a specific instance producer or writer process.

However, if each of the $n$-components is subjected to deadline failure while working with independent instances of the same module resulting in $n$-different producer processes that have identical distribution function, then the reliability with respect to the missing of a deadline will be equivalent to a series connection of the reliability of missing a deadline by each of the components.
For deterministic deadline time, the equivalent deadline reliability is given by:

\[
R_d = P\left\{ T_p \leq t_{d1} \land T_p \leq t_{d2} \land \ldots \land T_p \leq t_{dn} \right\}
\]

\[
R_d = P\left\{ T_p \leq t_{d1} \right\} P\left\{ T_p \leq t_{d2} \right\} \ldots P\left\{ T_p \leq t_{dn} \right\}
\]

\[
R_d = \prod_{i=1}^{n} F_p(t_{di})
\]

**Equation 8-82**

For independent stochastic deadline times, we have

\[
R_d = P\left\{ T_p \leq T_{d1} \land T_p \leq T_{d2} \land \ldots \land T_p \leq T_{dn} \right\}
\]

\[
R_d = P\{ T_p \leq T_{d1} \} P\{ T_p \leq T_{d2} \} \ldots P\{ T_p \leq T_{dn} \}
\]

\[
R_d = \prod_{i=1}^{n} P\{ T_p \leq T_{di} \}
\]

**Equation 8-83**

For the case of exponentially distributed deadline and exponentially distributed process times, the above equation will give us

\[
R_d = \prod_{i=1}^{n} \frac{\gamma_p}{\gamma_p + \gamma_{di}}
\]

**Equation 8-84**

For normally distributed deadline and processing times, with respective mean and standard deviation parameters given by, \((\mu_{di}, \sigma_{di})\) & \((\mu_p, \sigma_p)\), consider normally distributed random variables \(T_i = T_p - T_i\) with mean and standard deviations given by:

\[
\mu_i = \mu_p - \mu_{di} \quad \text{and} \quad \sigma_i = \sqrt{\sigma_p^2 + \sigma_{di}^2}
\]

The reliability with respect to not missing deadline by any one of the \(n\)-processes can be given by:

\[
R_d = \prod_{i=1}^{n} \Phi_{T_i}(0)
\]

where

\(\Phi_{T_i}(0)\) is the value of the cumulative normal distribution with mean and variance for the random variable \(T_i\).
Equation 8-85
The case of dependent random variables in the above cases requires the use of conditional probability evaluations. For instance, Equation 8-83 needs to be replaced by the following:

\[
R_d = P\left\{ T_p \leq T_{d_1} \land T_p \leq T_{d_2} \land \ldots \land T_p \leq T_{d_n}\right\} \\
= P\left\{ T_p \leq T_{d_1}\right\} P\left\{ T_p \leq T_{d_2} \mid T_p \leq T_{d_1}\right\} \ldots P\left\{ T_p \leq T_{d_n} \mid T_p \leq T_{d_1} \ldots T_p \leq T_{d_{n-1}}\right\} \\
= P\{T_p \leq T_{d_i}\} \prod_{i=2}^{n} P\{T_p \leq T_{d_i} \mid T_p \leq T_{d_1} \land \ldots \land T_p \leq T_{d_{i-1}}\} \\
\]

Equation 8-86

8.5.2 Series Connection of Modules in a Path
Usually, all the modules encountered in a given path may have to perform correctly for the path to function correctly. In such cases, the structure function for the path is given as a product of the Boolean variables representing the operational state of the modules, implying series connection. This series connection of modules on paths does not however exclude the possible existence of redundant paths for any given service or internal redundancies within the modules. In all the paths where this condition apply, the contents of \( S_R(P,M) \) remain unchanged (i.e. equal to that of \( S(P,M) \)).

8.5.3 Modular Redundancies in Services
Modular redundancies refer to conditions where there are more than one module that are diverse in their implementation in a system that map from the same set of required interfaces to the same set of provided interfaces, for instance using different coordinate systems that could potentially lead to variations in failure points. Whenever such types of redundancies exist in a system, the services that use the redundant modules will have separate paths corresponding to each of the modules in the redundant set. As a result, each module in any given path may appear as a critical module to the path and hence is considered as a component connected in series with other components in the path. Hence, the contents of \( S_R(P,M) \) remain unchanged (i.e. equal to that of \( S(P,M) \)). The effect of modular redundancies is reflected by having different alternative paths that provide the same service.
In some cases, the existence of the redundant modules may be hidden from the structure. This occurs when the existence of the redundant modules is made to be the secret of a super module that is externally accessible as a single unit and the choice among the redundant modules is not observable to the clients of the module. In this case, the reliability of the super module is considered to combine the reliability of the internal redundancies.

Unlike hardware redundancies, the effectiveness of software redundancies is limited by statistical dependencies among redundant versions that arise for various reasons. They may be effective in covering simple programming errors, but unreliability that arises due to complexity of problem domains and unavailability of tractable algorithmic solutions, limitation of computing resources, and specification of requirements are likely to equally affect redundant modules simultaneously. Thus, assessment of modular redundancy in software must be done based on analysis of statistical dependencies among diverse modules.

8.5.4 Fault Tolerance and Module Irrelevance

A system may be composed of a set of modules, where some of them perform critical system function while others provide support functions. One way of improving system reliability is through fault-tolerant design. If a service is designed to tolerate the failure of a module or component, then the module or component is said to be irrelevant to the service as far as reliability is concerned.

The need for making components irrelevant to services arise in situations where we would like to use the functions of a component as long as it is operational but we do not want the component to induce failure to other parts of the system, especially primary functions, if it fails. An example candidate of such component in a software system may be a logger/tracer. Usually, the purpose of logging is for improved diagnoses of software in case of failure. During normal system operation, the logger may be allowed to log events as long as it is available or as long as its resource requirement does not induce failure to other components. But, if the logger fails to function properly, or if the logger interferes with other components through resource consumption, then it may need to be removed from the system, while other components continue to operate.
For a module to be irrelevant to a service, its design must ensure that the probability for the failure mode causing damage to shared resources should ideally be zero and the outputs of the component be not critically important for the service. This may be achieved by using unidirectional connectors, where support modules are allowed only to ‘listen-to’, ‘read-outputs’ or ‘accept inputs’ from critical system modules without affecting the normal functioning of the critical modules. Such modules may be considered to be irrelevant to the service path so long as their outputs are not used as part of or input to any of the critical modules or their failure does not cause failure to the critical modules in the path.

When there are support modules that are irrelevant to services, then this may be indicated in the modified dependency matrix, \( S_{R}(P,M) \), by setting the values of the irrelevant modules to 0 in the paths where they are irrelevant.

In some cases, the reliability assessment of support services on their own may be needed in which case we can add new relevant paths to \( S_{R}(P,M) \) that contain the support modules and any others which they depend on.

### 8.5.5 Multimode Operation of Modules and Service Degradation

There are cases where complete inclusion or exclusion of modules as described in the previous section may not precisely describe the state of paths. This is when the failure of some of the modules is tolerated albeit with degradation of the path structure which may include increased chance of failure on other modules. In this case, the modules cannot be assumed to be statistically independent, and reliability assessment should consider the exact form of dependency among them.

In many cases, the interdependencies among module failures are indirectly observed via interaction failures through shared resources. Many of the interaction failures have been expressed in the form of probability distribution functions.

For instance, assume a system uses two modules ‘M_i’ and ‘M_j’ but tolerates the failure of ‘M_i’ although it causes degradation on ‘M_j’. The system is considered to be operational as long as M_j operates. The reliability can be obtained using basic probability laws as follows (where + stands for ‘OR’ or Union):
\[ P(M_iM_j + M_j) = P(M_iM_j) + P(M_i \overline{M_j}) \cdot P(M_j \mid M_i \overline{M_j}) \]
\[ = P(M_iM_j) + P(M_j \overline{M_i}M_j) = P(M_iM_j) + P(M_j(\overline{M_i} + \overline{M_j})) \]
\[ = P(M_iM_j) + P(M_j \overline{M_i}) = P(M_i)P(M_j \mid M_i) + P(M_j \overline{M_i})P(M_j \mid \overline{M_i}) \]
\[ = P(M_i)P(M_j \mid M_i) + (1 - P(M_i))P(M_j \mid \overline{M_i}) \]

**Equation 8-87**

One way of including such dependencies in the \( S_r(P,M) \) matrix is by considering the different modes of operations of dependent modules as separate modules. In the above example, \( M_j \) may be considered as two modules, \( M_{ji1} \) and \( M_{ji0} \), where \( M_{ji1} \) represents the state of module \( M_j \) when \( M_i \) is operational, i.e. \( M_i \mid M_j \), and \( M_{ji0} \) represents the state of module \( M_j \) when \( M_i \) is not operational, i.e. \( M_i \mid \overline{M_j} \). In the service-module dependency matrix, the column \( M_j \) is relabeled as \( M_{ji1} \) and a new column representing \( M_{ji0} \) as well as a new row/path representing the path for the degraded mode of operation are added. The contents of new row are copied from the original path with changes at \( M_i = -1 \) and \( M_{ji1} = 0 \) and \( M_{ji0} = 1 \), which is the only entry in the new column.

\[ P(M_iM_j + M_j) = P(M_i)P(M_{ji1}) + (1 - P(M_i))P(M_{ji0}) \]

**Equation 8-88**

If the failure of a module \( M_i \) degrades more than one module including the virtual components, then each of the modules that are affected by \( M_i \) need to be represented by dual modes similar to \( M_j \) as discussed above.

A special case of Equation 8-88 is when the reliability of \( M_{ji1} = 1 \), i.e., if the conditional probability of successful operation of \( M_j \) is 1 on the condition that \( M_i \) is operational. In this case, the equation is written as follows.

\[ P(M_iM_j + M_j) = P(M_i)P(M_{ji1}) + (1 - P(M_i))P(M_{ji0}) \]
\[ = P(M_i) + (1 - P(M_i))P(M_{ji0}) = P(M_i) + P(M_{ji0}) - P(M_i)P(M_{ji0}) \]

**Equation 8-89**

This equation is equivalent to statistically independent redundant components and can also describe the behaviour of protective components discussed earlier. In this case, it is not necessary to split \( M_j \)'s column into two as \( M_j \) can represent \( M_{ji0} \) i.e. the composite reliability in the absence of a protective component. However, it is necessary to replicate
each of the rows or paths that contain $M_i$ and $M_j$ so that they appear as parallel/redundant paths, where the value of $M_i$ is set to 1 while that of $M_j$ is set to 0 on the original path and the setting for the two components is reversed in the redundant path. The two paths need to be documented as parallel paths so that appropriate formula for combining the structure functions is used later on.

Components may not be fully relevant or completely irrelevant to a service, but partially relevant. Even those that may be desired to be irrelevant to the service may directly or indirectly affect the service if their effect on shared resources induces failure to other relevant components in the composite. For improved assessment of service reliability from component reliability, it is important to define component reliability function based on the various failure modes.

Software components may have different failure modes among which include:

- Failure to start execution or failure to respond for a request
- Provision of incorrect output that is within the range of its transfer function and with no indication of error
- Failure causing damage on shared resources, e.g. by making resources inaccessible to other components

The above list of failure modes is not exhaustive. Creating an exhaustive list of all failure modes of each module will result in a complex model that will not be tractable for a meaningful reliability analysis. One way of simplifying the analysis without losing information is by classifying the various failure modes of each component into three disjoint sets based on its effect on the system under study. These classes are: \{operational mode, derated mode, critical failure mode\}. A component in operational mode is able to provide correct outputs as per its specification whereas a component in derated mode may not be able to provide all its required outputs as specified in its requirements but its effect on the system understudy is that of degradation but not complete failure. A component is in critical failure mode if its failure mode causes failure to the system under study.

Since the classification is based on a component’s effect on system outputs, it is possible that a component that is excluded from a system may be considered to be in derated mode.
while a component that provides non-detectable incorrect output within its output range or a component that ‘damages’ a shared resource may be in critical failure mode.

The reliability function of multimode components is normally obtained using a mixture model (Wallace R. Blischke and Murthy, 2000 pp.115-116). Let a component have K failure modes. Then, its reliability function can be obtained as:

$$R(t) = \sum_{i=1}^{K} p_i R_i(t)$$

**Equation 8-90**

where $R_i(t)$ is the reliability function for the $i^{th}$ failure mode and $p_i$ is the probability of occurrence of the $i^{th}$ failure mode.

If the different failure modes of a component do not affect the state of other components in the system, then a multimode component can simply be considered as a single component with its reliability function given by Equation 8-90. Hence, its state in the $S_r(P, M)$ matrix remains unchanged.

However, if the different modes of operation influence other components in the system, or results in variations in system outputs, then the component can be represented by different mutually exclusive components having different reliability functions. Such components may be distinctly represented by their modes, i.e., the component’s column can be split into K columns and any row that contains the component is replicated into K rows. In each of these K rows, only one of the modes has value 1 while the others are set to 0. Any component that is s-dependent on the mode of the multimode component may have to be appropriately split and represented in multiple modes so that its appropriate modes are used in each of the K-rows.

Additionally, some of the modes may exist only if some other components in the system are operational. In this case, the values of the respective components need to be set to 1 in the rows representing the modes.

If one of the modes say the $K^{th}$ mode of the component is critical failure mode that affects the system output, then the rows corresponding to the $K^{th}$ mode need to be removed since the system has no operational path in this mode and all outputs/paths that can possibly be
affected by this mode should have -1 entries in their $K^{th}$ column. This indicates that the success of each row is on the condition that a critical failure mode has not occurred.

The different paths corresponding to the $K$ failure modes need to be documented as mutually exclusive multimode paths, so that appropriate reliability formulas are used. This may be done by partitioning the path sets in $S_K(P,M)$ into mutually exclusive set groups.

Example: consider a parallel system of two components: A and B with the modified service-module dependency matrix given as:

$$S_K(P,M) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Let each of these two components have three modes- normal mode ($x_n$), derated mode ($x_d$) and critical mode ($x_c$). The critical failure mode of any component results in system failure. In the absence of a critical failure mode, the system can be operational if there is at least one component in normal state. The system operational paths with the different modes of operation can be obtained by splitting both A and B into three columns each and adding two rows for each component (since the third rows for critical failure modes not an operational path).

$$S_K(P,M) = \begin{bmatrix} A_n & A_d & A_c & B_n & B_d & B_c \\ 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

### 8.5.6 Evaluating the Structure Function

From the modified service-module dependency matrix, $S_R$, and a vector of Boolean variables representing the state of the modules, $X$, the reliability structure function, in vector form, $\Phi$, consisting of individual structure functions $\Phi_i$, corresponding to the different paths may be evaluated:
\[ \Phi(X) = S_R \circ X^T \]

where

\[ \Phi_r(X) = \prod_{j=1}^{m} s_{ij} \circ x_j \]

and \( \circ \) is a selector operator defined by the function:

\( \{1 \circ x = x, \ 0 \circ x = 1,\ 1 \circ x = 1 - x\} \), and

\[ \prod \] represents Boolean product.

**Equation 8-91**

For the example discussed in the previous section, we have:

\[
S_R(P, M) = \begin{bmatrix}
A_n & A_d & A_c & B_n & B_d & B_c \\
1 & 0 & 0 & 0 & 0 & -1 \\
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & -1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

\[
\Phi(X) = S_R \circ X^T
\]

\[
\Phi(X) = \begin{bmatrix}
\Phi_1 & 1 & 0 & 0 & 0 & 0 & -1 \\
\Phi_2 & 0 & 1 & 0 & 1 & 0 & 0 \\
\Phi_3 & 0 & 0 & -1 & 1 & 0 & 0 \\
\Phi_4 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
x_{An} \\
x_{Ad} \\
x_{Ac} \\
x_{Bn} \\
x_{Bd} \\
x_{Bc} \\
\end{bmatrix}
= \begin{bmatrix}
x_{An} \cdot (1 - x_{Bc}) \\
x_{Ad} \cdot x_{Bn} \\
x_{Ac} \cdot (1 - x_{Ac}) \cdot x_{Bn} \\
x_{Bd} \cdot x_{An} \cdot x_{Bd} \\
x_{Bc} \\
\end{bmatrix}
\]

**8.5.6.1 Boolean Reductions in Structure Functions**

In evaluating structure functions that represent component states with Boolean variables, the following Boolean reductions need to be used:

1. \( (x_i)^m = x_i \) since \( x \cdot x = x \)
2. \( x_i + 1 = 1 \)
3. \( x_i \cdot 0 = 0 \)
4. \( x_i \oplus x_j = 1 \Rightarrow x_i \cdot x_j = 0 \)
The subset operators in reduction rules 5 is used to represent when a component is a subset of another component, as for instance, the relation between an access program and a module, or a component and a composite.

**8.5.7 Combining Different Paths:**

The question of how to combine the different paths of the structure function into service and system reliability structure functions depends on the design of the system and how the different paths are used. Different ways of combining the structure functions of paths depending on the design decisions are discussed below.

1. **Mutually-Exclusive Multimode Operations:** some of the paths that have been identified in the modified service-module dependency matrix exist in mutual exclusive mode containing 1 and -1 in one or more of their modules. The structure function of such paths can simply be added together to get the structure function of the service represented by the mutually exclusive paths. If \( M_i \) takes the value 1 in the path structure function \( \Phi_{M_{i1}}(X) \), and -1 in the path structure function, \( \Phi_{M_{i0}}(X) \), then the two can be added together to give one structure function, \( \Phi_{M_i}(X) \) i.e.

\[
\Phi_{M_i}(X) = \Phi_{M_{i1}}(X) + \Phi_{M_{i0}}(X).
\]

**Equation 8-92**

This is because of the well known property of the structure function where any structure function can be obtained as a linear combination of two structure functions decomposed on any of its Boolean variables as given by:

\[
\Phi(X) = x_i \Phi^1_i(X) + (1 - x_i) \Phi^0_i(X)
\]

where \( \Phi^b_i(X) \) is the structure function where \( x_i \in X \) takes the value \( b \in \{0, 1\} \).

In our case, \( x_i \) and \( (1-x_i) \) are already included in the evaluation of the respective path structure functions.

\[
\Phi_{M_{i1}}(X) = x_i \Phi^1_i(X)
\]

\[
\Phi_{M_{i0}}(X) = (1 - x_i) \Phi^0_i(X)
\]
Similarly, all other mutually exclusive rows that might have arisen due to multimode operation of components or other known reasons can be combined through summation operation.

If there are $K$ mutually exclusive paths, then their combined structure function can be given as:

$$\Phi(X) = \sum_{i=1}^{K} \Phi_i(X)$$

Equation 8-93

For the example given previously, we have:

$$\Phi(X) = \left[ \begin{array}{cccccc} A_n & A_d & A_c & B_n & B_d & B_c \\ \Phi_1 & 1 & 0 & 0 & 0 & 0 & -1 \\ \Phi_2 & 0 & 1 & 0 & 1 & 0 & 0 \\ \Phi_3 & 0 & 0 & -1 & 1 & 0 & 0 \\ \Phi_4 & 1 & 0 & 0 & 0 & 1 & 0 \end{array} \right] \left[ \begin{array}{c} x_{An} \\ x_{Ad} \\ x_{Ac} \\ x_{Bn} \\ x_{Bs} \\ x_{Bd} \end{array} \right] = \left[ \begin{array}{c} x_{An}(1-x_{Bc}) \\ x_{Ad}x_{Bn} \\ x_{Ac}(1-x_{Ac})x_{Bn} \\ x_{Bd}x_{Bd} \end{array} \right]$$

From the above, we can see that path 1 and path 2 are in mutual exclusion and so are path 3 and 4. Hence, the structure function can be further simplified by summing the mutually exclusive paths together which gives us:

$$\Phi(X) = \left[ \begin{array}{l} \Phi_{11} \\ \Phi_{12} \end{array} \right] = \left[ \begin{array}{c} x_{An}(1-x_{Bc}) + x_{Ad}x_{Bn} \\ (1-x_{Ac})x_{Bn} + x_{An}x_{Bd} \end{array} \right]$$

For a multi-state component, $c$, that has $m$-mutually exclusive states, at any time, the component can be only in one of its $m$-states. If the $m$-states are represented by $m$-binary variables, then we can write the following equation:
\[
\sum_{i=1}^{m} x_i = 1
\]

or
\[
1 - x_j = \sum_{i=1}^{m} x_i
\]

Equation 8-94

Thus, for the structure function of the example given above, we can write,
\[
1 - x_{Be} = x_{Bo} + x_{Bn}
\]
\[
1 - x_{Ac} = x_{Ad} + x_{An}
\]

Equation 8-95

Substituting this in the structure function, we get
\[
\Phi(X) = \begin{bmatrix}
\Phi_{11} \\
\Phi_{12}
\end{bmatrix} = \begin{bmatrix}
x_{An}(x_{Bn} + x_{Bd}) + x_{Ad}x_{Bn} \\
(x_{An} + x_{Ad})x_{Bn} + x_{An}x_{Bd}
\end{bmatrix} = \begin{bmatrix}
x_{An}x_{Bn} + x_{An}x_{Bd} + x_{Ad}x_{Bn} \\
x_{An}x_{Bn} + x_{Ad}x_{Bn} + x_{An}x_{Bd}
\end{bmatrix}
\]

Observe that the two rows in the above structure function have identical expressions and hence the two paths will always have the same state. Hence, effectively, the structure function has only a single path since if one fails, the other also fails. We can thus evaluate the structure function as:
\[
\Phi(X) = x_{An}x_{Bd} + x_{An}x_{Bn} + x_{Ad}x_{Bn}
\]

2. Non-Redundant Non-Mutually Exclusive Paths: due to the variations in the modules, it is possible for the different paths to represent different services, where failure of any one of the paths is considered as system failure. The system structure function in this case is obtained as a series connection of the different path structure functions
\[
\Phi(X) = \prod_{i=1}^{p} \Phi_i(X)
\]

Equation 8-96

If the modified dependency matrix is a binary matrix (with no -1 entries), one can alternatively, first obtain a vector that represents system structure function as a disjunction of all the path vectors in the service-module dependency matrix and then
apply the dot product with the component state vector to obtain the structure function.

\[ \Phi(X) = \left( \bigvee_{i=1}^{p} S(\Phi_i, M) \right) \circ X^T \]

**Equation 8-97**

3. Redundant Paths: it is possible for the different paths to represent redundancies where modules vary from each other in their internal structure, mainly in the subcomponents they use, the algorithms they implement, and/or the inputs or resources they depend on, and the paths vary in the combination of the modules they use. A system containing such redundant paths is considered to be operational so long as any one of the paths is operational. In terms of structure function, this may be stated as follows:

\[ \Phi(X) = 1 - \prod_{i=1}^{p} (1 - \Phi_i(X)) \]

**Equation 8-98**

Generally, the above three possibilities need not exist in mutual exclusion with each other. Redundancies may be provided for critical or primary services, while non-critical or secondary services may exist only in non-redundant paths that may or may not be mutually exclusive. Hence the reliability structure function of systems and services may be provided in various combinations of the above equations (Equation 8-92-Equation 8-98). Statements about path combinations may be given using Boolean expressions involving ‘or’, ‘and’, ‘not’ and ‘xor’ operators. For \( P, P_i, \) and \( P_j \) representing paths and \( \Phi, \Phi_i, \Phi_j \), representing their respective structure function, given path composition Boolean expressions, we can obtain the corresponding structure function using the following conversion:

\[ P = P_i \oplus P_j \Rightarrow \Phi(X) = \Phi_i(X) + \Phi_j(X) \]
\[ P = P_i \lor P_j \Rightarrow \Phi(X) = 1 - (1 - \Phi_i(X))(1 - \Phi_j(X)) = \Phi_i(X) + (1 - \Phi_i(X))\Phi_j(X) \]
\[ P = P_i \land P_j \Rightarrow \Phi(X) = \Phi_i(X)\Phi_j(X) \]
\[ P = \neg P_i \Rightarrow \Phi(X) = 1 - \Phi_i(X) \]
8.6 Reliability Function Evaluation

The reliability of the various services and systems can be evaluated from the structure function. The evaluation, however, must consider whether statistical independence assumption of failures can be made or otherwise.

8.6.1 Statistically Independent Failure Causes /Undesired Events

In cases where statistical independence of failure behavior (corresponding to the primary failure causes of modules), can be assumed, (this is generally difficult for software components), evaluating the reliabilities of each of the different services and systems is a simple substitution of the reliability vector, \( P = [p_1, \ldots, p_p] \), in place of the Boolean state vector \( X = [x_1, \ldots, x_p] \) of the modules.

\[ R = \Phi(P) \]

\( R = R(P) \) is called the reliability function.

Example: for the example given earlier, if the system is not considered to have failed as long as there is one operating path, then we can first evaluate the structure function followed by the system reliability as shown below:

\[
\Phi(X) = 1 - \prod_{i=1}^{4} (1 - \Phi_i(X)) = 1 - (1 - x_1 x_3)(1 - x_1 x_4)(1 - x_2 x_3)(1 - x_2 x_4) \\
\Phi(X) = x_1 x_3 + x_1 x_4 + x_2 x_3 + x_2 x_4 - x_1 x_3 x_4 - x_1 x_2 x_3 x_4 - x_1 x_2 x_3 - x_1 x_2 x_4 - x_2 x_3 x_4 \\
+ x_1 x_2 x_3 x_4 + x_1 x_2 x_3 x_4 + x_1 x_2 x_3 x_4 + x_1 x_2 x_3 x_4 - x_1 x_2 x_3 x_4 \\
R = \Phi(P) = p_1 p_3 + p_1 p_4 + p_2 p_3 + p_2 p_4 - p_1 p_3 p_4 - p_1 p_2 p_3 - p_1 p_2 p_4 - p_2 p_3 p_4 + p_1 p_2 p_3 p_4 \\
R = \Phi(P) = p_1 p_3 + p_1 p_4 + p_2 p_3 + p_2 p_4 - p_1 p_3 p_4 - p_1 p_2 p_3 - p_1 p_2 p_4 - p_2 p_3 p_4 + p_1 p_2 p_3 p_4
\]

8.6.2 Dependent Failure Causes /Undesired Events

If statistical independence assumption among the basic undesired events cannot be justified, then the reliabilities of each path and their combinations need to be evaluated using conditional probabilities. This can be facilitated by applying a flow graph based approach to reliability evaluation (Burroughs and Happ, 1962) on the modified service-module dependency matrix.

Let \( x_1, x_2, \) and \( x_3 \) represent operating states of three components and let their complements, \( \overline{x}_1, \overline{x}_2, \) and \( \overline{x}_3 \) represent failed states of the components. The graphs in
Figure 8-9 depict the representation of series and parallel connections of the three components in standard flow graph models.

A. Series Connections: \( p(x_1, x_2, x_3) \)

B. Parallel Connections: \( p(x_1 + x_2 + x_3) \)

The use of conditional probability helps to include the interdependence among different components in reliability evaluation. For instance, in software, creating primitive component redundancy through replication is trivial. In cases where the different components reside in two different redundant paths, it is possible that they may not fail at the same time, since their input conditions may vary from path to path. If \( x_1 \) and \( x_2 \) are replicated components in two parallel paths, then the probability of any one of them operating is given by:

\[
p(x_1 + x_2) = p(x_1) + p(x_1') \cdot p(x_2 | x_1') = p(x_1) + (1 - p(x_1)) \cdot p(x_2 | x_1').
\]

Depending on the analysis of causes of failure for the parent module and the variations in the input conditions in the two paths, assumptions can be made, starting from completely dependent relationship to independence. The most pessimistic assumption is made by \( p(x_2 | x_1') = 0 \), i.e. if \( x_1 \) fails then the probability of \( x_2 \) being operational is zero. This gives \( p(x_1 + x_2) = p(x_1) \). If independence assumption can be justified, then \( p(x_2 | x_1') = p(x_2) \) and hence \( p(x_1 + x_2) = p(x_1) + p(x_2) - p(x_1)p(x_2) \), which is the formula for parallel connection of two independent components.
8.7 System Failure Rate Estimation

8.7.1 System Failure Rate Estimation Using the Service-Module Dependency Derivative Matrix

Let \( R(p) \) be a reliability function for a structure consisting of \( n \)-independent non-identical components, with reliability function of each component given by \( p_i = F_i(t) \). The relation between the structure failure rate and component failure rates may be obtained as follows (Barlow and Proschan, 1996).

\[
\Lambda(t) = \frac{R'(P)}{R(P)} = -\sum_{i=1}^{n} \frac{\partial R}{\partial p_i} p_i' = -\sum_{i=1}^{n} p_i \frac{\partial R}{\partial p_i} = -\sum_{i=1}^{n} \frac{p_i \partial R}{R(P)} \frac{\partial p_i}{p_i} \\
= \sum_{i=1}^{n} \frac{p_i \partial R}{R(P)} \lambda_i(t)
\]

Equation 8-99

However, the component failure rate was related to ‘per demand’ reliability earlier in section 8.2.1 as:

\[
\hat{\lambda}_i = \eta(1 - R_i)
\]

Equation 8-100

Note that Equation 8-100 was formulated by assuming \( p_i \) to be a reliability function whereas Equation 8-99 assumes \( R_i \) to be probability of success per demand. However, the two are approximately equal under the following assumptions.

Software components normally operate per demand where the demand rate may be assumed to be exponentially distributed. This exponential distribution of demand gives rise to exponential distribution of failure rate since we assume that the probability of failure per demand is a constant that does not depend on time. The reliability function of a component is thus an exponential function. But the reliability function in a given service cycle \( \tau \), may be approximated by a first order Taylor series representation (valid for small \( \eta(1 - R_i)\tau \)), as shown below:
\[ p_i(\tau) = \bar{F}_i(\tau) = e^{-\lambda_i \tau} \approx e^{-\eta_1 (1 - R_i) \tau} \approx 1 - \eta_1 (1 - R_i) \tau \]

**Equation 8-101**

For \( \tau \) representing service cycle time, we have \( \eta \tau = 1 \). Hence, Equation 8-101 gives us

\[ p_i(\tau) \approx 1 - \eta_1 (1 - R_i) \tau = 1 - (1 - R_i) = R_i \]

**Equation 8-102**

The component demand rate to be used in Equation 8-100 to estimate its failure rate in a particular service is dependent on the mode of sharing of the component by different services. If various services share state-full components (or preferably state variables) that may allow cross-service interference, then the component demand rate is obtained as a sum of the demand rates for the component from all services that share it. This is because: failure of a component while processing input received from one service can result in unavailability of the component for other services.

However, this kind of sharing can usually be avoided through design restrictions that only allow the sharing of modules among services through component replications and/or independent instantiations that avoid cross-service interferences. In this case, the different services sharing modules can be considered to be using different components where the primary or secondary failure of any component in a service affects only those components included in that specific service. For such sharing modes, the component demand rate in each service is equal to the service demand rate for the component from that specific service. Hence, Equation 8-100 may be re-written as \( \lambda_i = \eta_{\text{service}} (1 - p_i) \) and be substituted in Equation 8-99 to get:

\[
\Lambda(t) = \sum_{i=1}^{n} p_i \frac{\partial R}{\partial p_i} \eta_{\text{service}} (1 - p_i) = \frac{\eta_{\text{service}}}{R(P)} \sum_{i=1}^{n} p_i (1 - p_i) \frac{\partial R}{\partial p_i}
\]

**Equation 8-103**

Given a structure function for a service, \( \Phi(x) \), it is known (for monotonic structures) that the reliability function can be obtained as: \( R = \Phi(p) \), where the vector variable \( x \) is substituted by the probability vector \( p \). The form of expressions in the reliability and structure functions remain the same, and hence the following holds true:

\[
\frac{\partial R}{\partial p_i} = \frac{\partial \Phi}{\partial x_i} (p) = \frac{\partial \Phi(p)}{\partial x_i}
\]
Therefore, Equation 8-103 can be re-written as follows.

$$\Lambda(t) = \frac{\eta_{\text{service}}}{\Phi(P)} \sum_{i=1}^{n} p_i (1 - p_i) \frac{\partial \Phi(p)}{\partial x_i}$$

**Equation 8-105**

Observe that Equation 8-105 has three parts:

1. \( \frac{\eta_{\text{service}}}{\Phi(P)} \) which is a function of service property only (service usage profile and reliability function),
2. \( p_i (1 - p_i) \) which is a function of component reliability only.
3. \( \frac{\partial \Phi(p)}{\partial x_i} \) which is a function of structure or service-module dependency only

\( \partial R / \partial p_i = I_{Bi}(i) \) is the Birnbaum reliability importance measure (Wallace R. Blischke and Murthy, 2000). Its significance is increasing \( p_i \) by an amount \( \Delta \) increases reliability \( R \) by \( I_{Bi}(i) \Delta \).

Note that Equation 8-105 gives the contribution of a component to service failure rate as 0 if either \( p_i = 1 \) or \( p_i = 0 \). The former is correct since component failure rate is zero if the reliability (per demand) of the component is 1. However, the later case, where the contribution of unreliable component (\( p_i = 0 \)) to service failure rate being zero is not correct. This is because the first order Taylor series approximation given in Equation 8-101 is valid only for small \( \eta(1 - R_i) \tau \), i.e small \( 1 - R_i \), or \( R_i \) near 1. If \( R_i \) near 0, then the approximation is not valid and neither are Equation 8-103 and Equation 8-105.

For any given service, the last two parts may be represented in vector form while the first one is given by a scalar value. Then, Equation 18-7 can be obtained as a dot product of the two vectors multiplied by the scalar value.

For a system providing \( m \) services from a composition of \( n \) modules, whose probability of success is given by an \( n \times 1 \) vector \( p \), the service demand rate may be represented by an \( m \times 1 \) vector, \( H \), the service reliability function by an \( m \times 1 \) vector \( \Phi(p) \), and the service-module dependency derivative by an \( m \times n \) matrix \( D \), where \( D = \partial \Phi / \partial x_i \). Then, the failure rate of the various services in the system can be given by:
\[ \Lambda_s = (D(p) \times (p \cdot (1 - p))) \cdot (H/\Phi(p)) \]

Equation 8-106

\( H/\Phi(p) \) is a vector of size \( m \) whose elements are formed from the ratio of service demand rate to service reliability function.

\( 1 \) is a unit vector of size \( nx1 \) (same as that of \( p \))

\( \cdot \) is an element by element product operator on vectors of equal size, i.e.

for \( x \) & \( y \) vectors of size \( n \), \( z = x \cdot y \), then \( z_i = x_i \times y_i \), for \( 1 \leq i \leq n \)

Example:
Consider the example given above.

\[
\Phi(X) = \begin{bmatrix}
\Phi_1 \\
\Phi_2 \\
\Phi_3 \\
\Phi_4
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1
\end{bmatrix}, \quad \begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4
\end{bmatrix} = \begin{bmatrix}
x_1 \cdot x_3 \\
x_2 \cdot x_3 \\
x_1 \cdot x_4 \\
x_2 \cdot x_4
\end{bmatrix}
\]

By taking the four structure functions to represent four services, we can evaluate service failure rates for each of the services as follows.

\[
D = \begin{bmatrix}
\partial \Phi_1 \\
\partial \Phi_2 \\
\partial \Phi_3 \\
\partial \Phi_4
\end{bmatrix} = \begin{bmatrix}
x_3 & 0 & x_1 & 0 \\
0 & x_2 & x_3 & 0 \\
x_4 & 0 & 0 & x_1 \\
0 & x_4 & 0 & x_2
\end{bmatrix}
\]

For a system that considers the four paths as redundant paths and use statistically independent components, the structure function and its partial derivatives are given by:

\[
\Phi(X) = x_1 \cdot x_3 + x_1 \cdot x_4 + x_2 \cdot x_3 + x_2 \cdot x_4 - x_3 \cdot x_1 \cdot x_2 \cdot x_3 - x_3 \cdot x_1 \cdot x_2 \cdot x_4 - x_2 \cdot x_3 \cdot x_4 + x_1 \cdot x_2 \cdot x_3 \cdot x_4
\]

\[
\tfrac{\partial \Phi(X)}{\partial x_1} = (x_3 + x_4 - x_3 \cdot x_4) (1 - x_2)
\]

\[
\tfrac{\partial \Phi(X)}{\partial x_2} = (x_3 + x_4 - x_3 \cdot x_4) (1 - x_1)
\]

\[
\tfrac{\partial \Phi(X)}{\partial x_3} = (x_1 + x_2 - x_1 \cdot x_2) (1 - x_4)
\]

\[
\tfrac{\partial \Phi(X)}{\partial x_4} = (x_1 + x_2 - x_1 \cdot x_2) (1 - x_3)
\]

Hence, the system failure rate can be evaluated as follows.
8.7.1.1 System Failure Rate Estimation Using the Service-Module Covariance Matrix

Consider the term \( D(\mathbf{p}) \times [\mathbf{p} \cdot (1 - \mathbf{p})] \) which is a product of an mxn matrix by an nx1 vector in Equation 8-106. The entries of \( D(\mathbf{p}) \) corresponding to \( j^{th} \) service and \( i^{th} \) component takes the value \( \partial R_j / \partial p_i \) while the entries of the vector contains \( p_i (1 - p_i) \) corresponding to \( i^{th} \) component. The two entries can be combined into one matrix, \( C \), whose \((j,i)^{th}\) content is equal to \( p_i (1 - p_i) \partial R_j / \partial p_i \). This entry represents the covariance between the service and its component (Barlow and Proschan, 1996). Therefore, the matrix \( C \) can be called the covariance matrix. Note that this covariance matrix multiplied by a unit vector is equivalent to the derivative matrix \( D \) multiplied by the vector \( \mathbf{p}(1 - \mathbf{p}) \),

\[
D(\mathbf{p}) \times [\mathbf{p} \cdot (1 - \mathbf{p})] \equiv C \times \mathbf{1}
\]

Hence, the failure rate of services can be given as follows:

\[
\Lambda_s = (C \times \mathbf{1}) \bullet (H / \Phi(\mathbf{p}))
\]

where \( \mathbf{1} \) is a unit vector of size \( n \) 

Equation 8-107
8.7.2 Including the Effect of Duty Cycle

If necessary, the variations of failure rate contributions to service failure rate that comes due to differences in usage period may be captured by the concept of duty cycle (Bazovsky, 1961). The duty cycle is a measure of how long a component is in use (or needed) in a particular service and may be obtained from the following.

\[ d_i = \frac{\tau_{ci}}{\tau_s}, \text{ where} \]

\[ \tau_{ci} \text{ is the mean-component duty time per service cycle} \]

\[ \tau_s \text{ is the mean-service period per service cycle, or} \]

\[ d_i = \frac{\eta_i}{\gamma_i + \eta_i} \]

for \( \gamma_i \) is the component-operational rate

\( \eta_i \) is the component demand rate

Equation 8-108

The duty cycle is then used to as a multiplicative factor in estimating the effective component failure rate contribution to a service as in:

\[ \lambda_{ic} = d_i \lambda_c \]

Equation 8-109

When a component has both failure rate and probability of success per demand, \( R \), indices, the duty cycle applies to the combined failure rate from both indices. That is,

\[ \lambda_{ic} = d_i (\lambda_c + \eta(1 - R)) \]

where \( \eta \) is the demand rate

Equation 8-110

If a component is replicated a number of times and all are required to be operational at the same time, then, the duty cycle may be multiplied by the number of replicates.

8.7.3 Testability Analysis Using the Service-Module Derivative Matrix

The service-module derivative matrix, \( D \), that is used above for evaluation of failure rate has an additional advantage in identifying testability conditions. The entries of this matrix
provide the testing conditions of each service with respect to the correctness of every component. If an entry in D(s,c) is zero, it implies component c is irrelevant to service s. If the entry is 1, it means that service s is dependent only on c irrespective of the state of other components. In many cases, the entry may be some partial structure function involving other component states. This normally implies any correctness testing of service s with respect to component c requires the satisfaction of the partial structure function.

Example:

Consider three services \{S1, S2, S3\} composed of two modules \{M1, M2\} with the following structure functions.

\[
\Phi_1(x) = x_1 \\
\Phi_2(x) = x_1x_2 \\
\Phi_3(x) = x_1 + (1 - x_1)x_2
\]

(i.e. S1 depends only on M1, S2 is a series connection of M1 and M2 and S3 is a parallel connection of M1 and M2). The D matrix can be given as follows.

\[
D = \begin{bmatrix}
\frac{\partial \Phi_1}{\partial x_1} & \frac{\partial \Phi_1}{\partial x_2} \\
\frac{\partial \Phi_2}{\partial x_1} & \frac{\partial \Phi_2}{\partial x_2} \\
\frac{\partial \Phi_3}{\partial x_1} & \frac{\partial \Phi_3}{\partial x_2}
\end{bmatrix} = \begin{bmatrix}
x_1 & x_2 \\
1 & 0 \\
x_2 & x_1 \\
1 - x_2 & 1 - x_1
\end{bmatrix}
\]

From the above matrix, we can see that, S1 is entirely dependent on M1 but is independent of M2. The correctness of M1 is directly observable as the correctness of S1 in all conditions, since the entry at D(S1,M1) is 1. However, any failure of M2 cannot be observed in Service S1 in any condition.

The condition \(x_2\) at D(S2,M1) implies that correctness testing of service S2 with respect to M1 requires the operational state of M2 to be 1, i.e., M2 must function correctly for any failure of M1 to be detected at service S2.

On the other hand, the condition \(1-x_2\) at D(S3,M1) implies that the correctness of S3 with respect to that of M1 can only be observed if M2 is in failed state.
One question that may be asked about the practicality of the above structure functions is whether such mixture of functions, ranging from series to parallel connection of components, may exist at all for any given system. This is possible as can be seen from the following example. Consider a web-based system that supports two browsers. The reliability of the service provided to a user that has only one of the browsers may be given by the first structure function, while the reliability of the service provided by a user that uses both browsers as redundant components has the structure function given by three. For the system developer however, both browsers appear in series connection since any failure with any of the browsers is considered as system failure.

### 8.8 Chapter Summary

In this chapter, a novel approach of structural reliability prediction of software systems is introduced and thoroughly discussed. Reliability assessment is seen in three levels, component (primary) reliability, interaction reliability and service reliability.

In most software systems, developing correct modules is not a sufficient condition to get reliable systems. Various types of interaction failure causes may occur. These include race conditions, deadlocks, livelocks, resource depletions and congestions, mis-ordered events, incompatible input-outputs, and incomplete requirements. Based on basic probability and reliability theories, relevant reliability functions have been derived. Most of these functions depend on the service rates and demand rate of components whose estimations were given in the previous chapter.

The interaction reliabilities and component reliabilities basically affect the various services provided by a system, and the two are combined. The well known structure function is adapted to work with these reliability functions and relevant connection information, namely service-module dependency matrix. Unlike other structural reliability prediction models in the literature that view all software components to be in series connection, the model presented in here can combine arbitrary type of structure functions including parallel connections. System failure rate estimation formulas have also been derived. The technique discussed in here will be used in a later chapter.
9. Design for Reliability Predictability Applied on a Mobile Streaming System as a Case Study

9.1 Introduction

This chapter presents a case study in which the design and documentation approach that has been developed and discussed in the previous chapters of this thesis is applied. The case study is a subset of a mobile streaming system (MSS) consisting of a network of software, hardware, and wireless communication components in a distributed, scarce resource and stochastic environment with real-time characteristics.

The design of the system is presented in the following sections. For improved readability, the discussion will use informal diagrams and textual explanation since most of the reference documentations used to describe the system such as TFM descriptions, connection matrices, and event-flow-graphs would be too technical to communicate the essence of the system and the design approach followed. A partial set of the mathematical documentation materials are provided in Appendix A.

9.2 Introductory Description of MSS

MSS processes and streams large volume of data that can reach up to hundreds of megabyte per job, from an enterprise network, collectively called remote server (RS), over long range communication link (LRCL) such as GPRS (General Packet Radio Service), to a mobile devices (MD) and from the mobile device over a short range wireless communication link (SRCL) such as Bluetooth connection, to a local device (LD) in real time.

Figure 9-1 shows the network or connectivity structure of the MSS components. The deployment structure, where the software components reside can be stated as follows:

- Mobile side components (MSC): components deployed on MD
  - Two or more user applications (UA) issue request for MSS service,
  - A third application (PA) hosts MSS client, along with other elementary systems (OES), and provides display and user control services
There are over 6 deployment components which either make use of MSS or that have some parts used by MSS, and MSS client side composite alone consists of over 36 classes and interfaces.

- **Server side services (SSS):** components deployed on RS
  - All server side components including all MSS server components that require high computing resource: such as data storage and transformation, authentications, directory searching services, file uploading services.

- **Communication networks (CN):**
  - Long range communication link (LRCL): carrier wireless networks (GPRS),
  - Short range communication link (SRCL): Bluetooth radio link.

- **Local device (LD):**
  - Final consumer
  - Only pre-installed components

---

**Figure 9-1: MSS Network Structure**
9.3 Simplified View of Interaction with MSS

Figure 9-2: shows a simplified view of interaction with the mobile streaming system (MSS). MSS can be considered as part of a large set of mobile based applications or services. It interacts with other user applications, mainly a set of user applications which may be referred to as UA and one hosting parent application, referred to as PA. Its functions can be categorized into two: primary functions and secondary functions

- Primary functions:
  - Organize user input and prepare job order (PJO)
  - Process requested job on server (SSS)
  - Stream data from server to mobile and then to local device
  - Provide job control functions to user (Cont)

- Secondary functions:
  - Provide status update about each job on a display (SD)

![Figure 9-2: Simplified View of Interaction with MSS]
The user interacts with one of the user applications (UA), such as mail viewer or file explorer, to issue a streaming job. The user application passes the necessary information and the request to the parent application, which issues a job order to the remote server, and creates a job connector for managing the job. On this event, the Job module requests execution resources from the resource manager, RM, which in turn contacts the user for selecting the SRCL port to connect to. Depending on the selected port, either a new MSS client composite structure is created with all the necessary resource initialization or the job is placed to a pre-existing active structure. The streaming processes then continues which may take from a few minutes to hours per job. In the mean time, the user can issue other jobs or cancel existing ones, reorder, etc.

A formal specification of the requirements of the mobile service is provided in using the TFM method which is included in Appendix A.

9.4 System Constraints and Stochastic Properties in MSS
MSS has to function in a stochastic environment under various system constraints. The time sensitivity and unreliability of the wireless channels, the non-fault tolerant behavior of the peripheral device where missing of a single bit from tens of megabits causes unacceptable output, the limited computing and storage resource size of the mobile device make the system non-trivial to design, document and make reliability assessment. Some of the main properties that make the system non-trivial include:

1. The system connects a peripheral or local device (LD) with almost no-computing power to an enterprise network of servers with all the necessary computing power for the required services (collectively referred to as a remote server (RS)) via a mobile device (MD) with limited computing and storage power, giving the feature of a distributed system. All the three hardware components RS, MD, and LD can exist in many versions and variations.

2. The MD device has limited computing power. As a result, there are limits to the number of threads to be created in any given application, its limited memory size has to be shared among all services provided by the device, and access to its user interface display (UI) has to be shared among various components which have to acquire a lock to get access.
3 The connections among RS, MD and LD uses a long range communication link (LRCL) such as GPRS and a short range communication link (SRCL) such as Bluetooth, both of which having deadline issues as well as frequent disconnections, giving real-time stochastic characteristics. The connections may also be lost due to low battery conditions on device, mobility of the MD in relative to LD, poor network coverage, congestion with other device services. The LRCL has maximum limit to the size of data to be streamed stream at a time. This limit is 1MB with default setting being 128KB.

4 The LD device is non-fault tolerant, in that the large volume of data to be streamed to it has to be error free, where missing of a single bit from tens of megabytes can cause unacceptable output. Different LD devices require different drivers where incorrect mapping of drivers can result in an undesired output.

5 The LD device may occasionally expect intervention by users if it run out of supplies, during which time, it may not be able to consume data coming through SRCL. Some local devices also cancel a job if streaming of a job has been delayed by a couple of minutes.

6 The SRCL requires adaptor devices that may be purchased from third party vendors independently and connected to LD devices to enable connectivity. Different SRCL adaptors may have different features – in terms of maximum distance coverage, pass key requirements, etc. adding to the heterogeneity of the system.

7 The software components comprise a large set of modules that are designed and developed to provide the required service in addition to a number of commercial off-the-shelf components (COTS). The platforms, operating systems and programming language tools on the RS and MD are different forcing even sub modules that could be deployed on the two different nodes to be implemented in different programming languages, raising interoperability issues.
9.5 DRP Imperatives as Applied to MSS

As is the case with many software systems, by looking at the outer level (partial) specification of the requirements, the system seems deceivingly simple and straightforward. However, the various system constraints and stochastic environment described in section 9.4 made the system design, documentation and analysis of its behavior non-trivial.

The design of the system follows the DRP imperatives that were discussed in chapters 6 and 7. Some of the key features are highlighted below.

9.5.1 Design for Modular Redundancy

Modularization in DRP is done through the application of the information hiding principle. As a result each module is characterized by the design decision it hides from other modules making it a unit of change.

Modules are used to create or instantiate component instances which are connected to each other largely through abstract interfaces and attachable and detachable connectors. Mostly, container components specifically queues are used as input and output channels for the component instances since the nature of the application requires queuing for non-partitionable resources. Additionally, Synchronizer instances (also named as EventChannels) are attached to the component instances for monitoring and control purpose.

These connections are designed to allow runtime replacement of component instances, for restoration of interrupted processes. A composite structure is created by dynamically binding the necessary components to provide a required service. The connections in the composite structure may be changed at runtime for restoration purpose in response to occurrence of undesired events. As many composite structure as the number of ports available on the device can run in parallel allowing replicated redundancy. However, since much of the reliability issues result from timing issues, diversified redundancy has no much effect on reliability improvement even if supported for some of the modules.

Other design features of MSS include aggregation and hierarchy. The modular structure follows hierarchy based on the ‘is-secret-of’ relation. Aggregation of runtime state variables at a particular time defines the behavior of composites and the service provided
by them. Other hierarchical structures that may be found in MSS include interface hierarchy for restricting dependencies and process hierarchy for prioritized allocation of resources.

### 9.5.2 Ensure Non-interfering Behavior of Connections

The various components in the system are connected in a way that reduces or minimizes interferences among each others. This is done by separation of connectors as necessary and decomposition of various events into communicating processes. While a straightforward design would have lead to connecting all the components for a given task in a form of sequential process, structural reliability concerns, timing issues that have been revealed through TFM descriptions of some of the components, restorability requirements, lead this author to design an asynchronous concurrent system where the different component instances execute their own thread of control, synchronizing at connector components- the queues and synchronizers/event channels. This allows runtime extensibility and contractibility of the various components in the system.

One area this organization helped is in the management of undesired events, which is traditionally one common source of interactions and interference. In MSS design, undesired events are considered not as exceptional events that need to force transfer of control (like exception throwing-catching), but possible events that may occur on any component at any time. The various asynchronous processes that work in a given composite structure may share and communicate the occurrence of undesired events, including warning signals in case of non-responsiveness from some of the components. This allows for many of the processes in the composite structure to continue with their execution even in cases where one or more of the processes are unable to continue with a task waiting for an intervention from other components or the user. The communication provides all relevant components to get involved in the ‘maintenance’ of the failed part while the rest of the system continues to provide its service.

Another common area of interference and interaction is through resource sharing. Rather than allowing all components to depend on each other which may lead to interaction failure causes such as deadlocks, responsibilities are assigned to resource management
components. For instance, jobs may have to queue up in active composites when busy ports are selected or new composites are created if idle ports are chosen by users.

### 9.5.3 Improve Statistical Independence among Redundant Components and Services

Although the possibility of supporting diverse redundant modules is included in the design, only few modules exist in diverse implementation. Much of the reliability concerns in this particular system result from timing and resource management issues where diversified redundancy helps little. Yet, limited redundancy through input diversity and parallel running of composites is supported. Some of the services in the full application have overlapping functionalities that may lead some users to consider some of the services as redundant. Such redundancies cannot be considered statistically independent and the analysis must this into consideration.

### 9.5.4 Mathematical Description of Components and System

Various mathematical documents have been produced to precisely specify and describe the various components and the system behavior. These documents include the following:

#### 9.5.4.1 The Module Guide:

The decomposition of the system into modules is described by a module guide (MG) (Parnas et al., 1985, Parnas, 2009a), the only semi-formal document that combines a formal hierarchical description based on the ‘is-secret-of’ relation among modules, and a brief textual description of the hidden secret by each module. All MG documents of the MSS are summarized by one hierarchical tabular expression – which is a tree structure with depth 4 (corresponding to the different levels of decomposition) and number of leave nodes 61 (corresponding to the concrete modules, i.e., the lowest level of decompositions that hide the implementation details), for the subset of modules included in this case study. A summary of the module guide is given in Appendix Table A-1.

#### 9.5.4.2 Module Interface Specifications (TFM Specification of Interfaces)

The behavior of each of the interfaces of modules and components used in the system is specified using TFM. In addition to the various modules identified in MSS design, some of the relevant components that have been obtained as commercial off-the-shelf (COTS) but are part of the MSS system are also described precisely using TFM. This description
has been found instrumental in understanding the capabilities and limitations of these components, and in improving reliability by analyzing different design alternatives. The complete TFM specification of the interfaces used in this case study appear in a TFM document of 167 pages size, which is about 2.7 page per module since about 61 concrete modules have been identified in the module guide. Due to the large size of these TFM documents, only the descriptions of a few of components are given in appendix A.3, while the rest is not included in here for the sake of space.

9.5.4.3 Connection Matrices
A number of different composite structures can be created from the set of modules provided in the module guide. Figure A-1 in appendix 1 depicts the partial set of a composite structure instantiated or configured to provide the BP streaming service. The composite structure consists of the following components, inputs and outputs:

- Components C={LD, SRCL, LA, LW, LL, LB, RA, JO, MFS, EventChannel (EC), EventLogger(EL), Exec (EX), Job, Encrypt (ENC), Decrypt (DCR), Compress (COM), Decompress (DCM), JSS, RLM, UISD, UIJC, IMS, WFS, Checksum (CHK)}
- System Inputs: I ={j,c,r, rj}
- System Outputs: O={po, pc, sd, jr, Log, chk}

The input-output connections among the various runtime components are given in the matrix shown in Equation A-14-1. The entry of each cell, (i,j) represents the outputs of the i\textsuperscript{th} component provided as inputs to the j\textsuperscript{th} component. The symbols used in the matrix as well as the connection block diagram, in the matrix represent the events associated with the input/output variables. In most cases, they are abbreviated forms of the underlying events (for instance geb to mean get empty buffer, where as ebg is empty buffer got-basically the response of geb). The connection matrix can also be equivalently represented by a flow-graph, which is depicted in Figure A-2.

9.5.4.4 Event-Flow-Equations and Event-Flow-Graph
There are various design options that one can choose from, in structuring events or their partitioning across processes, irrespective of other structures namely: modules, connectors and resources. Among some of the design options that could be taken include:
Structuring all events for all tasks in one sequential process with a single thread of control

Structuring all events of each task in one sequential process but different tasks in different parallel processes

Structuring different subtasks of one task in communicating parallel processes

Scheduling each event as it occurs and allowing different events to occur in parallel only synchronizing when necessary.

From the various options, the first approach will make almost all components in a system to appear in series connection at least with respect to their failure mode that may terminate a thread of control. Therefore, if one aims for restoration, fault tolerance and redundancy, that option must be avoided.

The second option, is a straightforward design approach many designers even today may follow. It is usually chosen mainly for its direct mapping to most imperative languages and to avoid the risk of failure due to interaction of concurrent processes. However, as will be seen through reliability assessment given in the next chapter, it has a number of undesired properties that may make provision of some time-critical services infeasible.

The third and the fourth options are close to each other, where the fourth is a special case of the third one in which each process from the parallel processes consists only one event. Given that the distinction between events and processes can be dependent on the level of observation, in cases where the different subtasks are undertaken by different components and the actual processes or threads are internal, and the external communications appear in synchronized observable events, the two approaches may be equivalent for analysis.

Except probably for time delays, these approaches, especially the last one, may be considered close to how hardware devices work. When supported by synchronization and relevant input/output connectors, to avoid interaction failures, these design options are found to have many desirable properties, such as restorability, fault-tolerance, and redundancy, that affect the reliability structure of the system positively. Hence, these design options are used for structuring MSS events. However, whether some events are put in a sequential process or in parallel processes, certain ordering and causation
relations apply. These relationships are described using event-flow-equations which is derived using the connection matrix and the TFM description of the components used in the system. The equations are also represented in the equivalent event-flow-graph and both given in the appendix.

9.5.4.5 Implementation Code
The system is implemented using two object-oriented languages due to differing platform requirements on the mobile device and the remote server. This author would like to make a note that while MSS is a new system constructed based on the design approach discussed in this thesis by the author himself, it exists as part of a whole set of pre-existing applications as well as third software components. Table 9-1 shows the size of the code for MSS and related components excluding third part components.

Table 9-1: Source Code Statistics for MSS & its Environment

<table>
<thead>
<tr>
<th></th>
<th>Lines of code</th>
<th>Classes and interfaces</th>
<th>Files</th>
<th>Methods per class</th>
<th>Average Statements per Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>UA’s and PA’s on client</td>
<td>63,555</td>
<td>710</td>
<td>238</td>
<td>4.84</td>
<td>8.28</td>
</tr>
<tr>
<td>MSS Client Engine</td>
<td>6,876</td>
<td>36</td>
<td>31</td>
<td>16.89</td>
<td>4.29</td>
</tr>
<tr>
<td>SSS</td>
<td>45,927</td>
<td>220</td>
<td>171</td>
<td>7.21</td>
<td>11.12</td>
</tr>
</tbody>
</table>

9.6 Chapter Summary
In this chapter, a case study that is constructed following the principles of design for reliability predictability has been discussed. Samples of the various mathematical documents that have to be produced are given in the appendix. These include: precise specification of system requirements document as a TFM document, summary of module guide as hierarchical tabular representation, TFM interface specification documents for various components, connection matrix, event flow equations/graphs. These documents will be used in the next chapter for the analysis and prediction of system reliability.
10. Document Driven Structural Reliability Assessment of a Mobile Streaming System (Case Study)

10.1 Introduction
In this chapter, the mobile streaming system that was discussed in the previous chapter from design, structure and mathematical documentation aspects is used as a case study to illustrate the document driven structural reliability prediction technique proposed in this thesis. The TFM description of each module depicts the desired outputs of each module when operating with inputs in its domain as well as the undesired outputs that may happen when encountering undesired events. The composite structure created to provide the required service and the event-flow-graph that shows the flow of events in the connection structure to provide the required state transitions are used as inputs for obtaining the reliability functions of interaction failure causes. These interaction reliabilities and the primary component reliabilities are combined together based on connection matrices to estimate service reliability.

The case study demonstrates the importance of various structure, especially the event structure to system reliability. The fact that the system has to connect various heterogeneous components with stochastic behavior and constrained resource environment makes the reliability sensitive to the different ways of structuring the system. Different design alternatives are analyzed based on their predicted reliability and the design with the best reliability is implemented.

10.2 Analysis of Undesired Events
The event flow graph in Figure A-3 shows the possible occurrence of undesired events that can cause failure to the service being modeled. The graph can be used to analyze the possible occurrence of interaction failure causes and represent them as virtual components. Table 10-1 lists the interaction failure concerns that are visible in the event structure.
### Table 10-1: Interaction Failure Components:

<table>
<thead>
<tr>
<th>Possible Interaction Failure Cause</th>
<th>Virtual Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing Deadline</td>
<td>SRCL.TimeOUT: { \text{elt}(b_{n, a_{n-1}}) \text{&gt; SRCL.TimeOUT}}, \text{LRCL.TimeOUT}: { \text{elt}((g_{x \lor g_{p}})<em>{n}, d</em>{n-1}) \text{ &gt; LRCL.TimeOUT}} \text{RLM.MaxWait}: { \text{elt}(u_{n, j_{n-1}}) \text{&gt; RLM.MaxWait}}</td>
</tr>
<tr>
<td>Shared Resources</td>
<td>{LD, ldPort, LRCL, SRCL, Processors, Memory, UserScreen}</td>
</tr>
</tbody>
</table>

\{LD, ldPort\} – Non-partitionable resources. Hence, client components have to queue to use these resources. RM ensures that there is only one executor composite structure per each ldPort. There can be as many executor composites as the number of ldPorts. JO and LB are queues that ensure the resources are used through first-come-first-serve policy.

Memory- space partitionable resource. Each component can be provided with its own memory area whose lifetime covers from the moment a component is required to undertake a set of tasks, until the component completes all of its tasks. When there are no tasks to perform, RR releases all the memory resources used by the execution engine so that the system garbage collector collects them in its next cycle of operation.

The largest amount of memory required on the mobile device (MD) is that required for temporarily storing the data to be streamed from RS to LD. For unrestricted design where the producer process transferring data from RS to MD and the consumer process that transfers from MD to LD are allowed to work on their own speed, the entire memory capacity of the MD can be consumed in short period of time and may crash the application, since each stream from RS to MD can bring up to 1MB of data and the consumer processes is usually slower than the producer process.

To reduce the risk of memory depletion and fragmentation a reserve and recycle policy is employed through LB. For this, about 1 MB of memory is required. If the available memory is less than 1MB, the system will not be able to work. Hence, undesired event due to memory depletion occurs only if \( RD_m = \{M<1MB\} \)

Processors- time partitionable resources allocated to runtime units called processes. The system is mainly an asynchronous distributed system where various events from different sources (external hardware, users, wireless networks) may occur asynchronously and concurrently. Some of the events could be what is expected from users such as starting of new jobs, canceling or reordering of previous jobs. Other events may be undesired events such as loss of wireless connection, or non-responsiveness from connected devices. Different
components in the system may have to respond to these events to enable the system provide its required service. The events that result from execution of programs are partitioned into units of communicating processes in order to allow fault-tolerance and to reduce undesired events due to missing of deadlines.

LRCL- time and frequency partitionable communication resource that is shared by communication related services which may be initiated at an arbitrary time.

SRCL-communication resource that is partitionable by a finite set of separate ports. Multiple ports can be opened to communicate with different devices and the communication channels will be independent except for the possibility of cross interference that increases the unreliability of each channel.

UserScreen – the MD user screen for this particular system can show only one screen at a time and requires all user programs to queue for it (acquire an eventlock) in order to get access to it.

| Deadlock | No “resource hold and wait” allowed. Resources are managed either through queuing if they are not partitionable, or through time and space partitioning. Partial responsibilities of RM, JO, LB are for managing resources in a way that avoids hold and wait by individual user components. Hence, the conditions for occurrence of deadlock do not exist. |
| Race conditions | At MD: EventChannel and container components perform synchronization operations and protect components on the MD from race conditions.  
At RS: (remote server), we may have race condition if  
\[ RC_{sd,ec,t2} = \{ sd < ec < f2 \mid ec: 'LRCL.\text{UploadLimit}(t(ec)) < LRCL.\text{UploadLimit}'(t(ec))) \}, \text{ occurs, i.e., if the maximum upload limit is reduced in the middle of streaming process.} \] |
| Input-Output Incompatibilities | For all software components inside the composite structure, input-output compatibilities are ensured through interface declarations and syntax analysis. Some of the components, LD and SRCL adaptors, in the system are however wide range of commercial components that may require their own mappings. Ensuring the compatibility of these devices to the system is partially left for selection and configuration by users. For instance, users have to choose an appropriate device driver for each specific LD. Some universal device drivers that cover wide range of devices are included as default settings.  
One input-output incompatibility that has been observed in system development is related to passkeys for SRCL adaptors. Some commercial SRCL adaptors require passkeys as combination of alphanumeric characters while the MD-SRCL input mechanism
allows only numeric digits to be input as passkeys. This incompatibility excludes many adaptors from being possible members of the composite structure.

| Event Ordering | Required event orderings that may not always hold due to concurrency is \( \{s_d < f_2\} \). The condition \( EO_{sd_c,f_2n+1} = \{s_d > f_2\} \), may happen if sequence of events that start with \( u \) and arrive at \( sd_c \) and \( f_2 \) branching from event \( j_d \) are allowed to run in parallel. In this case, event \( f_2 \) may arrive to collect data \( sd_c \) that is not yet completely produced. Without a synchronizer component on the server side to synchronize the two events, race conditions may occur due to mis- ordered events, whose reliability depends on the relative speed of the two processes \( p_1: \{j_d \rightarrow sd_c\} \) and \( p_2: \{j_d \rightarrow f_2\} \).

An alternative design that would guarantee the occurrence of \( sd_c \) before \( f_2 \) is to avoid parallel execution of the two event sets, but rather start event \( u \) only after \( sd_c \) is completed. The risk with this design alternative is increased delay of event \( u \) that reduces the reliability with respect to missing deadline due to RLM.MaxWait.

A third alternative design can be to include a synchronization component on the server side which will of course add its own primary reliability concern to the system.

In most other cases, either the desired event-order is maintained through design, where some events that will always have to be performed in a specific sequence are put in a single thread and those that must be synchronized at certain points are synchronized through event channels. For others, the design is made to be invariant.

Various modes of operating the service can be considered for reliability prediction, the most important being estimating the reliability of successful completion of non-cancelled jobs. Estimating the probability of the system to successfully cancel a job when required can by itself be a reliability measure, but the undesired events that need to be analyzed in here- missing deadline, race conditions, etc. have no significance for cancelled jobs. Hence, it is possible to set \( c=0 \) (and \( \neg c=1 \)) for all transitions in the event flow graph. Additionally, the parts of the graphs that have no direct impact on the undesired events to be analyzed can be removed. A reduced event flow graph displaying the undesired events is shown in Figure 10-1. The analysis is made on the basis of a service cycle and the probability indices representing the reliability measure indicate the probability of success per each service cycle. This can be converted into service failure rate, and service reliability function using the formulas derived in chapter 8.
10.2.1 Deadline Risk Analysis:

There are three deadline failure causes in this system identified based on the component behavioral descriptions:

1. RLM.MaxWait: \( \text{elt}(u_n, j_{n-1}) > \text{RLM.MaxWait} \)

**Legend:**

- **E** Event
- **z** Conditioned trigger
- **Events communicated between system and user/environment**
2 SRCL.TimeOUT: \{ elt(b_n, a_{n-1}) > SRCL.TimeOUT \},

3 LRCL.TimeOUT: \{ elt((g x v g p)_n, d_{n-1}) > LRCL.TimeOUT \}

The first deadline, elt(u_n, j_{n-1}) > RLM.MaxWait, involves the time spent between event j and event u. There are a sequence of ‘four’ events shown in here.

Based on the discussion in 8.3.1.1, the processing time for the various events may be approximated using normal distribution. Since, the sum of normally distributed random variables is itself normally distributed; the distribution of the time spent to reach u from j is normally distributed with mean and variance obtained as the sum of the mean and variance of each of the four events. Hence, the deadline time can be given as:

\[
R_{d, ju}(t_{RLM.MaxWait}) = F_{ju}(t_{RLM.MaxWait}) = \frac{1}{\sigma_{ju}\sqrt{2\pi}} \int_{-\infty}^{t_{RLM.MaxWait}} e^{-\frac{(u-\tau_{ju})^2}{2\sigma_{ju}^2}} du
\]

where

\[
\tau_{ju} = \tau_j + \tau_{je} + \tau_{jd} + \tau_u
\]
\[
\sigma_{ju}^2 = \sigma_j^2 + \sigma_{je}^2 + \sigma_{jd}^2 + \sigma_u^2
\]

**Equation 10-1**

If the time distribution of \( \tau_{ju} \) follows exponential distribution, the deadline reliability can be given as:

\[
R_{d, ju}(t_{RLM.MaxWait}) = 1 - e^{-t_{RLM.MaxWait}/\tau_{ju}}
\]

**Equation 10-2**

The second deadline, elt(b_n, a_{n-1}) > SRCL.TimeOUT, involves the time traversed from a to b. The relevant event flow graph is shown in EFG.

![EFG in relation to SRCL-Timeout](image-url)
There are two time paths: one that directly connects a to b – with probability $p_{ld}$, and another that goes through peb, gfb and fbg with probability $q_{ld}$. The direct connection from a to b does not take time (time elapses at the nodes not on the edges unless there is conditional dependency on other nodes) and hence it can be ignored from being a risk factor for missing deadline. Thus, the contributor for missing deadline is the time path peb, gfb and fbg. The events in this path require transition from gfb to fbg that is conditional on LB not being empty with respect to fb. If LB is empty, the transition has to wait until an ‘fbp’ event arrives. Hence, the waiting time at gfb depends on the time distribution of the producer event, i.e. fbp, as well as the size of the queue, LB. By approximating the various processing time distributions with normal distribution, one can obtain the time distribution between events a and b as follows.

Evaluating elapsed time in flow-graphs requires addition operation while traversing the graph while the basic signal flow-graph evaluation rules are obtained through multiplication. However, different transforms exist that convert the addition to multiplication, one among them being the moment generating functions (MGF) (Jenab et al., 2008). The MGF for the time distribution between event a and b, is a product of the probability $q_{ld}$ and the MGF’s of the time distribution for the three events.

$$MGF_{ab}(s) = p_{ld} + q_{ld}e^{\mu_{peb}s + \frac{\sigma_{peb}^2s^2}{2}}e^{\mu_{gfb}s + \frac{\sigma_{gfb}^2s^2}{2}}e^{\mu_{fbg}s + \frac{\sigma_{fbg}^2s^2}{2}}$$

$$= p_{ld} + q_{ld}e^{(\mu_{peb} + \mu_{gfb} + \mu_{fbg})s + (\sigma_{peb}^2 + \sigma_{gfb}^2 + \sigma_{fbg}^2)s^2}$$

Equation 10-3

This MGF represents the sum of a constant and a defective normal distribution function, and hence the reliability with respect to not missing the SRCL.TimeOut deadline is given as follows.
\[ R_{d,ab}(t_{SRCL.TimeOut}) = F_{ab}(t_{SRCL.TimeOut}) = p_{ld} + \frac{q_{ld}}{\sigma_{ab} \sqrt{2\pi}} \int_{-\infty}^{t_{SRCL.TimeOut}} e^{-\frac{(u-\tau_{ab})^2}{2\sigma_{ab}^2}} du \]

where

\[ \tau_{ab} = \tau_{peb} + \tau_{gfb} + \tau_{fbg} \]

\[ \sigma_{ab}^2 = \sigma_{peb}^2 + \sigma_{gfb}^2 + \sigma_{fbg}^2 \]

**Equation 10-4**

Alternatively, if the cumulative waiting time \( \tau_{ab} = 1/\gamma_{ab} \) is approximated by exponential distribution, the deadline reliability can be given as follows.

\[ MGF_{ab}(s) = p_{ld} + q_{ld} \frac{\gamma_{ab}}{\gamma_{ab} + s} \]

\[ R_{d,ab}(t_{SRCL.TimeOut}) = p_{ld} + q_{ld} \left( 1 - e^{-\gamma_{ab} t_{SRCL.TimeOut}} \right) = 1 - q_{ld} e^{-\gamma_{ab} t_{SRCL.TimeOut}} \]

**Equation 10-5**

The third deadline, \{elt((gx\lor gp)n, dn-1) > LRCL.TimeOUT\} involves the time traversed from d to gx or gp. Figure 10-3 shows the event flow graph of those events directly related to the undesired event.

**Figure 10-3:** EFG in relation to LRCL TimeOut

In the above event flow graph, the transition from pfb to fbp is conditional to LB not being full with respect to its fb store. If it is full, the transition will have to wait for the fbg event to occur.
From the time analysis point of view, this event flow graph is equivalent to the one given in Figure 8-4. Therefore, we can use the equations obtained there, either Equation 8-32, normal distribution approximation or Equation 8-33- saddle point approximation to estimate the time distribution function.

Taking Equation 8-32, which is re-written below again,

\[
F_{lo}(t) \approx \frac{p_r}{\sqrt{2\pi(\sigma_r^2 p_r + (1 - p_r)\tau_r^2)}} \int_{-\infty}^{t} e^{-\frac{(x - \tau_r)^2}{2(\sigma_r^2 p_r + (1 - p_r)\tau_r^2)}} dx,
\]

The appropriate substitutions is made to adapt it to the event flow graph shown above. i.e.,

\[
\tau_{dg} = \tau_{geb} + \tau_{obj} + \tau_{ppb} + \tau_{fdp}
\]
\[p_r = q_{lb}, \quad \bar{p}_r = p_{lb}
\]  
\[t = t_{LRCL.TimeOU}
\]

\[
R_{d, dg}(t_{LRCL.TimeOU}) = F_{d, dg}(t_{LRCL.TimeOUT})
\]

\[
R_{d, dg}(t_{LRCL.TimeOUT}) \approx \frac{q_{lb}}{\sqrt{2\pi(\sigma_{dg}^2 q_{lb} + p_{lb}\tau_{dg}^2)}} \int_{-\infty}^{t_{LRCL.TimeOUT}} e^{-\frac{(x - \tau_{dg})^2}{2(\sigma_{dg}^2 q_{lb} + p_{lb}\tau_{dg}^2)}} dx,
\]

**Equation 10-6**

For exponentially distributed times, one gets

\[
R_{d, dg}(t_{LRCL.TimeOUT}) = 1 - e^{-q_{lb}t_{LRCL.TimeOUT} / \tau_{dg}}
\]

**Equation 10-7**

### 10.2.1.1 Effect of Process Structure on Reliability with Respect to not Missing Deadline

Reliabilities with respect to not missing deadlines are highly dependent on the process structure i.e. whether events for execution of a given job are structured in one sequential process or they are partitioned and run in a number of concurrent/parallel processes. This can be shown as follows:
### 10.2.1.2 Reliability with Respect to not Missing Deadline in a Sequential Process

A sequential process is a process in which all events (for execution of a particular job) in the set are ordered by design.

If all the events for the streaming process were to be structured in one long sequential process, which is still a common design practice in software design for its simplicity in avoiding race conditions and deadlocks, and among the first design alternative considered, then the various events become interdependent on each other and the failure rate of time sensitive components will increase significantly.

Two major changes in the above event structure influence the deadline reliabilities. The first one is event u will no more follow event jd but has to wait until sd is completed. This will reduce $R_{d,ju}$ if the same deadline time is used. The second change is events fdp and fbg will merge and become one event, where producer process that streams data from RS to MD has to wait until all the consumer process to transfer from MD to LD, and the consumer has to wait until the producer produces the data. This increases the waiting times $\tau_{fdp}$ and $\tau_{fbg}$, decreasing $R_{d,dg}$, $R_{d,ab}$ respectively. The relevant event flow graph that shows the dependency between fdp and fbg is given in Figure 10-4.

The changes in deadline reliabilities are discussed below.

#### 10.2.1.2.1 Deadline for RLM.MaxWait

For RLM.MaxWait: \(\{elt(u_{i,n}, j_{n-1}) > RLM.MaxWait\}\), if the process \(p1: \{jd\rightarrow sd\}\) has \((\mu_1, \sigma_1)\) mean-execution time and standard deviation respectively, then mean and standard deviation of the waiting time distribution is changed as follows.

\[
R_{d,ju}(t_{RLM.MaxWait}) = F_{ju}(t_{RLM.MaxWait}) = \frac{1}{\sigma_{ju} \sqrt{2\pi}} \int_{-\infty}^{t_{RLM.MaxWait}} e^{-\frac{(u-\tau_{ju})^2}{2\sigma_{ju}^2}} du
\]

where

\[
\tau_{ju} = \tau_j + \tau_{je} + \tau_{jd} + \tau_u + \mu_j
\]

\[
\sigma_{ju}^2 = \sigma_j^2 + \sigma_{je}^2 + \sigma_{jd}^2 + \sigma_u^2 + \sigma_l^2
\]

**Equation 10-8**

For exponentially distributed waiting time, the deadline reliability can be given as:
\[ R_{d,ju}(t_{RLM,MaxWait}) = 1 - e^{-\gamma_{RLM,MaxWait}t} \]

Equation 10-9

10.2.1.2.2 Deadline for SRCL.TimeOut

For SRCL.TimeOUT: \{ elt(b_n, a_{n-1}) > SRCL.TimeOUT \}, consider the relevant event flow graph in Figure 10-4.

In sequential process mode, the mean waiting time at the event fbg, \( \tau_{fbg} \), is the mean recurrent time of the fbp event, i.e., the time taken starting from fdp until it comes back to the same event (this is the time where either a previously opened stream is read or new stream is fetched from RS to MD). The events that could follow fbp are geb, gp or gx. Out of these three possibilities, the transition to gx occurs when a new job is to be retrieved after completion of the job that was being processed and hence missing of deadline has no negative effect on the system. Thus, the waiting time that affects system reliability is determined by the transitions from fdp to geb or fdp to gp. Event b follows after return of the producer process to fdp. Since the interest in here is to find the time elapsed between event a and b on the condition that fdp and fbg lumped as one event, the event flow graph need to be redrawn as given below. The events pg, gp, f1, fe, f2, dc,
and d appear in sequence (although each of them could involve many parallel processes at lower level of analysis) and can thus be lumped together as one event (LRCL), whose mean and standard deviation of the time is obtained as a sum of the mean and standard deviation of each of the events. Similarly, events peb and geb can be considered as one event and so can events ebg, pf b, and fbp since all appear as sequential events. Hence, the event flow graph for the relevant time can be simplified and re-drawn as shown in Figure 10-5:

Figure 10-5: Simplified EFG for the SRCL timeout event (MSS sequential process)

Approximating the time distributions by normal distribution, the moment generating function for the time between fbp_s to fbp_e can be obtained as:

\[ MGF_{ab}(s) = p_{id} + q_{ld}e^{\tau_{ab} + \sigma_{ab}^2 s^2 / 2} \left( q_{lb} + \left( p_{lb} + p_f q_{lb}e^{\tau_{LRCL} + \sigma_{LRCL}^2 s^2 / 2} \right) e^{\tau_{lb} + \sigma_{lb}^2 s^2 / 2} \right) \]

\[ = p_{id} + q_{ld} \cdot p_{lb} e^{(\tau_{ab} + \tau_{lb})s + (\sigma_{ab}^2 + \sigma_{lb}^2)s^2 / 2} + q_{lb} q_{ld} e^{\tau_{lb} + \sigma_{lb}^2 s^2 / 2} + q_{ld} p_f q_{lb} e^{(\tau_{LRCL} + \tau_{lb} + \tau_{ab})s + (\sigma_{LRCL}^2 + \sigma_{lb}^2 + \sigma_{ab}^2)s^2 / 2} \]

This MGF is the sum of a constant and three (defective) normal distribution functions. Thus, the deadline reliability from the cumulative distribution function represented by the above MGF can be obtained as follows:
In the sequential process configuration, the major event that takes much processing time is the event related to fetching of one split file from RS to MD, i.e. the LRCL event. Compared to this event, the time taken by other events peb,gfb,fbg, geb,ebg and pfb are insignificant. These events may have significance in the concurrent process setting where the producer and consumer processes operate on their own speed independently but synchronize on these time points. One can use these facts to simplify the above equation, and obtain the following.

\[
R_{d,ab}(t_{SRCL\text{TimeOut}}) = F_{ab}(t_{SRCL\text{TimeOut}}) = p_{ld} + \frac{q_{ld}p_{lb}}{\int_{-\infty}^{t_{SRCL\text{TimeOut}}}} e^{-\frac{(u-(\tau_{lab}+\tau_{eab}))^2}{2(\sigma_{\text{lb}}^2 + \sigma_{\text{eb}}^2)}} du + \frac{q_{f} q_{lb} q_{ld}}{\sqrt{2\pi(\sigma_{\text{lb}}^2 + \sigma_{\text{eb}}^2)}} \int_{-\infty}^{t_{SRCL\text{TimeOut}}+\tau_{lab}} e^{-\frac{(u-\tau_{lab})^2}{2\sigma_{\text{lb}}^2}} du \cdot p_{f} \cdot q_{lb} \cdot q_{ld} + \frac{q_{ld} \cdot p_{f} \cdot q_{lb}}{\sigma_{LRCL} \cdot \sqrt{2\pi}} \cdot \int_{-\infty}^{t_{SRCL\text{TimeOut}}} e^{-\frac{(u-(\tau_{LRCL}+\tau_{eb}))^2}{2(\sigma_{LRCL}^2 + \sigma_{\text{lb}}^2 + \sigma_{\text{eb}}^2)}} du
\]

Equation 10-10

In the above equation, \(\tau_{LRCL}\) is the mean production time of data size of one split file, i.e., the time to transmit one stream of data from RS to MD. \(\sigma_{LRCL}\) is the corresponding standard deviation. To be able to work with different file size, the parameters need to be converted into per unit parameters. Let \(\tau_{pb}\) be the mean time (in seconds) to produce a unit (e.g. byte or kilobyte) of data, and \(\sigma_{pb}\) the corresponding standard deviation. With the assumption of linear scaling of streaming time with data size, one can write, \(\tau_{LRCL} = \tau_{pb} \cdot \text{size(SplitFile)}\). Similarly, the standard deviation can be given as:

\(\sigma_{LRCL}^2 = \sigma_{pb}^2 \cdot \text{size(SplitFile)}\).

This, gives the deadline reliability with respect to SRCL.TimeOut as:

\[
R_{d,ab}(t_{SRCL\text{TimeOut}}) = p_{ld} + \frac{q_{ld}p_{lb}}{\int_{-\infty}^{t_{SRCL\text{TimeOut}}}} e^{-\frac{(u-(\tau_{lab}+\tau_{eab}))^2}{2(\sigma_{\text{lb}}^2 + \sigma_{\text{eb}}^2)}} du + \frac{q_{f} q_{lb} q_{ld}}{\sqrt{2\pi(\sigma_{\text{lb}}^2 + \sigma_{\text{eb}}^2)}} \int_{-\infty}^{t_{SRCL\text{TimeOut}}+\tau_{lab}} e^{-\frac{(u-\tau_{lab})^2}{2\sigma_{\text{lb}}^2}} du \cdot p_{f} \cdot q_{lb} \cdot q_{ld} + \frac{q_{ld} \cdot p_{f} \cdot q_{lb}}{\sigma_{LRCL} \cdot \sqrt{2\pi}} \cdot \int_{-\infty}^{t_{SRCL\text{TimeOut}}} e^{-\frac{(u-(\tau_{LRCL}+\tau_{eb}))^2}{2(\sigma_{LRCL}^2 + \sigma_{\text{lb}}^2 + \sigma_{\text{eb}}^2)}} du
\]
\[ R_{d,ab}(t_{\text{SRCL TimeOut}}) = p_{lb} + q_{lb} p_{fb} + q_f q_{lb} q_{ld} + \]
\[
\frac{q_{lb} p_f q_{ld}}{\sigma_{pb} \sqrt{2\pi \times \text{size(SplitFile)}}} \int_{-\infty}^{t_{\text{SRCL TimeOut}}} e^{-\frac{(u - \tau_{pb} \times \text{size(SplitFile)})^2}{2\sigma_{pb}^2 \times \text{size(SplitFile)}}} du
\]

\[
= 1 - q_{ld} + q_{ld} - q_{ld} q_{lb} + q_f q_{lb} q_{ld} + \]
\[
\frac{q_{ld} p_f q_{lb}}{\sigma_{pb} \sqrt{2\pi \times \text{size(SplitFile)}}} \int_{-\infty}^{t_{\text{SRCL TimeOut}}} e^{-\frac{(u - \tau_{pb} \times \text{size(SplitFile)})^2}{2\sigma_{pb}^2 \times \text{size(SplitFile)}}} du
\]

\[
= 1 - q_{ld} p_f q_{lb} \left(1 - \frac{1}{\sigma_{pb} \sqrt{2\pi \times \text{size(SplitFile)}}} \int_{-\infty}^{t_{\text{SRCL TimeOut}}} e^{-\frac{(u - \tau_{pb} \times \text{size(SplitFile)})^2}{2\sigma_{pb}^2 \times \text{size(SplitFile)}}} du \right)
\]

Equation 10-12

If the time distribution for the LRCL network follows exponential distribution instead of the normal distribution, the deadline reliability would look like as follows.

\[ R_{d,ab}(t_{\text{SRCL TimeOut}}) = 1 - q_{ld} p_f q_{lb} e^{-t_{\text{SRCL TimeOut}} \times \text{Size(SplitFile)}} \]

Equation 10-13

Observe that for small streaming jobs that can be streamed in one cycle, where \( q_f = 1 \) and \( p_f = 0 \), the above deadline reliability is 1.

1.1.1.1.1 Deadline for SRCL.TimeOut

The waiting time for LRCL.TimeOUT: \{elt((gx\lor gp)n, dn-1) > LRCL.TimeOUT\} is the time between d and gp or gx as before, but with sequential process, it includes the time of the consumer – i.e. the recurrent time for the fbg-fbp combined event. In the sequential process configuration, the next event following fbp-fbg depends on the preceding event. If the entry to fbp event is from pfb which contains data to be streamed, the next event has to be event b. If the entry is from gfb, then the next event could be geb, if there is more data to read or either gp or gx depending on whether the streaming process for a
given job is completed or not. The modified event-flow graph is shown below in Figure 10-6:

![Event Flow Graph](image)

**Figure 10-6: EFG for the LRCL Timeout event (MSS sequential process)**

The reliability with respect to not missing the LRCL’s deadline in one stream cycle can be obtained by finding the cumulative distribution for the time spent by the system in traversing from event d to gp/gx. Despite the presence of a number of events in the above event flow graph, many of them take insignificant time especially in the sequential process setting since the local buffer is empty and the consumer has been waiting to consume data. The major time taking events are those from b to p and back to a, which represent the processing time of the streamed data by the consumer. The event-flow graph can further be simplified as given below:

![Simplified Event Flow Graph](image)

**Figure 10-7: Simplified EFG for the LRCL Timeout event (MSS sequential process)**

One can consider two distribution functions for the elapsed time in consuming a stream of data of size ld.buffer: normal distribution with mean $\tau_{ld}$ and standard deviation $\sigma_{ld}$ and exponential distribution with rate $\lambda_{ld} = 1/\tau_{ld}$. 

---

276
The reduction of feedback loops in evaluating transfer function for both distributions was discussed in 8.3.1.1.2.1. For the exponential distribution, the transfer function of the feedback loop was itself exponentially distributed. Thus, applying successive loop reduction in the above flow graph, one can obtain the time distribution between event d and gp/gx as exponentially distributed with rate $\lambda_{dg} = \lambda_{ld}q_{lb} = q_{ld}q_{lb}/\tau_{ld}$. Hence, the deadline reliability of LRCL, for an exponentially distributed consumption process, can be given as:

$$R_{d,dg}(t_{LRCL.TimeO}) = F_{d,dg}(t_{LRCL.TimeO}) = 1 - e^{-\lambda_{ld}t_{LRCL.TimeO}^R} = 1 - e^{-q_{ld}q_{lb}t_{LRCL.TimeO}/\tau_{ld}}$$

Equation 10-14

In the above, $\tau_{ld}$ is the mean data consumption time of size ld.buffer. For a system that takes mean time of $\tau_b$ seconds to consume a byte of data and whose consumption time linearly scales with size of data, one can write, $\tau_{ld} = \tau_{cb}.size(ld.buffer)$. Similarly, for standard deviation $\sigma_{ld}^2 = \sigma_{cb}^2.size(ld.buffer)$

Additionally we have,

$$q_{lb} = \begin{cases} size(lb.buffer)/size(splitFile), & \text{for } size(lb.buffer) < size(splitFile) \\ 1, & \text{otherwise} \end{cases}$$

$$q_{ld} = \begin{cases} size(ld.buffer)/size(lb.buffer), & \text{for } size(ld.buffer) < size(lb.buffer) \\ 1, & \text{otherwise} \end{cases}$$

$$q_{lj} = \begin{cases} size(SplitFile)/size(Job.Doc), & \text{for } size(SplitFile) < size(Job.Doc) \\ 1, & \text{otherwise} \end{cases}$$

In the system under discussion, the q-values are less than 1 giving:

$q_{ld}.q_{lb} = size(ld.buffer)/size(SplitFile)$. Substituting these and the values for $q_{ld}.q_{lb}$, in Equation 21-6, one gets:
\[ R_{d, dg} (t_{LRCL.TimeOUT}) = 1 - e^{-f_{LRCL.TimeOUT} / \tau_{cb.size(SplitFile)}} \]

**Equation 10-15**

For normal distribution, the mean and standard deviation may be obtained using Equation 8-31:

The inner loop parameters are,

\[ \tau_1 = \frac{\tau_{ld}}{q_{ld}} \]
\[ \sigma_1^2 = \frac{\sigma_{ld}^2 + \tau_{ld}^2 p_{ld}}{q_{ld}} \]

While the outer loop and hence the time \( d \) to \( gp/gx \) are given by:

\[ \tau_{dg} = \frac{\tau_1}{q_{lb}} = \frac{\tau_{ld}}{q_{ld} q_{lb}} = \frac{\tau_{cb.size(ld.buffer)}}{size(ld.buffer) / size(SplitFile)} = \tau_{cb.size(SplitFile)} \]
\[ \sigma_{dg}^2 = \frac{\sigma_{ld}^2 + \tau_{ld}^2 p_{ld}}{q_{lb}^2} + \frac{\tau_{ld}^2 p_{ld}}{q_{lb} q_{ld}^2} + \frac{\tau_{ld}^2 p_{lb}}{q_{lb}^2 q_{ld}^2} = \frac{\sigma_{ld}^2 + \tau_{ld}^2 p_{ld}}{q_{lb}^2} + \frac{\tau_{ld}^2 p_{lb}}{q_{lb}^2 q_{ld}^2} \left( \frac{p_{ld}}{q_{ld}} + \frac{p_{lb}}{q_{lb}} \right) \]
\[ = \frac{\sigma_{cb.size(SplitFile)}^2}{size(ld.buffer) / size(SplitFile)} + \tau_{cb.size(SplitFile)}.\tau_{cb.size(ld.buffer)} \left( \frac{p_{ld}}{q_{ld}} + \frac{p_{lb}}{q_{lb}} \right) \]
\[ = \sigma_{cb.size(SplitFile)}^2 + \tau_{cb.size(SplitFile)}.size(ld.buffer).\frac{size(lb.buffer)}{size(ld.buffer)} \left( \frac{p_{ld}}{q_{ld}} + \frac{p_{lb}}{q_{lb}} \right) \]
\[ = \sigma_{cb.size(SplitFile)}^2 + \tau_{cb.size(SplitFile)}.size(lb.buffer)(1 - \frac{size(lb.buffer)}{size(ld.buffer)}) + \frac{(size(SplitFile) - size(lb.buffer))}{size(lb.buffer)} \]
\[ = \sigma_{cb.size(SplitFile)}^2 + \tau_{cb.size(SplitFile)}.size(lb.buffer) \left( \frac{size(SplitFile)}{size(lb.buffer)} - \frac{size(lb.buffer)}{size(ld.buffer)} \right) \]

If the two ratios in the above have near equal values, then the variance of the resulting time distribution will be the same as the unit variance scaled by the size of data to be streamed. Using this variance to approximate with normal distribution function, the deadline reliability can be estimated as:
The above three deadline probabilities can now be combined together to estimate the probability of successful completion of a given streaming job. The reliability structure (for the deadline reliability only) can be obtained from the event flow graph by identifying the possible occurrence time points and their recurrence conditions.

The starting event for a streaming job is j. The first deadline failure may occur once in the beginning of each streaming job, i.e. before reaching event d. However, the second deadline failure can possibly occur within each streaming cycle, between events d and fdp, until the completion of a given job. The third deadline may also occur at a more frequent rate, i.e. on each occurrence of event a. The probability of completing a job in one streaming cycle is given by:

\[ q_f = \frac{\text{size(SplitFile)}}{\text{size(Job.Doc)}} \]

The flow graph for evaluation of the reliability is given in Figure 10-8.

**Equation 10-16**

Equation 10-6 is the same as what would have been obtained based on the loading factor discussion in 8.3.1.1.1.1.
The cumulative deadline reliability for streaming a job can be obtained as a transfer function from \( j \) to \( gx \). To find this, the well-known flow-graph evaluation formulas (Burroughs and Happ, 1962) can be applied, where the loops are successively replaced by single links. The result is given below.

\[
R_d = \frac{1}{1 - p_{ld} R_{d,ab}} \frac{q_{ld} R_{d,lg}}{1 - p_{lb} R_{d,ab}} \frac{q_{lb} R_{d,g}}{1 - p_{jd} R_{d,ab}} R_{d,lg} R_{d,g} q_f = R_{d,ju} q_f \frac{q_{ld} q_{lb} R_{d,lg}}{1 - p_{ld} R_{d,ab} - p_{lb}} \frac{q_{jd} q_{ld} R_{d,lg}}{1 - p_{jd} R_{d,ab} - p_{lb}}
\]

\[
R_d = \frac{q_{ld} q_{lb} q_f R_{d,ju} R_{d,lg}}{1 - p_{ld} R_{d,ab} - p_{lb} q_{ld} - p_{jd} q_{ld} R_{d,lg} R_{d,g}}.
\]

Equation 10-17

10.2.1.2.2.1 Example with Numerical Data:

The following are sample data that are related to the system under study. The main component for LRCL is a GPRS network, which has a standard transmission rate of 115 kilo bits per second. The performance of the streaming from MD to LD is measured based on test data. The various parameters are given in Table 10-2.
Table 10-2: Parameter Values in MSS

<table>
<thead>
<tr>
<th>Component</th>
<th>Parameter</th>
<th>Value</th>
<th>Source of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRCL</td>
<td>( \gamma_{pb} )</td>
<td>14.375 KBps</td>
<td>GPRS standard speed</td>
</tr>
<tr>
<td></td>
<td>( \tau_{pb} = 1/\gamma_{pb} )</td>
<td>0.06956 s per KB</td>
<td>Calculated</td>
</tr>
<tr>
<td></td>
<td>( t_{LRCL\cdot TimeOut} )</td>
<td>120s</td>
<td>Assumption (based on observation)</td>
</tr>
<tr>
<td>SRCL</td>
<td>( \gamma_{cb} )</td>
<td>2.84 KBps</td>
<td>Performance testing on a sample</td>
</tr>
<tr>
<td></td>
<td>( \tau_{cb} = 1/\gamma_{cb} )</td>
<td>0.3521 s per KB</td>
<td>Calculated</td>
</tr>
<tr>
<td></td>
<td>( t_{SRCL\cdot TimeOut} )</td>
<td>120s</td>
<td>Assumption (based on observation)</td>
</tr>
<tr>
<td>RLM</td>
<td>( \tau_{ju} )</td>
<td>1 min</td>
<td>Performance testing on sample data</td>
</tr>
<tr>
<td></td>
<td>( \gamma_{ju} = 1/\tau_{ju} )</td>
<td>1 pm</td>
<td>Calculated</td>
</tr>
<tr>
<td></td>
<td>( t_{RLM\cdot MaxWait} )</td>
<td>20min</td>
<td>Design parameter (setting)</td>
</tr>
<tr>
<td>Job</td>
<td>Size(Job.Doc)</td>
<td>20MB</td>
<td>Average job size</td>
</tr>
<tr>
<td></td>
<td>Size(SplitFile)</td>
<td>1MB=1024KB</td>
<td>Maximum allowable upload</td>
</tr>
<tr>
<td></td>
<td>( q_f )</td>
<td>1/20=0.05</td>
<td>Calculated</td>
</tr>
<tr>
<td></td>
<td>( p_f )</td>
<td>0.95</td>
<td>Calculated</td>
</tr>
</tbody>
</table>

The deadline reliability based on the exponential distribution is given in Table 10-3:

Table 10-3: Deadline Reliability

<table>
<thead>
<tr>
<th>Reliability Parameter</th>
<th>Relevant Equation</th>
<th>Estimated value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_{d,ju} )</td>
<td>( R_{d, ju}(t_{RLM\cdot MaxWait}) = 1 - e^{-\gamma_{ju}t_{RLM\cdot MaxWait}} )</td>
<td>0.999999998</td>
</tr>
<tr>
<td>( R_{d,dg} )</td>
<td>( R_{d,dg}(t_{LRCL\cdot TimeOUT}) = 1 - e^{-t_{LRCL\cdot TimeOUT} / (\tau_{cb}\cdot size(SplitFile))} )</td>
<td>0.283104</td>
</tr>
<tr>
<td>( R_{d,ab} )</td>
<td>( R_{d,ab}(t_{SRCL\cdot TimeOut}) = 1 - q_{ld} \cdot p_f \cdot q_{lb} e^{-t_{SRCL\cdot TimeOut} / (\tau_{pb}\cdot Size(SplitFile))} )</td>
<td>0.197291</td>
</tr>
<tr>
<td>( R_d )</td>
<td>( R_d = \frac{q_{ld}q_{lb}q_f}{1 - p_{ld}R_{d,ab} - p_{lb}q_{ld} - p_fq_{ld}q_{lb}R_{d,dg}} )</td>
<td>0.019363</td>
</tr>
</tbody>
</table>

The values for the reliability parameters are as follows:

- \( q_{ld} = q_{lb} = 1, q_f = 0.05 \) for \( R_{d,ju} \)
- \( q_{ld} = q_{lb} = 1, q_f = 1, \) for \( R_{d,dg} \) when \( \text{SplitFile} = 1024\KB \)
- \( q_{ld} = q_{lb} = 1, q_f = 1, \) for \( R_{d,ab} \) when \( \text{SplitFile} = 128\KB \)
As can be seen from the above reliability estimate (with respect to not missing deadline alone), the reliability of the system in sequential process configuration is extremely poor for streaming data that exceeds 1MB. This estimate is obtained even by ignoring delays in the various events such as updating job status on user screen which occurs frequently almost with every event and requires waiting for acquiring a lock for a user screen that could be shared by other applications in the system.

The system may only work for small jobs but for the required task, it is not acceptable at all. Hence, this design option is rejected and the concurrent process option discussed below is implemented.

10.2.1.3 Reliability with Respect to not Missing Deadline in Concurrent/Parallel Processes with Arbitrary Speeds

Concurrent/parallel processes are a set of processes where events from different processes can arbitrarily be ordered except on certain synchronization or waiting time points.

The MSS event-flow graphs shown in Figure A-3 (Appendix-A), represents events from a set of concurrent processes that can occur in parallel, while communicating at certain synchronization time points. The events can be partitioned into a set of concurrent processes are shown in Table 10-4:

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Process ID</th>
<th>Description</th>
<th>Members of the Event Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Input Process</td>
<td>UIP</td>
<td>Set of events related to providing input to the system by the user.</td>
<td>{j,rj,r,c}</td>
</tr>
<tr>
<td>Construction &amp; Initialization Processes</td>
<td>CIP</td>
<td>Events related to the construction and initialization of instance components for the required job.</td>
<td>{n,nla, la, ne, nlb, njo, ra, lb, jo, ra, e, i}</td>
</tr>
<tr>
<td>Remote Link Process</td>
<td>RLP</td>
<td>Set of events related to streaming from remote server to mobile device</td>
<td>{gx, xg, gp, pg, f1, f2, d, d, geb, ebg, pf1, fpb}</td>
</tr>
<tr>
<td>Local Link Process</td>
<td>LLP</td>
<td>Events for streaming from MD to LD</td>
<td>{gfb, fbg, b, a, peb, ebp}</td>
</tr>
<tr>
<td>Status update process</td>
<td>SUP</td>
<td>Events for updating status of various jobs and components to the user screen</td>
<td>{s, t, ds}</td>
</tr>
<tr>
<td>Local Device Process</td>
<td>LDP</td>
<td>Events for converting streamed data to final output form</td>
<td>{p,po}</td>
</tr>
<tr>
<td>Remote Server Process</td>
<td>RSP</td>
<td>Events for transformation and processing</td>
<td>{ja, f0, sd, sdc}</td>
</tr>
</tbody>
</table>
The partitioning of events into parallel processes allows the various processes to work in their own speed, synchronizing only when necessary. Events within each process may be partially or completely ordered. Some of the events across different processes may also have to be ordered at synchronization time points.

The relation among all events is given by the event flow equation given earlier whose event-flow graph was shown in Figure 10-1. This makes the deadline reliability estimation equations given in Equation 10-1-Equation 10-6 applicable for concurrent/parallel processes. The equations are repeated below for ease of reference, but our main analysis in this section would be estimation of the waiting time distributions that affect the reliability estimates.

1 For RLM.MaxWait: \( \{\text{elt}(u_n, j_{n-1}) \text{> RLM.MaxWait}\} \), Equation 10-1 is directly applicable for concurrent process, and the time delay is the sum of the time for events between \( j \) and \( u \).

\[
R_{d, ju}(t_{\text{RLM.MaxWait}}) = F_{ju}(t_{\text{RLM.MaxWait}}) = \frac{1}{\sigma_{ju} \sqrt{2\pi}} \int_{-\infty}^{t_{\text{RLM.MaxWait}} - (u - \tau_{ju})^2} e^{-\frac{(u - \tau_{ju})^2}{2\sigma_{ju}^2}} du
\]

where

\[
\tau_{ju} = \tau_j + \tau_{je} + \tau_{jd} + \tau_u
\]

\[
\sigma_{ju}^2 = \sigma_j^2 + \sigma_{je}^2 + \sigma_{jd}^2 + \sigma_u^2
\]

2 For the SRCL.TimeOut event, we have the equations:

\[
R_{d, ab}(t_{\text{SRCL.TimeOut}}) = F_{ab}(t_{\text{SRCL.TimeOut}}) = p_{ld} + \frac{q_{ld}}{\sigma_{ab} \sqrt{2\pi}} \int_{-\infty}^{t_{\text{SRCL.TimeOut}} - (u - \tau_{ab})^2} e^{-\frac{(u - \tau_{ab})^2}{2\sigma_{ab}^2}} du
\]

where

\[
\tau_{ab} = \tau_{peb} + \tau_{gfb} + \tau_{fbg}
\]

\[
\sigma_{ab}^2 = \sigma_{peb}^2 + \sigma_{gfb}^2 + \sigma_{fbg}^2
\]

or, for exponential distribution assumption,

\[
R_{d, ab}(t_{\text{SRCL.TimeOut}}) = 1 - q_{ld} e^{-\gamma_{ab} t_{\text{SRCL.TimeOut}}}
\]
For LRCL.TimeOUT events, we have

\[
R_{d,q} (t_{LRCL\cdot TimeOUT}) = F_{d,q} (t_{LRCL\cdot TimeOUT})
\]

\[
R_{d,q} (t_{LRCL\cdot TimeOUT}) = \frac{q_{lb}}{\sqrt{2\pi(\sigma_{dg}^2 q_{lb} + p_{lb} \tau_{dg}^2)}} \int_{-\infty}^{\infty} e^{-\frac{(x q_{lb} - \tau_{dg})^2}{2(\sigma_{dg}^2 q_{lb} + p_{lb} \tau_{dg}^2)}} dx,
\]

\[\tau_{dg} = \tau_{geb} + \tau_{ebg} + \tau_{pfb} + \tau_{fdp}\]
\[\sigma_{dg}^2 = \sigma_{geb}^2 + \sigma_{ebg}^2 + \sigma_{pfb}^2 + \sigma_{fdp}^2\]

or for exponentially distributed time,

\[
R_{d,q} (t_{LRCL\cdot TimeOUT}) = 1 - e^{-q_{lb} t_{LRCL\cdot TimeOUT} / \tau_{dg}}
\]

The last two equations are not completely independent of each other since the time delays for the event transitions \(gfb \rightarrow fbg\) and \(pfb \rightarrow fdp\) are dependent on the relative speed of the two processes RLP (\(\gamma_{rlp}\)) and LLP (\(\gamma_{llp}\)) and the depth of the queue (LB) on the mobile device. The waiting times \(gfb \rightarrow fbg\) and \(pfb \rightarrow fdp\) events can be estimated based on the following queuing model. Assume \(n\) to be the depth of the queue for temporary storing stream data on MD, measured in terms of the size of data to be streamed at once. Each streaming process from RS to MD takes a unit space from the queue and each streaming process from MD to LD frees up a unit space on the queue. Let the state of the queue at any time be represented by the number of occupied slots on the queue. The state transition may be modeled by the following flow-graph.

![Flow Graph for Representing State of Queue for availability of space](image)

**Figure 10-9: Flow Graph for Representing State of Queue for availability of space**

The stationary (steady state) probability of each of the different states can be obtained using the well known CTMC model, also discussed in section 8.3.3.1. This model gives us the following probabilities for each of the different states

284
\[
\pi_k = \pi_0 \prod_{i=0}^{k-1} \frac{\gamma_{rlp}}{\gamma_{llp}} = \pi_0 \left( \frac{\gamma_{rlp}}{\gamma_{llp}} \right)^k
\]

Since the sum of all stationary probabilities has to be 1, we get
\[
\sum_{k=0}^{n} \pi_k = 1 \Rightarrow \pi_0 \sum_{k=0}^{n} \left( \frac{\gamma_{rlp}}{\gamma_{llp}} \right)^k = 1
\]

\[
\pi_0 = \frac{1}{\sum_{k=0}^{n} \left( \frac{\gamma_{rlp}}{\gamma_{llp}} \right)^k} = \frac{\gamma_{llp}^n \left( \gamma_{rlp} - \gamma_{llp} \right)}{\gamma_{rlp}^{n+1} - \gamma_{llp}^{n+1}}, \text{ for } \gamma_{rlp} \neq \gamma_{llp}
\]

\[
\pi_k = \left( \frac{\gamma_{rlp}}{\gamma_{llp}} \right)^k \frac{\gamma_{llp}^n \left( \gamma_{rlp} - \gamma_{llp} \right)}{\gamma_{rlp}^{n+1} - \gamma_{llp}^{n+1}} = \frac{\gamma_{rlp}^k \gamma_{llp}^{n-k} \left( \gamma_{rlp} - \gamma_{llp} \right)}{\gamma_{rlp}^{n+1} - \gamma_{llp}^{n+1}}
\]

Equation 10-18

The goal here is to estimate the waiting times in the event transitions from gfb\(\rightarrow\)fbg and pfb\(\rightarrow\)fdp. These events are issued by the RLP and LLP processes and the times are dependent on the state of the queue. If \(\tau_k\) corresponds to the mean waiting time when the queue is at state \(k\), then the mean waiting time for the process can be obtained as a weighted sum of the mean waiting time in each state.

\[
\tau = \sum_{i=0}^{n} \pi_k \tau_k
\]

Equation 10-19

\[
\tau_{gfb} = \sum_{i=0}^{n} \pi_k \tau_{k.gfb}
\]

\[
\tau_{pfb} = \sum_{i=0}^{n} \pi_k \tau_{k.pfb}
\]

The waiting time for the gfb (get full buffer) event is negligibly small to cause deadline failure if the queue is not empty. However, when the queue is empty, i.e. at state 0, it can be large, equal to size (SplitFile)/\(\gamma_{rlp}\), which is enough to cause undesired event. Similarly, the waiting time for the pfb event is negligibly small for all states but when the queue is full. In the later case, the waiting time is equal to - size(SplitFile)/\(\gamma_{llp}\).
\[ \tau_{gfb} = \sum_{i=0}^{n} \pi_{k} \tau_{k,gfb} = \pi_{n} \tau_{n,gfb} = \text{size}(\text{SplitFile}) \frac{\gamma_{rlp}^{n} (\gamma_{rlp} - \gamma_{llp})}{\gamma_{rlp}^{n+1} - \gamma_{llp}^{n+1}} \]

\[ \tau_{pfb} = \sum_{i=0}^{n} \pi_{k} \tau_{k,pfb} = \pi_{n} \tau_{n,pfb} = \text{size}(\text{SplitFile}) \frac{\gamma_{rlp}^{n} (\gamma_{rlp} - \gamma_{llp})}{\gamma_{rlp}^{n+1} - \gamma_{llp}^{n+1}} \]

Equation 10-20

Basically, these waiting times are the dominant ones that affect the deadline reliability for the system under study. And hence, we can use these values with the formulas given earlier to estimate the reliability with respect to not missing deadlines.

\[ \tau_{ab} = \tau_{pfb} + \tau_{gfb} + \tau_{pfb} = \text{size}(\text{SplitFile}) \frac{\gamma_{rlp}^{n} (\gamma_{rlp} - \gamma_{llp})}{\gamma_{rlp}^{n+1} - \gamma_{llp}^{n+1}} \]

\[ \tau_{dg} = \tau_{gfb} + \tau_{pfb} + \tau_{pgb} + \tau_{pfb} = \text{size}(\text{SplitFile}) \frac{\gamma_{rlp}^{n} (\gamma_{rlp} - \gamma_{llp})}{\gamma_{rlp}^{n+1} - \gamma_{llp}^{n+1}} \]

Equation 10-21

Table 10-5 shows the estimates for the waiting times and state probabilities

Table 10-5: Waiting times and deadline estimates

| \gamma_{rlp} | \gamma_{llp} | \text{Size}(
| SplitFile) | n | \pi_{0} | \tau_{ab} | \pi_{n} | \tau_{dg} | R_{ab} | R_{dg} | Rd |
|---------|--------|-----------------|--------|-------------|-----------------|--------|-------------|--------|--------|-----|
| 14.375  | 2.84   | 1024            | 1      | 0.164972    | 0.441464        | 0.835028 | 0.328717    | 0.999963 | 0.023899 |
| 3       | 0.006197 | 0.441464        | 3      | 0.006197    | 0.441464        | 0.803659 | 0.328717    | 0.999963 | 0.023899 |
| 5       | 0.000242 | 0.017206        | 5      | 0.000242    | 0.017206        | 0.802483 | 0.328717    | 0.999963 | 0.023899 |
| 7       | 9.43E-06 | 0.000672        | 7      | 9.43E-06    | 0.000672        | 0.802437 | 0.328717    | 0.999963 | 0.023899 |
| 9       | 3.68E-07 | 2.62E-05        | 9      | 3.68E-07    | 2.62E-05        | 0.802435 | 0.328717    | 0.999963 | 0.023899 |
| 11      | 1.44E-08 | 1.02E-06        | 11     | 1.44E-08    | 1.02E-06        | 0.802435 | 0.328717    | 0.999963 | 0.023899 |
Table 10-6 show deadline reliabilities for the system described above.

Table 10-6: Deadline Reliability

For \( n>1 \),

<table>
<thead>
<tr>
<th>Reliability index</th>
<th>Relevant Equation</th>
<th>Estimated value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_{d,ju} )</td>
<td>( R_{d,ju} (t_{RLM,MaxWait}) = 1 - e^{-\gamma_{ju}t_{RLM,MaxWait}} )</td>
<td>0.9999999998</td>
</tr>
<tr>
<td>( R_{d,dg} )</td>
<td>( R_{d,dg} (t_{LRCL,TimeOUT}) = 1 - e^{-q_{lb}t_{LRCL,TimeOUT}/\tau_{dg}} )</td>
<td>0.339497</td>
</tr>
<tr>
<td>( R_{d,ab} )</td>
<td>( R_{d,ab} (t_{SRCL,TimeOut}) = 1 - q_{ld}e^{-t_{SRCL,TimeOut}/\tau_{ab}} )</td>
<td>1</td>
</tr>
<tr>
<td>( R_d )</td>
<td>( R_d = \frac{q_{ld}q_{lb}q_{f}R_{d,ju}R_{d,dg}}{1 - p_{ld}R_{d,ab} - p_{lb}q_{ld} - p_{f}q_{ld}q_{lb}R_d} )</td>
<td>0.025056</td>
</tr>
<tr>
<td></td>
<td>( q_{ld}=q_{lb}=1, \ q_{f}=0.05 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( q_{ld}=q_{lb}=1, \ q_{f}=1, \ SplitFile = 1024KB )</td>
<td>0.339497</td>
</tr>
<tr>
<td></td>
<td>( q_{ld}=q_{lb}=1, \ q_{f}=1, \ SplitFile = 128KB )</td>
<td>0.963776</td>
</tr>
</tbody>
</table>

Although one of the deadline risks is almost removed by this design option as can be seen by the increase of \( R_{d,ab} \) from 0.197291 to 1, the overall deadline reliability is still unacceptable as low as 0.025 for streaming 20MB data, reaching only a maximum of 0.339 for 1MB. This is because, the producer process can still fail while waiting for the consumer to consume part of the data and open up space in the queue.

Fortunately, for this particular system, which is characterized by a slow consumer and fast producer, the producer process can be halted/blocked from producing new data until there is sufficient room for storing it on the MD device. This would not have been possible if the relative process speeds were reversed, i.e. if we had a fast consumer that can be starved to death and a slow producer that cannot be speeded up.

Another alternative to avoid the occurrence of deadline failure on the producer would be to extend room in the queue whenever the producer produces data. This however, would either require infinite memory resources or in any real system that has finite resource, it would result in depletion of resources, another reliability concern discussed later.
10.2.1.4 Reliability with Respect to not Missing Deadline in Concurrent Processes With Synchronized Speeds

In this design alternative, the producer is allowed to produce only when there is sufficient room in the queue to put its data. Once it is permitted to produce, as a result of availability of space on the queue, it will be allowed to write all its data (of maximum size it can produce) without waiting for more space even if that may require extending the queue space. This extension is however limited just to cover one time growth only, i.e. until the producer puts all the data produced in one streaming cycle. After each streaming cycle, the producer has to wait for the availability of sufficient space to produce the next data stream.

With this design approach, the risk of the producer to miss the deadline is nearly removed, since the only waiting time would be the time to copy its data onto the queue and it does not have to wait for the consumer to give free space. This waiting time is very small compared to the deadline time for the process making $R_{d,dg}$ to be nearly 1. However, the consumer can not be assumed to maintain the high reliability obtained in the previous section, since the probability of being in waiting state will be increased due to the halting of the producer process. To estimate the probability, a new model that depicts the change in the design needs to be constructed.

Let $K$ be the maximum number of occupied slots in the queue that trigger producer to move from waiting state to production state. Let $n$ be the maximum number of slots that each stream cycle would require. Let the state of the queue be represented by the number of occupied slots on the queue as before. The possible transition between the various states is shown in Figure 10-10. The system can be in any one of the following group states.

1. The producer is in the blocked state, consumer in operational state, and the state of the queue is $K$ or greater, $s \geq K$. These states are shown on the top level in Figure 10-10.

2. The producer and the consumer are both in operational state and the state of the queue is 1 or greater. These states are those shown at the bottom of Figure 10-10, excluding the first one labeled as 0.
3. The producer is in operational state (fetching data) but the consumer is in waiting state because the state of the queue is empty. This state is the first state in the lower group of states, labeled by 0, in the figure. The waiting time in this state for the consumer is the longest one that can possibly cause missing of deadline since the time can be as large as the producer’s production time. In all other states, both the producer and the consumer can work in their own speed without the risk of missing their deadline.

Figure 10-10: Flow Graph for State Transition of Producer-Consumer Processes with Synchronized speeds.

The above state transitions are explained as follows.

1. If the system is in any one of the upper group, only the consumer is operational and the possible transitions are from state s to state s-1. The transition out of this group to the lower one is when the number of occupied spaces reaches K.

2. If the system is in the lower group, the producer will have to produce up to a maximum of n chunks of data. However, since the consumer is also working in parallel to the producer, the state of the queue may increase in the direction of n+K or decrease up to 0.

3. The producer may finish writing one stream of data on any of this lower group of states. If this state happens to be from 0 to K-1, the producer can continue producing new stream. If it is greater than K, it will have to be blocked and hence
the system goes to the upper state group. \( p_j \) is the probability of completing a
stream of data at state \( j \) in the lower group and moving to the upper group.

The interest here is to evaluate the steady state probability of being at state 0 in the
lower state group, which will then be used to obtain the mean waiting time at this state.
This can be solved by writing a system of linear equations based on the CTMC model.
This may be done by providing numerical values for the different parameters and by
approximating the state-based transition probabilities \( p_j \).

Alternatively, one can use a simplified approach to estimate the probability of failure due
to missing of deadline. Consider the time point at which the producer is triggered to
produce a new data stream as \( t=0 \). From this time point, the producer would take time \( T_p = \frac{\text{size(SplitFile)}}{\gamma_{rlp}} \) to produce the new stream. During the same time frame, the consumer
will also consume the remaining data in the queue. The time taken by the consumer for
this may be given by \( T_c = k \cdot \frac{\text{size(SplitFile)}}{\gamma_{llp}} \) where \( k \) is the triggering ratio i.e. \( k=K/n \).
Both \( T_c \) and \( T_p \) are assumed to be random variables since the two processes may be
influenced by a number of unknown factors. If we define a new random variable, \( T \) as
\[
T = T_c - T_p,
\]
then, \( T \) will give us the random variable representing the waiting time of the consumer
process for the producer. For normally distributed random variables \( T_p \) and \( T_c \), with
respective mean and standard deviation parameters given by, \( (\tau_p, \sigma_p) \) & \( (\tau_c, \sigma_c) \), \( T \) has
also normal distribution with
\[
\tau = \tau_c - \tau_p \quad \text{and} \quad \sigma = \sqrt{\sigma_c^2 + \sigma_p^2} \quad F(t_d) = \frac{1}{\sigma \sqrt{2\pi}} \int_{-\infty}^{t_d} e^{-\frac{(u-\tau)^2}{2\sigma^2}} du
\]

To estimate the deadline reliability, first turn to the event-flow graph of Figure10-4
to see how the derivation of the time distribution function affects the reliability function.

In the timing assessment, the production process consists of the events d, geb, ebg, pfb
and fdp, while the consumption process consists of the events fbg, b, a, p, peb, and gfb.
The timing behavior of each group was represented by one random variable each, and the
two random variables were related at their dependency points that is, through the fdp &
fbg events of the queue (LB). For the consumer process, this approach is different from
the previous one where the time distribution from a to b was sought. Therefore, the occurrence of the undesired events move to event fbg, from event a. As a result, the reliability of missing the deadline is given by the value of the time distribution function, without the effect of $p_{ld}$ and $q_{ld}$. That is,

$$R_{d,ab}(t_{SRCL.TimeOut}) = F_{ab}(t_{SRCL.TimeOut}) = \frac{1}{\sigma \sqrt{2\pi}} \int_{-\infty}^{t_{SRCL.TimeOut}} e^{-\frac{(u-\tau)^2}{2\sigma^2}} du$$

Equation 10-23

Additionally, the undesired event connected to d does not exist now since the missing of deadline by the producer process is removed. For this and the above reason, the flow graph for evaluating the cumulative deadline will look like as shown in Figure 10-11.

Figure 10-11: Flow Graph for evaluating cumulative deadline (MSS Concurrent Processes)

From this, the overall deadline reliability can be obtained as the transfer function from j to $g_X$.

$$R_d = \frac{q_f R_{d,ju} R_{d,ab}}{1 - p_f R_{d,ab}}$$

Equation 10-24

Table 10-7 gives the deadline reliability of the event from ab for different values of k.
Selecting $k=0.2$, as a design parameter, the overall deadline reliability can now be estimated and results for different work loads (different $q_f$ and $p_f$) is shown in Table 10-8.

Table 10-8: Overall deadline reliability for deferent workloads

<table>
<thead>
<tr>
<th>$q_f$</th>
<th>$p_f$</th>
<th>$R_{ab}$</th>
<th>$R_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>0.95</td>
<td>0.990644</td>
<td>0.841123</td>
</tr>
<tr>
<td>0.1</td>
<td>0.9</td>
<td>0.990644</td>
<td>0.913706</td>
</tr>
<tr>
<td>0.15</td>
<td>0.85</td>
<td>0.990644</td>
<td>0.940767</td>
</tr>
<tr>
<td>0.25</td>
<td>0.75</td>
<td>0.990644</td>
<td>0.963598</td>
</tr>
<tr>
<td>0.35</td>
<td>0.65</td>
<td>0.990644</td>
<td>0.973725</td>
</tr>
<tr>
<td>0.45</td>
<td>0.55</td>
<td>0.990644</td>
<td>0.979444</td>
</tr>
<tr>
<td>0.55</td>
<td>0.45</td>
<td>0.990644</td>
<td>0.983118</td>
</tr>
<tr>
<td>0.65</td>
<td>0.35</td>
<td>0.990644</td>
<td>0.985678</td>
</tr>
<tr>
<td>0.75</td>
<td>0.25</td>
<td>0.990644</td>
<td>0.987564</td>
</tr>
<tr>
<td>0.85</td>
<td>0.15</td>
<td>0.990644</td>
<td>0.989011</td>
</tr>
<tr>
<td>0.95</td>
<td>0.05</td>
<td>0.990644</td>
<td>0.990156</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.990644</td>
<td>0.990644</td>
</tr>
</tbody>
</table>
As can be seen from Table 10-8, the reliability with respect to not missing deadline has improved quite significantly to an acceptable level. The following table provides comparison among the various deadline reliability values for different design alternatives:

<table>
<thead>
<tr>
<th>Design Option</th>
<th>$q_f$</th>
<th>$R_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Processes</td>
<td>0.05</td>
<td>0.019363</td>
</tr>
<tr>
<td>Concurrent Processes with Unsynchronized Speeds</td>
<td>0.05</td>
<td>0.025056</td>
</tr>
<tr>
<td>Concurrent Processes with Synchronized Speeds</td>
<td>0.05</td>
<td>0.841123</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>0.913706</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>0.987564</td>
</tr>
</tbody>
</table>

The above sections show us the effect of the process structure on reliability especially in the presence of time sensitive components (as in real time systems). The programs may be ‘correct’ in that there may not be syntax or semantic errors, but a design problem in the process structure can make a software product quite unreliable or its behavior unpredictable.

### 10.2.2 Race Conditions Risk Analysis:

There are two sources of undesired events in the system under study that may be considered as race condition.

The first one results from a possible change of LRCL’s upload limit (by users, system administrators, communication service providers, etc.) while some users have started streaming service.

$$RC_{sd,ec,f2} = \{sd < e_c < f_2 \mid e_c: 'LRCL.UploadLimit(t(e_c))< LRCL.UploadLimit'(t(e_c))\}$$

The change of LRCL.UploadLimit may be considered as a rare but possible event. Let this rate be its arrival rate be represented by $\eta_{lrcl}$.

For each streaming job, event sd occurs towards the beginning of a streaming process while $f_2$ recurs until the job is completed. Thus, the time span between sd and $f_2$ is nearly equal to the service execution time for a given job. Let the execution rate be
exponentially distributed, with execution rate $\gamma$. Based on Equation 8-44, the reliability with respect to race condition may be given as:

$$R_{RCsd,ec,f2} = \frac{\gamma}{\eta_{lrc} + \gamma}$$

**Equation 10-25**

Normally, configuration of LRCL may come very infrequently, i.e. small $\eta_{lrc}$ compared to the fast service rate having large $\gamma$.

$$\eta_{lrc} \ll \gamma \Rightarrow R_{RCsd,ec,f2} \approx 1.$$  

The second race condition is related to event ordering between $sd_c$ and $f_2$, i.e.

$$EO_{sd_c,f2n+1} = \{sd_c > f_2\}$$

This undesired event can be considered as race condition between two processes: $p1: \{jd \rightarrow sd_c\}$ the sequence of events from $jd$ to $sd_c$ and $p2: \{jd \rightarrow f_2\}$ – the sequence of events from $jd$ to $f_2$. Let the process speeds of these processes be approximated by exponential rates - $\gamma_1$ & $\gamma_2$. Then, the reliability with respect to the ordering of the two events is given by:

$$R_{EO,sd_c,f2} = \frac{\gamma_1}{\gamma_1 + \gamma_2}$$

**Equation 10-26**

Unlike the race condition given by Equation 10-25, the processes in Equation 10-26 may have comparable speeds – since they are parallel processes that start at the same event and perform two different tasks. In this case, $\gamma_1 = \gamma_2$, making the reliability with respect to the event order alone around 0.5. The same result will be obtained if the normal distribution is used instead of the exponential distribution. If $(\mu_1, \sigma_1)$ represent the mean-execution time and standard deviation of the first process, $p1$ and $(\mu_2, \sigma_2)$ represent the mean-execution time and standard deviation of the second process, $p2$, then

$$R_{EO,sd_c,f2} = \Phi_T(0)$$

where $T$ represents a random variable with

$$\mu = \mu_1 - \mu_2$$

and

$$\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$$

For comparable speeds, we have $\mu_1 - \mu_2 = 0$, giving $R_{EO,sd_c,f2} = 0.5$. 


This value is unacceptable and hence alternative design must be sought. There are two alternatives to avoid this risk.

1. Include a synchronizer component on the server side or an active component that properly orders p1 and p2, i.e. p2 waits for p1 incase of early arrival, or

2. Ensure that p1 always comes before p2 by putting them in a sequential process, i.e. by firing u only after sd, instead of jd.

The first option requires the inclusion of a new component into the system with its own primary reliability risk. Additionally, delaying p2 can delay event ‘fdp’ which can in turn increases $\tau_{fbg}$ and decrease the reliability with respect to missing the SRCL deadline.

The second option increases the delay between j and u by a further $\mu_1$ time and hence reduces the reliability with respect to missing RLM.MaxWait. However, RLM.MaxWait is a design parameter under the control of service being designed and can be increased so as to counter effect the additional delay. Hence, choosing this option will give higher reliability than the first one. In this case, the reliability with respect to not missing RLM.MaxWait deadline is given by:

$$R_{d,ju}(t_{RLM~MaxWait}) = F(t_{RLM~MaxWait}) = \frac{1}{\sigma_{ju} \sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{(u-\tau_{ju})^2}{2\sigma_{ju}^2}} du$$

where

$$\tau_{ju} = \tau_j + \tau_{je} + \tau_{jd} + \tau_u + \mu_1$$

$$\sigma_{ju}^2 = \sigma_j^2 + \sigma_{je}^2 + \sigma_{jd}^2 + \sigma_u^2 + \sigma_1^2$$

Equation 10-27

10.2.3 Resource Depletions and Congestions

Memory, processors and communication channels are finite resources that may be depleted (even if temporarily) during service provision and may cause system failure. Since the size of resources on the server are relatively very large compared to the requirements of the system, the risk due to depletion on the server is negligible. Hence, one needs to focus the analysis on the mobile device, which has limited resource capacity but provides various services.
The BP service uses two wireless communication channels – LRCL and SRCL, memory and processor. The size of the data to be streamed from remote server RS to local device LD ranges from few megabytes to hundreds of megabytes. But, the available memory size on the mobile device can be a few dozen megabytes, e.g. 32MB.

The speed of transmissions from RS to MD and from MD to LD are determined by various factors including the baud rates of the communication channels, the speed of data consumption at LD, the availability of supplies at LD, which may cause temporary halting of the consumption process, etc. These speeds may vary from device to device and from communication network to communication network. For this particular application, these factors generally create condition in which the consumption process, which transfers data from MD to LD, is slow compared to the production process, which transfers data from RS to MD.

10.2.3.1 Unsynchronized Processes
For unrestricted design in which the production and consumption processes are allowed to run in their full speed, without synchronization there can be system failure due to depletion of memory on the MD, especially if the producer runs faster than the consumer, since the data that is yet to be consumed has to be stored on the MD. The probability of occurrence of memory depletion can be estimated using the birth-death process, where the resource depletion rate is equal to the rate of production and the resource restoration rate is equal to the rate of consumption. The analysis can be conducted similar to the birth-death process discussed in 8.3.3.1.2, and can use Equation 8-56.

For the system under study, the resource depletion rate is equal to the rate of production, i.e. the rate of streaming from RS to MD over the LRCL channel. Let this be represented by $\gamma_{LRCL}$. The resource restoration rate is equal to the rate of consumption, i.e. the rate of streaming from MD to LD over the SRCL channel. Let this be represented by $\gamma_{SRCL}$. Hence, the reliability with respect to depletion of memory can be given by:
A typical mobile device for the MSS system has 32MB (i.e. \( n=32 \)) memory. The producer and consumer processes use 1 MB (\( m=1 \)) to stream at a time. For the rates of production and consumption given by \( \gamma_{LRCL} = 14.375 \) kbps (Kilobytes per second) and \( \gamma_{SRCL} = 2.84 \) kbps, the probability of depletion per demand is:

\[
R_{rd,m} = P(K \leq n-m) = \begin{cases} 
1 - \left( \frac{\gamma_{LRCL}}{\gamma_{SRCL}} \right)^{n-m+1} & \text{for } \gamma_{LRCL} \neq \gamma_{SRCL} \\
1 - \left( \frac{\gamma_{LRCL}}{\gamma_{SRCL}} \right)^{n+1} & \text{for } \gamma_{LRCL} = \gamma_{SRCL}
\end{cases}
\]

Equation 10-28

The above reliability figure, which is based on the actual speed of producer and consumer rates obtained based on test data (for the consumer) and standard speed (for GPRS) for the producer rate, shows that the system can have an extremely poor reliability with respect to resource depletion alone unless the total data to be streamed are small (e.g. 1MB), and jobs are issued only one at a time with no overlap between any of the subtasks of the job.

10.2.3.2 Synchronized Processes

The above problem can be avoided or its effect reduced through synchronization of processes, i.e., by making the producer to work with a synchronized speed as the consumer. The producer can be halted and made to be idle until the consumer consumes the data following each production process. This will make the two processes to have equal rate of operation and hence give \( R_{rd,m} = 0.96875 \).

As can be seen from the reliability figure, even when the two processes are operating at equal rate, there can still be a chance of memory depletion. This is possible for two reasons:
1. The process speeds given in the above are random variables which can actually take different values at different times. Thus equal rate of production and consumption does not imply equal processing speeds at all times.

2. If the producer process requests and is granted a new 1MB memory resource each time it streams data and the memory is left for the garbage collector to return it to the system after the consumer consumes the data, then various possibilities may exist for the producer not to be able to get the 1MB memory resource the next time, among which include:
   2.1. Memory fragmentation may occur and the garbage collector may not get processor time for it to restore the released memory resources to their usable forms
   2.2. Other processes may claim the released memory and memory may not be available for the streaming system to work on.

Thus, further improvements on reducing the risk of memory depletion for the system can be done through memory reserve and recycle strategy discussed below.

10.2.3.3 Synchronized Processes with Memory Reserve and Recycle Strategy
One way of further improving the reliability with respect to depletion of memory is through the implementation of memory reserve and recycle strategy. At the start of a streaming process, the service requests 1MB of memory using which it creates a local buffer (LB) (reserve). Then, the producer process takes an empty chunk from the local buffer and fills it with data. The consumer process takes a full chunk from the local buffer and returns back the empty chunk after consuming the data (recycle). The memory is recycled locally and hence the producer does not have to request the system for new memory in each streaming cycle. With this strategy, the only risk for the streaming service with respect to memory depletion is whether it gets the initial memory resource from the system to start operation. The memory depletion and restoration rate in here is caused by different services that may be sharing the MD, but not on the relative speeds of the consumer and producer processes of the BP service. Hence, it makes sense to assume the restoration rate to be state dependent and hence apply Equation 8-53.
\[ \begin{align*}
R_{rd,m} &= P(K \leq n-m) = \sum_{k=0}^{n-m} \frac{\lambda_r}{\mu_r}^k \frac{1}{k!} = \frac{\sum_{k=0}^{n-m} \left( \frac{\lambda_r}{\mu_r} \right)^k}{\sum_{k=0}^{\infty} \left( \frac{\lambda_r}{\mu_r} \right)^k} \\
&= e^{\lambda_r/\mu_r} - e^{\xi \left( \frac{\lambda_r}{\mu_r} \right)^{n-m+1}} \quad \text{for some } 0 \leq \xi \leq \frac{\lambda_r}{\mu_r}
\end{align*} \]

For \( n=32 \), \( m=1 \), and assuming \( \lambda_r=\mu_r \), we can get
\[ R_{rd,m} = \frac{e - e^{0.5}}{(32-1+1)!} = 1 \]

### 10.2.4 Fault Tolerance and Multimode Operation

It is possible for the different component instances in the composite structure to fail to perform their function correctly. But the service reliability structure is dependent on the critical components that affect the primary outputs of the system, since the design of the system allows fault-tolerance.

For instance, the failure of many of the support components can be tolerated. These include \{ENC/DCR, EL, CHK, UISD, COM/DCM\}. The failure of some of the components such as UISD have degrading effect while others EL and CHK have no effect as far as this service is concerned. Their relevance is in providing system diagnostic information that may be used to analyze system behavior and possible failure cause if any.

Fault tolerance of the ENC/DCR and COM/DCM components is done by excluding them from the composite structure if they are not well-functioning. For instance, stream data that may arrive on MD may come in compressed form or in non-compressed form. If in compressed form, the appropriate de-compressor is selected. Otherwise, the stream will be opened and read as sequence of bytes. Similarly, if data send from the mobile device to the server is encrypted, the relevant decryptor will be selected. If not, it will be read in
plain form. Due to the fact that both the ENC/DCR and COM/DCM components lie in the critical paths of the system, they can possibly have critical failure modes that affect the system output. This happens if a compressed data can not be decompressed or an encrypted data cannot be decrypted.

ENC/DCR is a protective component against possible security threat. Exclusion of the component pair results in increased risk of attack but not necessarily automatic system failure. If \( R_s \) represents the probability of system failure due to security threat, then we can have two reliability values corresponding to the two modes of system operation. If the arrival rate of security threats is given by \( \eta_{st} \), and the reliability of the ENC/DCR per demand is \( R_{ED} \), then the reliability with respect to security threats can be given by the following function.

\[
R_{st}(t) = R_{ED} R_{st|ED}(t) + (1 - R_{ED}) R_{st|ED}(t) = R_{ED} e^{\eta_s t (1 - R_{ED})} + (1 - R_{ED}) e^{\eta_s t}
\]

Equation 10-29

For a service cycle time of \( \tau \), the reliability with respect to security threat per service cycle may be given by:

\[
R_{st}(\tau) = R_{ED} e^{\eta_s (1 - R_{ED}) \tau} + (1 - R_{ED}) e^{\eta_s \tau}
\]

Equation 10-30

On the other hand, COM/DCM provides a support service whose effect is to reduce the size of data communicated from RS to MD. The compression ratio for the COM/DCOM components is measured to be half. The effect of the inclusion or exclusion of the COM/DCM module on reliability is indirectly reflected on those reliabilities that are dependent on data size. A notable example is the deadline reliability, which is dependent on \( q_f \). It was given that

\[
q_f = \frac{\text{size(SplitFile)}}{\text{size(Job.Doc)}}.
\]

But \( \text{size(Job.Doc)} \) is dependent on the presence or absence of COM/DCM (CD for short). For a CD that reduces document size by half, and whose speed of compression and compression is small enough compared to the LRCL speed to be ignored, we get

\[
q_{f|CD} = \frac{\text{size(SplitFile)}}{\text{size(COM(Job.Doc))}} = 2. \frac{\text{size(SplitFile)}}{\text{size(Job.Doc)}} = 2 q_{f|CD}
\]
Therefore, for a mean document size of 20MB and a maximum split file size of 1MB, the conditional deadline reliability with the CD operational and non operational may be given as follows.

\[ R_{d|CD} = 0.913706, \text{ since } q_{f|CD} = 0.1 \]

\[ R_{d|-CD} = 0.841123, \text{ since } q_{f|CD} = 0.05 \]

The equivalent reliability of the two components can be given as:

\[ R_{d,CD} = R_{CD} \times R_{d|CD} + (1-R_{CD}) \times R_{d|-CD} = 0.841123 + 0.07268 \times R_{CD} \]

Equation 10-31

Both of the above analyses assume the failure of the support components to have only degrading failure modes. However, since both of the support components are involved in the transformation of data that contributes to the primary output, it is possible for the components to have critical failure modes, where undetected error output may reach the final destination. In this case, the expression \( 1 - R_s \) has to be replaced by the probability of the system to be in degrading mode, \( R_{xd} \). The previous equations can be modified as follows:

For the ENC/DCR component, we have

\[ R_{st} (t) = R_{ED} e^{-\eta_s (1-R_{ED})t} + R_{EDd} e^{-\eta_s t} \]

Equation 10-32

For a service cycle time of \( \tau \), the reliability with respect to security threat per service cycle may be given by:

\[ R_{st} (\tau) = R_{ED} e^{-\eta_s (1-R_{ED})\tau} + R_{EDd} e^{-\eta_s \tau} \]

Equation 10-33

Similarly for the COM/DCM component, we have:

\[ R_{d,CD}=R_{CD} \times R_{d|CD}+R_{CDa} \times R_{d|-CD}=0.913706 \times R_{CD} + 0.841123 \times R_{CDa} \]

Equation 10-34

In cases where the components have critical failure modes, degradation happens only when the failure mode is degrading but not in critical failure mode.
10.2.5 Restorability and Effectiveness:

Some of the components in the MSS play restorative role. For instance, temporary disconnection of LRCL during fetching of data from RS to MD can be restored by the RLAgent and the missing data can be re-fetched. This is possible because of the partitioning of different activities in different processes, and on MD, there are components that can monitor the required behaviour of the component responsible for streaming of the data to the MD. The only risk associated if there is a frequent and long disconnection of LRCL is increased risk of missing of SRCL deadline time, but which is made to be insignificant through concurrency.

However, in cases where there are no monitoring components that can observe the state transition of other components, the effectiveness of restorations may be limited. Generally, the effectiveness of restorations is dependent on the ability to detect undesired events and the restorability of the affected components.

Consider the mobile streaming system – specifically the interaction between MD and LD through SRCL. The TFM description of the interaction among these three components is given below, where events of interest are selected.

10.2.5.1 Output Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>po</td>
<td>PrintOut</td>
</tr>
<tr>
<td>ue</td>
<td>&lt;UndesiredEvent&gt;</td>
</tr>
</tbody>
</table>

10.2.5.2 Input Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ds</td>
<td>chunk[depth]</td>
</tr>
<tr>
<td>PGM</td>
<td>(Start, Stop)</td>
</tr>
</tbody>
</table>
### 10.2.5.3 Event Descriptors:

<table>
<thead>
<tr>
<th>Program name</th>
<th>Abbreviated Event Descriptor</th>
<th>Event IDs in EFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Externally Activated Events (inputs provided from outside of network)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td>(start, po’, ue’)</td>
<td>St</td>
</tr>
<tr>
<td>Stop</td>
<td>(stop, srcl’)</td>
<td>Sp</td>
</tr>
<tr>
<td>Locally Generated Events (Events occurring within the network)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send</td>
<td>(send, ‘ds, ld’)</td>
<td>Sd</td>
</tr>
<tr>
<td>receive</td>
<td>(receive, po’)</td>
<td>R</td>
</tr>
<tr>
<td>ack</td>
<td>(ack, ds’, md’)</td>
<td>A</td>
</tr>
<tr>
<td>connect</td>
<td>(connect, srcl’)</td>
<td>C</td>
</tr>
<tr>
<td>disconnect</td>
<td>(disconnect, ue’)</td>
<td>D</td>
</tr>
<tr>
<td>resend</td>
<td>(resend, ds’, ld’)</td>
<td>Rsd</td>
</tr>
</tbody>
</table>

### 10.2.5.4 Auxiliary Functions

#### 10.2.5.4.1 \( n(T) = \)

\[
\begin{align*}
(T = _) & \\
\neg(T = _) & \wedge \quad \text{PGM}(r(T)) = \text{stop} \\
\neg \quad & \text{PGM}(r(T)) = \text{start} \\
\neg & \text{srcl.isConnected} \\
\text{srcl.isConnected} & \\
\neg & \text{srcl.isConnected} \\
\neg & (\text{ex}(\text{PGM} = \text{stop})(\text{re} \\
(\text{mrcal} (=\text{start}, T), T))) & \\
\neg & (r(T) = \text{ack}) \\
\neg & (r(T) = \text{send} \lor (r(T) = \text{resend})) \\
\neg & (r(T) = \text{receive}) \\
\neg & (\text{srcl.isConnected}) \\
\neg & (\text{srcl.isConnected}) \\
\neg & (\text{srcl.isConnected}) \\
\neg & (\text{srcl.isConnected}) \\
\neg & (\text{srcl.isConnected}) \\
\end{align*}
\]

#### 10.2.5.4.2 \( \text{Restore}(OP) = \)

| Option 1: | Resend |
| Option 2: | Send |
10.2.5.4.3 \((\text{srcl.isConnected}(T), \text{srcl.event}(T)) =\)

\[
\begin{array}{|c|c|c|}
\hline
(T=\_)& \neg \text{inRange} & \text{True} \\
\neg (T=\_) & \neg \text{inRange} & \text{False} \\
\neg (T=\_)(r(T)=\text{connect}) & \text{inRange} & \text{True} \\
\neg (r(T)=\text{disconnect}) & \text{False} & \text{disconnected} \\
\neg ((r(T)=\text{connect}) \lor (r(T)=\text{disconnect})) & \text{isConnected}(p(T)) & \text{False} \\
\hline
\end{array}
\]

10.2.5.4.4 \(\text{byt}(T) =\)

\[
\begin{array}{|c|c|c|}
\hline
(T=\_) \lor \neg(0\leq \text{index}<\text{ds.depth}) & \neg \text{srcl.isConnected} & \text{ds}[\text{index}++] \\
(0\leq \text{index}<\text{ds.depth}) & \neg \text{srcl.isConnected} & \text{ds}[\text{index}-1] \\
\neg (0\leq \text{index}<\text{ds.depth}) \land (r(T)=\text{send}) & 0<\text{index} & \text{ds}[\text{index}++] \\
\neg (0\leq \text{index}<\text{ds.depth}) \land (r(T)=\text{resend}) & 0=\text{index} & \text{ds}[\text{index}-1] \\
\neg (0\leq \text{index}<\text{ds.depth}) \land (r(T)=\text{prev}(\text{ds})) & 0<\text{index} & \text{ds}[\text{index}-1] \\
\hline
\end{array}
\]

10.2.5.5 Output Functions

10.2.5.5.1 \((\text{po}, \text{ue})(T) =\)

\[
\begin{array}{|c|c|}
\hline
(T=\_) & \neg(\text{ex}((\text{PGM}=\text{receive}))(p(T))) \land \text{in}(r(T)) \land \neg(\text{bytes}(r(\text{po}(p(T))))=\text{prev}(\text{ds})) \\
\neg (T=\_) \land (r(T)=\text{receive}) \land (\text{ex}(\text{PGM}=\text{receive}))(p(T)) & \text{po}(n(T)) \\
\hline
\end{array}
\]

10.2.5.6 Local Dictionary

Type (\text{chunk}) = \text{byte[]} \; // \text{Array of bytes}

Type (\text{depth}) = \text{integer} \; // \text{integer constant}

OP: Options \; // \text{design alternatives}
The SRCL connection has been observed to show frequent unexpected loss of connections, which can further be aggravated by mobility of the devices (MD) in relative to LD, and possible interferences from the environment, including other SRCL communication waves. However, when such sudden loss of connection occurs, the event is detected through an event channel following which the Locallink Agent (LA) instantiates a new LL connector to replace the previous one. All other components of the system can continue operating except LW while the replacement is done. In many of the LD devices, this restorative action can avert the possible loss of data that is sent to the primary output. But in some cases, there can be loss of data during disconnection that will result in unacceptable output.

Two design options in case of disconnection of SRCL are considered. Option 1 is to resend data on reconnection while Option 2 is to simply send the next chunk of data after reconnection. The peripheral device (LD) has no means of communicating whether a previously sent data was consumed or not, giving rise to the possibility of encountering undesired events in both cases – due to possible repeat of data in case of Option 1, and due to missing of data in case of option 2.

10.2.5.7 Undesired Event Analysis through Event Flow Graph (EFG)
Observe the output functions of from the TFM description. The desired output po will be in the correct state only if the transmitted data/chunks appear in their original sequence at ds, that is, without missing any part of it or without any repetition. This is due to the nature of the LD devices, not being fault-tolerant. Since SRCL disconnections can come at any time, either while sending data before it is fully received, or after receiving but before acknowledgement is received by MD, both design options, can fail to prevent system failure, either due to repetition of data in the first case, and missing of data in the second option. The event sequences for the two design options can be studied using the event flow graphs in Figure 10-12.
Let $\gamma_{sd}$ be the execution/service rate of sending event (i.e. $1$/mean duration for sending event), $\gamma_R$ be the execution/service rate of the receiving event (i.e. $1$/mean duration for receiving event), $\eta_d$ be the arrival rate of the disconnection event $d$. Then, the probability
of not encountering disconnections either before sending is completed (and data received by LD), or before receiving event is completed (and acknowledgement is sent to MD), can be found by using the formals for race conditions as follows.

\[
R_{sd} = \frac{\gamma_{sd}}{\gamma_{sd} + \eta_d}
\]

Equation 10-35

\[
R_r = \frac{\gamma_r}{\gamma_r + \eta_d}
\]

Equation 10-36

Had we had a mechanism of knowing exactly when disconnection occurs, either after data is received but acknowledgment was not sent or before data is received, then we would be able to adjust our restoration accordingly, that is resend if data is not received or just send if data was received but acknowledgement was not sent. This type of restoration could have shorted out both unreliability risks, giving reliability structure where each of the above reliabilities appear in parallel to the probability of successful restoration \(R_{rs}\) and the two groups are connected in series, giving a reliability with respect to not encountering disconnection as

\[
R_d = (R_{sd} + R_r - R_{sd} \cdot R_r)(R_r + R_{rs} - R_r \cdot R_{rs})
\]

Equation 10-37

If the probability of successfully reconnecting and resuming the job in the above is always assured, i.e. \(R_{rs}=1\), then we would get \(R_d=1\), and we could claim that temporary disconnections of SRCL would not cause failure to the system.

Unfortunately, the above scenario is not possible in this particular system for the following two reasons:

1. There is no way of knowing the exact disconnection time point in the above system, i.e. whether it lies between send and receive or between receive and acknowledge events. Thus, only one restoration option can be chosen.

2. Restorations may not always be successful for all devices. Some LD devices are observed to respond to disconnections by complete interruptions of jobs.
For the above reasons, only one of the failure events can be covered and hence, the reliability equations corresponding to design options 1 and 2 may be given as follows.

\[ R_d = (R_{sd} + R_{rs} - R_{sd}R_{rs})R_r \]

Equation 10-38

\[ R_r = (R_r + R_{rs} - R_rR_{rs})R_{sd} \]

Equation 10-39

The choice between the above design options is therefore dependent on the relative values of \( R_r \) and \( R_{sd} \), which in-turn depends on the execution or service rate of the two events.

The event that takes longer duration on average will have smaller service rate and hence smaller reliability with respect to disconnection of SRCL as can be deduced from Equation 10-35 and Equation 10-36. Thus, if known, it is this event that needs to be covered to improve the reliability. However, one cannot measure the time duration of the two events as they cannot be observed separately. Thus, a design decision is made based on an assumption in which some LD devices perform better in repeated data items than missing data items (e.g. in case of a printing device as LD, a print out that has a duplicate line would be better than a print out with a missing line), choosing design option 1 over design option 2. Hence, the system reliability with respect to loss of connection of the SRCL, which may be caused by movement of the MD in relative to LD beyond the range of SRCL and environmental interferences,

\[ R_g = (R_{sd} + R_{rs} - R_{sd}R_{rs})R_r \]

Equation 10-40

In cases where the probability of successful reconnection is close to one (which was usually is the case as observed during product testing), we have

\[ R_{srcld} \approx R_r = \frac{\gamma_r}{\gamma_r + \eta_{srcld}} \]

Equation 10-41

The above equation gives the probability of not encountering disconnection per single stream cycle from MD to LD. The corresponding failure rate or reliability per unit job has
to be evaluated by considering the occurrence rate of the undesired event. For this, Equation 8-4, can be used, but first the occurrence rate of the failure event need to be obtained as a reciprocal of the sum of the mean times spent in sending-receiving cycle, i.e.

\[ \eta = \frac{1}{\tau_r + \tau_{sd}} = \frac{\gamma_s}{\gamma_r + \gamma_{sd}} \]

**Equation 10-42**

For a simple approximation of failure rate, where continuous use of the streaming process may be assumed, we may use the referred equation and obtain the following.

\[ \lambda_{src1.d} = \eta(1 - R_{src1.d}) = \frac{\eta \eta_{src1,d}}{\gamma_r + \eta_{src1,d}} \]
\[ = \frac{\gamma_s}{\gamma_r + \gamma_{sd}} \frac{\eta_{src1,d}}{\gamma_r + \eta_{src1,d}} = \frac{\gamma_s \gamma_{sd} \eta_{src1,d}}{(\gamma_r + \gamma_{sd})(\gamma_r + \eta_{src1,d})} \]

**Equation 10-43**

Since, separate measurement of the durations for sending and receiving operations is not possible for the system being studied; we take an approximation assumption, where the two rates to be equal. With this, the failure rate can be simplified to:

\[ \lambda_{src1,d} = \frac{\gamma_s \gamma_{sd} \eta_{src1,d}}{(\gamma_r + \gamma_{sd})(\gamma_r + \eta_{src1,d})} = \frac{\gamma \eta_{src1,d}}{2(\gamma + \eta_{src1,d})} \]

**Equation 10-44**

The service rate for this portion of the network is given in Table 10-3, as \( \gamma_{cb} = 2.84 \) KBps estimated based on performance testing. Since a chunk of data that is send in one SRCL streaming cycle is 4KB, the rate per cycle can be given as \( \eta = 2.84/4 = 0.71 \)ps. \( \gamma = 2\eta = 1.42 \). For a disconnection rate that occurs 1 per 10 hour, the failure rate can be estimated from:

\[ \lambda_{src1,d} = \frac{1.42 \times 1/36000}{2(1.42 + 1/36000)} = 1.38886 \times 10^{-5} \]

**Equation 10-45**
For reliability estimation per execution of a given job, we may convert the event flow graph of Figure 10-12 a to the signal flow graph shown in Figure 10-13, where P is the probability of repeating a job:

Using the signal flow graph, we may obtain the failure density function in Laplace transform as follows.

\[
f(s) = \frac{\eta(1 - R_{srcl,d})}{s + \eta} = \frac{\eta(1 - R_{srcl,d})}{s + \eta(1 - PR_{srcl,d})}
\]

\[
f(t) = \eta(1 - R_{srcl,d})e^{-\eta(1 - PR_{srcl,d})t}
\]

\[
F(t) = \frac{1 - R_{srcl,d}}{1 - PR_{srcl,d}}(1 - e^{-\eta(1 - PR_{srcl,d})t})
\]

Equation 10-46
\[ R_{SRCL,d}(t) = 1 - F(t) \]
\[ = \frac{(1 - P)R_{srcl,d}}{1 - PR_{srcl,d}} + \frac{1 - R_{srcl,d}}{1 - PR_{srcl,d}} e^{-\eta(1 - PR_{srcl,d})t} \]
\[ \lambda(t) = \frac{f(t)}{R_{srcl,d}(t)} = \frac{\eta(1 - R_{srcl,d})e^{-\eta(1 - PR_{srcl,d})t}}{(1 - P)R_{srcl,d} + (1 - R_{srcl,d})e^{-\eta(1 - PR_{srcl,d})t}} \]

Equation 10-47

Substituting the expressions in Equation 10-41 and Equation 10-42 in Equation 10-46, and with \( q = 1 - P \), we get.

\[ R_{srcl,d}(t) = \frac{(1 - P)R_{srcl,d}}{1 - PR_{srcl,d}} + \frac{1 - R_{srcl,d}}{1 - PR_{srcl,d}} e^{-\eta(1 - PR_{srcl,d})t} \]
\[ = \frac{1}{\eta_{srcl,d} + q\gamma} \left\{ q\gamma + \eta_{srcl,d} e^{\gamma\left(\frac{\eta_{srcl,d} + q\gamma}{\eta_{srcl,d} + \gamma}\right)} \right\} \]

Equation 10-48

\[ \lambda(t) = \frac{f(t)}{R_{SRCL,d}(t)} \]
\[ = \frac{\gamma(\eta_{srcl,d} + q\gamma) e^{\gamma\left(\frac{\eta_{srcl,d} + q\gamma}{\eta_{srcl,d} + \gamma}\right)}}{2(\eta_{srcl,d} + q\gamma) q\gamma + \eta_{srcl,d} e^{\gamma\left(\frac{\eta_{srcl,d} + q\gamma}{\eta_{srcl,d} + \gamma}\right)}} \]

Equation 10-49

For an average job size of 20MB, the time for the streaming process would be 20*1024/2.84=7211sec. \( q = 4\text{KB}/20\text{MB} \approx 2 \times 10^{-4} \), the reliability with respect to not failing due to this cause per job may be calculated as:

\[ R_{srcl,d} = 0.94. \]
10.2.6 Overall Service Reliability Assessment:

The following structure identifies the paths (in terms of components) corresponding to each required output.

Table 10-10: Service output classification

<table>
<thead>
<tr>
<th>Output Importance</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>BP={po, pc, pr, jr}</td>
</tr>
<tr>
<td>Secondary</td>
<td>sd = {Status, Transported, Streamed}</td>
</tr>
<tr>
<td>Irrelevant (with respect to the service under study)</td>
<td>Log chk</td>
</tr>
</tbody>
</table>

Classification of components by their reliability role:

Table 10-11: Classification of components by their reliability role

<table>
<thead>
<tr>
<th>Reliability Role</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical (components between user inputs and primary outputs in the connection graph)</td>
<td>Job, Exec, LA, LL, LW, RL, WFS, JO, LB, SRCL, LRCL, IMS, LD, UIJC, JSS</td>
</tr>
<tr>
<td>Protective</td>
<td>ENC/DEC (Encrypt/Decrypt) – from external security threats</td>
</tr>
<tr>
<td></td>
<td>RM – from race conditions on resources</td>
</tr>
<tr>
<td></td>
<td>EC (EventChannel) – from race conditions and undesired failure dependencies</td>
</tr>
<tr>
<td>Monitoring and Restorative</td>
<td>LA – attempts to restore SRCL connection if it is disconnected either due to interference from the environment or temporary movement of the MD from LD outside the maximum range of SRCL coverage. The reconnection may or may not successfully resume system operation depending on the behavior of the LD devices.</td>
</tr>
<tr>
<td></td>
<td>RR – returns memory and storage spaces that have been used by the composite to the system resource pool.</td>
</tr>
<tr>
<td></td>
<td>CHK (Checksum) – can be used to detect the integrity of streamed data after they pass through the LRCL module.</td>
</tr>
<tr>
<td>Redundant</td>
<td>There are diverse redundant modules (e.g. ENC/DEC (Encrypt/Decrypt) and COM/DCM (Compress/Decompress)) that</td>
</tr>
</tbody>
</table>
mainly differ in the algorithms used, from which any pair of them or even a combination may be selected to be member of the composite network. However, for this particular application, only selected single versions are used at any time since any one of the choices seem to satisfy the required reliabilities from the components, and the main system reliability problems are related to timing issues.

Almost all connections are through abstract interfaces and dynamic binding allowing possibilities for reconfiguration of the composite structure based on user’s choice or occurrence of undesired events. For instance, the connections between Exec, LA and RA are created by attaching container components JO and LB as input-output ports of the three components. The separation of input-output object containers from the components allows the components to be replaceable by new instances if and when desired including while the system is under execution. This connection-oriented design also allows inclusion of reliability improvement components such as checksum evaluators as additional components to the composite structure that can read streamed data directly from LB but work in their own process space, avoiding their possible effect on the processes in critical system components.

Replicated redundancy of components and composites is trivially supported since the full composite structure can be created as many times as the number of non-sharable ports. The different composites can run in parallel. However, the reliability improvement from replication of the composite structure on the same device is not significant due to statistical dependency among composites and indirect interactions through shared resources – mainly memory and SRCL communication signals which are susceptible to cross-interferences.

A limited version of redundancy through input variation and repetition can be applied through selection of different device drivers for the local (LD). Stream files generated by different drivers can vary significantly in terms of size. This variation in size of files can result in different inputs, for communication and data management system components – LRCL, MFS, JSS, LD, RLM, LB, LW, LL, LA, Compress/Decompress, SRCL.

At service level, there are different services provided by different composite structures that can, in some cases, be considered as alternative or redundant paths to the service described in here. The variations in some of the modules used in the services as well as the connections in the structures provide some level of diversity that hopefully reduces the statistical failure correlation among the redundant paths. (Example: for some users, the system may be considered to be operational if one of the three services with similar outputs or functionalities but realized by different composite
structures is operational: \( S_u = \text{FXS} \lor \text{BPS} \lor \text{NPS} \). This however
does not exclude the requirement for the developer company to
consider all the three services in series connection since any failure
in any of these services will have to be addressed by the company.
\( S_c = \text{FXS} \land \text{BPS} \land \text{NPS} \).

Supportive

UISD, COM/DCM (Compress/Decompress) – all are necessary for
the service but not critical so long as their failure does not affect the
correctness of the critical output obtained through LD.

Irrelevant

EventLogger, CHK – used for monitoring and debugging of the
system. They may be considered irrelevant as far as the correctness
of the main output is concerned.

Let the combined primary reliability of the critical components be given by \( R_p \) as:

\[
R_p = R_{Job} \cdot R_{Exec} \cdot R_{LA} \cdot R_{LL} \cdot R_{LW} \cdot R_{RL} \cdot R_{WFS} \cdot R_{RO} \cdot R_{LB} \cdot R_{SRCL} \cdot R_{LRCL} \cdot R_{IMS} \cdot R_{LD} \cdot R_{UIJC \cdot R_{ISS} \cdot R_{RM}}.
\]

To obtain the structure and reliability functions, we first obtain the modified service-
module dependency matrix as follows.

\[
S_u(P, M) = \begin{bmatrix}
P & CD & ED & LD & LA & ST_{ED} & ST_{-ED} & D_{CD} & D_{-CD} & MD & S & L & C & SRCL \\
BP & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & \\
BP & 1 & -1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
BP & 1 & 1 & -1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \\
BP & 1 & -1 & -1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \\
SD & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & \\
LOG & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & \\
CHK & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & \\
\end{bmatrix}
\]

Equation 10-50

The first four rows are mutually exclusive paths as can be seen from the combination of
CD and ED columns. Therefore, these rows are additive. The last three are independent
reliability measures for different outputs.
For $X = [x_p \ x_{CD} \ x_{ED} \ x_{LD} \ x_{LA} \ x_{ST_{ed}} \ x_{ST_{cd}} \ x_{D_{cd}} \ x_{MD} \ x_s \ x_L \ x_C \ x_{scdl,d}]$

$$
\Phi(X) = S_R^c \circ X^T
$$

$$
\Phi_{BP}(X) = x_p \cdot x_{LD} \cdot x_{LA} \cdot x_{MD} \cdot x_{scdl,d} \cdot (x_{CD} \cdot x_{ED} \cdot x_{ST_{ed}} \cdot x_{D_{cd}} \cdot x_{SCD} + (1 - x_{CD}) \cdot x_{ED} \cdot x_{ST_{ed}} \cdot x_{D_{cd}} + x_{CD} \cdot (1 - x_{ED}) \cdot x_{D_{cd}} \cdot x_{ST_{ed}} + (1 - x_{CD})(1 - x_{ED})x_{D_{cd}} \cdot x_{ST_{ed}})
$$

$$
\Phi_{SD}(X) = x_p \cdot x_s \cdot x_{MD}
$$

$$
\Phi_{LOG}(X) = x_p \cdot x_L \cdot x_{MD}
$$

$$
\Phi_{CHK}(X) = x_p \cdot x_C \cdot x_{MD}
$$

**Equation 10-51**

The reliability for the complete service can be obtained by considering all outputs in series connection, which is equivalent to finding the conjunction among the above four structure functions.

$$
\Phi_{service}(X) = x_p \cdot x_{LD} \cdot x_{LA} \cdot x_{MD} \cdot x_{scdl,d} \cdot x_s \cdot x_L \cdot x_C \cdot (x_{CD} \cdot x_{ED} \cdot x_{ST_{ed}} \cdot x_{D_{cd}} + (1 - x_{CD}) \cdot x_{ED} \cdot x_{ST_{ed}} \cdot x_{D_{cd}} + x_{CD} \cdot (1 - x_{ED}) \cdot x_{D_{cd}} \cdot x_{ST_{ed}} + (1 - x_{CD})(1 - x_{ED})x_{D_{cd}} \cdot x_{ST_{ed}})
$$

**Equation 10-52**

However, when service degradation or fault tolerance is acceptable, each structure function can be evaluated independently.

The reliability functions can be obtained by substituting the probability of success distribution functions for each of the components in the above.

$$
R_{BP} = \Phi_{BP}(R) = R_p \cdot R_{LD} \cdot R_{LA} \cdot R_{MD} \cdot R_{scdl,d} \cdot (R_{CD} \cdot R_{ED} \cdot R_{ST_{ed}} \cdot R_{D_{cd}} + (1 - R_{CD}) \cdot R_{ED} \cdot R_{ST_{ed}} \cdot R_{D_{cd}}
+ R_{CD} \cdot (1 - R_{ED}) \cdot R_{D_{cd}} \cdot R_{ST_{ed}} + (1 - R_{CD})(1 - R_{ED})R_{D_{cd}} \cdot R_{ST_{ed}})
$$

$$
= R_p \cdot R_{LD} \cdot R_{LA} \cdot R_{MD} \cdot R_{SCDL,d} \cdot (1 - R_{CD}) \cdot R_{ST_{ed}} + (1 - R_{ED}) \cdot R_{ST_{ed}} + (1 - R_{CD})(1 - R_{ED})R_{CD} \cdot 0.913706 + (1 - R_{CD}) \cdot 0.841123(R_{ED} \cdot R_{ST_{ed}} + (1 - R_{ED}) \cdot R_{ST_{ed}}))
$$

**Equation 10-53**

For supportive components that have critical failure mode, we can write:

$$
R_{BP}(P) = R_p \cdot R_{LD} \cdot R_{LA} \cdot R_{MD} \cdot 0.94(R_{ED} \cdot R_{ST_{ed}} + R_{EDd} \cdot R_{ST_{ed}}) \cdot (0.913706 R_{CD} + 0.841123 R_{CDd})
$$

**Equation 10-54**

Even if all the modules in the above have correct implementation having primary reliability of 1, the reliability per demand of the service for an average of 20 MB streaming job size can be estimated as:

$$
R_{BP}(P) = 0.913706 \times 0.94 = 0.865
$$
The reliability for other services that do not have requirement for streaming of large data can be close to 1.

10.3 Chapter Summary
This chapter has covered detailed document driven structural reliability assessment as applied on a mobile streaming system that is built as a case study. It has been shown how the various design decisions and especially the event structure can influence the reliability of the product. Unlike various reliability models that have been proposed over the years, which consider software system as a mere series connection of components, and where the reliability assessment is reduced to estimation of duty cycle of components, the approach described in here looks deeply into the failure cause and effect process and the reliability role played by various components. The quantification of different design alternatives helps to select the design option with the best reliability estimate. The analytical results were also in line with actual observed performance of the different design alternatives.
11. Results, Discussion and Conclusion

11.1 Introduction
In this chapter, the author summarizes the results obtained in this research work and discusses the achievements of the thesis. Based on these results and the experience acquired throughout the research, the author provides his concluding remarks along with a discussion on the way forward.

11.2 Results and Contribution of the Thesis
One of the important measures of the maturity of any engineering discipline is its ability to quantify the quality of its products precisely. Undoubtedly, the most important quality of all products in general, and software in particular, is their reliability or in a broader sense their dependability.

This quality is not only the most important but is also inclusive of other qualities such as maintainability, configurability and restorability, since they affect parameters such as restoration rate which are used in reliability and availability measures. In this research, the author has focused on identifying the necessary criteria that will make the reliability of a software product to be predictable from analysis of information about the reliability of its components and analysis of their interconnections.

To this end, this author has identified and integrated three areas: design for change, precise specification and description, and structure based reliability prediction, into one unified design paradigm, which is named as design for reliability predictability (DRP).

11.2.1 Design for Modular Redundancy and Fault Tolerance
Design for change has brought tremendous developments in software construction. Today, for properly structured systems, replacement of one component by another or making drastic modifications to the hidden parts of modules without affecting other modules is possible and well-known at least in the research community. This however, requires a careful analysis of the decomposition and restriction of the way various components are interconnected. In practice, developers in the industry rarely follow the required strict principles of design for change, resulting in designs where it is hard even
to make the smallest change let alone complete component replacements. Changes to specific aspects have been observed to result in complete breakage of other products. Seemingly compatible components become hard to seamlessly connect with each other or work together. These problems have been known by different names among which include:

- **Architectural mismatch** -(Bierhoff et al., 2007, Cotè, Garlan et al., 1995, Garlan et al., 2009)
- **Tangled codes** - (Gregor Kiczales et al., 1997)
- **The fragile base class problem** (Szyperski, 2002)
- **The dependency hell or dll hell** (James Donald, 2003)

Although many solution proposals to the above problems are usually suggested as improved tools or programming languages, the problems mostly arise due to design flaws, usually as a manifestation of lack of independent changeability of software modules.

Information hiding is the fundamental principle for solving changeability problems that has long been recognized. According to this principle, each module should hide as much information as possible about its internal workings, and only export those parts that are less likely to change. These parts constitute its interfaces.

Design for reliability predictability takes this feature of changeability to a level that supports redundancy, if necessary, and fault tolerance. In designs that allow changeability through proper information hiding, abstraction, and dynamically bound connectors, the replacement of one component by another can be done at maintenance time when the system is not operational or even at runtime through dynamic binding of replacement components that are already deployed on the required computing node. This requires, careful structuring of connectors and events or processes so that replacement of one component by another is supported even when the system is operational. Such changeability properties can be used effectively to produce a system that provides different services with reliability structures that can possibly be different from the mere series connection of components.

Components to be replaced or substituted for other components may be:
• Different instances of the same module
• Functionally equivalent component instances from different modules
• Interface compatible component instances from different modules

From a reliability structure, the first may be used as a means of component restoration and/or replicated redundancy. The second can be used for provision of diversified modular redundancy, while the last is used as a means of providing various services from a set of modules, by selectively connecting only the relevant modules and avoiding cross-interferences by different services.

Despite these possibilities for redundancy and fault tolerance, reliability improvement can be severely limited by statistical dependencies among component failures and may even backfire due to various forms of interaction failure causes. Therefore, design decisions must be thoroughly analyzed based on the various risks that may cause system failure. These risks are not generally limited to coding errors but can involve various types of failure causes where many of them are related to indirect interactions through shared resources while others may even lie outside the designers and developers control.

11.2.2 Precise Specification and Description of Modules and Structural Relations among System Elements

To master the complexity of software systems, the analysis needs to be conducted at different levels of detail by moving through what this author calls a ‘hierarchical aggregate’ structure consisting of elements such as programs, modules, components, composites, variables, connectors, events, processes, services and systems. The relationship among these components is that of aggregation and the categorization is rough and sometimes may depend on the level of choice of analysis.

At a more detailed level, the relation between the various elements must be precisely defined for one to make detailed reliability assessment. This author considers the module as the smallest unit of work for reliability analysis, below which only the implementation code exists that is too detailed to be useful to make any tractable model. To support redundancy and polymorphism, interfaces are considered to have self-standing position in the system. The author connects interfaces with modules using two relations: the provides and the requires.
The reliability of each module can then be established as a function of its ability to provide the interfaces it implements by properly using the interfaces it requires from other modules in the system without failure. This gives interfaces a critical role in reliability assessment. Besides, many problems and system failures, including those that were listed in section 11.2.1, arise due to a mismatch in assumptions that are taken by modules about each other, i.e. about their interfaces. This necessitates having a precise specification of interfaces. The specification should be a semantic one that gives complete behavioral information for the implementer or user modules, as opposed to mere listing of syntactic signatures.

The author applied the TFM specification, a newly developed interface specification technique at the Software Quality Research Laboratory (SQRL), University of Limerick, to describe the behavior of the interfaces precisely. The technique may be applied not only to specify the upper faces of a module which are visible to user modules but also to its lower faces involving those modules that serve as servers or inner modules for the module being described. Such a specification approach is called a full specification since it describes each module’s environment completely.

Producing full specification is necessary for reliability assessment at module level, since failure normally results from a mismatch of the assumptions which the modules can make about each other. Yet, a full specification of modules and construction of correct modules that satisfy the specified requirements is not sufficient to make reliability predictions at a system level. Different forms of interdependencies, direct and indirect, may exist among various entities in a system that affect system reliability as well as many other qualities. These interdependencies make ‘correctness’ itself a relative quality, which can only be assessed with respect to a given operational environment, since a required property for one module can be undesired property for another.

To understand the dependencies among various system elements and to capture any problems that may occur during connections, the author introduced various structure functions in the form of connection matrices that can describe the static and dynamic properties of software systems. These functions include a module-connection matrix, an interface-connection matrix, an input-output connection matrix, and event-flow-
equations, which are found to be instrumental in assessing the reliability of various services provided by a system.

11.2.3 Structure Based Reliability Prediction

The mathematical functions or the reference documentations developed to describe the connections or relations among the various entities in the software system are used to further assess the reliability of the system. Unlike other modeling approaches that view software as a series connection of all components, this author introduced three level of reliability assessment, namely: component reliability, interaction reliability and service reliability. Depending on the organization of various modules, their interconnection, fault-tolerance, redundancy provisions, the resulting structure can range from being worse than series connection to full fledged diverse redundant and fault-tolerant systems.

From the three levels, the author assumes the reliability of the atomic components to be known. This may be obtained through assessment of the reliability of the algorithm or inner components used, the correctness of the implementation code and the correctness of the specification, which would be in series connection to give component reliability. Such assessments can be used to estimate the probability of successful operation of the component per demand. The author then provides a mathematical model that relates the probability of ‘success per demand’ with component demand rate to obtain the component failure rate.

The second level of reliability analysis that is novel, both in the concept and in the approach, is interaction reliability. Using this concept, the author explains and develops a method of quantifying the various forms of failure causes that are usually difficult to understand, test, and even to avoid them from occurring. Interaction failure causes are truly stochastic in that their behavior is determined mostly by information that may be unknown to the designers as well as developers. They consist of failure causes such as race conditions, missing of deadlines, occurrence of deadlocks, resource depletions, incompatible input outputs, etc.

These failure causes are represented in the form of virtual components whose reliability function is derived from the various mathematical functions, especially from event flow graphs, described earlier. These virtual components along with a component’s primary
reliability are then used to derive service reliability functions by using relevant connection functions, making up the third level of reliability assessment. In here, the structure functions are not necessarily in series connection. Components can play different reliability roles such as protective, restorative, supportive or other that improves reliability. Their effect is taken into consideration in evaluating service reliability. Such types of reliability assessments are not common in other structural reliability models, discounting the designers’ efforts in mitigating various forms of failures including interaction failure causes such as race conditions. The full reliability assessment makes the third tributary component of DRP.

11.2.4 DRP Design Imperatives

Based on a close study of the three tributaries of DRP, the author has identified five major imperatives of design for reliability predictability, which are:

1. Design for Modular Redundancy
2. Ensure Non-interfering Behavior of Connections especially among Redundancies
3. Improve Statistical Independence among Redundant Components and Services
4. Describe Component and System Behavior Precisely/Mathematically
5. Model and Predict System Reliability

These imperatives need to be applied at different levels of the software construction process spanning the decomposition, composition, architectural decisions, documentations and reliability assessment. At each point in the design process, designers need to ask different questions and set objectives for their decisions. The kind of design decisions that this author believes are needed to be made are summarized in Table 11-1.

<table>
<thead>
<tr>
<th>Category</th>
<th>Entity</th>
<th>Principle</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Programs</td>
<td>Modularization Based on Information Hiding (Horizontal Decomposition)</td>
<td>Changeability of modules, Hiding likely-to change design decisions</td>
</tr>
<tr>
<td></td>
<td>Variables/Connectors</td>
<td>Reliability Predictability</td>
<td>Resource allocation, distinctly identifiable module outputs, restricted dependencies and connectivity among different</td>
</tr>
<tr>
<td><strong>DRP</strong></td>
<td>323</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modules/Components, Separability of states of different services and avoidance of interferences among different services,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime activities/tasks</td>
</tr>
<tr>
<td>Services (System functionalities)</td>
</tr>
<tr>
<td>Composition</td>
</tr>
<tr>
<td>Connectors</td>
</tr>
<tr>
<td>Shared resource</td>
</tr>
<tr>
<td>Architecture</td>
</tr>
<tr>
<td>Program Structure</td>
</tr>
<tr>
<td>Type Structure</td>
</tr>
<tr>
<td>Interface Structure</td>
</tr>
<tr>
<td>Input-output connection structure</td>
</tr>
<tr>
<td>Event/Process Structure</td>
</tr>
</tbody>
</table>
As can be seen from Table 11-1, reliability is affected by almost all decisions in software design, and construction, including the usage process. Decomposition, composition, resource allocation, scheduling, will all affect whether a network of components will continue to work providing the required service, or not.

The author considers the merging of the three areas of research: design and construction, specification and description, and structure based reliability prediction into one uniform design paradigm as the main contribution of this research work. Traditionally, designers may know that certain design decisions would perform better than others, but would have been unable to quantify by how much the improvement would be made. On the other hand, reliability experts may easily calculate the reliability of a complex network structure that does not necessarily consist of a series connection of components, but when they deal with software, they would get no information about which parts of software belong to which components and what each component is expected to do in the system. Hence, even redundant, protective or restorative components would be considered to be in series connection with all other components. This is where the mathematical
documentation helps most. Designers should be able to communicate the function that must be implemented by each module, the inputs, outputs, and the transfer functions and the relationships among modules, in classical mathematics, i.e. relations, functions and recursions, instead of specialized programming language constructs that are used for implementation. When the mathematical documentations reflect the true behavior of the actual product, the documents can be used to make detailed reliability assessment.

11.3 Discussion and Conclusion
The complexity of software products and the challenges of constructing dependable software systems have been largely understood by software engineers as well as academics in the field, although not so much by people outside the area. A lot has been written on the ever persisting software ‘crisis’ problem, the disasters created by software failures, the complexity of software systems and the absence of any ‘silver bullet’ to tackle that complexity.

To address these issues, various approaches have been proposed over the years by different research groups or schools. These groups include those who consider software construction as:

- as an art where each programmer is set free to write programs just like novels.
- as a management problem where various aspects of the development process has to be managed through a variety of management approaches.
- as a process where the organization (not the product) which undergoes the production process needs to be measured by various forms of maturity models
- as a mathematical exercise that should be based on a set of assumptions and formal language theories where, correctness must be established through rigorous mathematical proofs.
- as a technology or programming language problem where new syntax or programming languages have to be developed every year for addressing the next problem.
- as an engineering problem whose product development should be supported by relevant applied mathematics and science, and the product characteristics must be explained precisely by mathematical functions.
This author’s view belongs to the last school. Similar to other engineering disciplines, software designs and product characteristics must be precisely and unambiguously communicated through relevant classical mathematics. Electrical engineers, for instance, do not need to open and show their products in order to describe and analyze its behavior. They rather use mathematical models, transfer functions, system of equations, etc. to precisely describe what happens with the various section of the product.

In contrast, software designers either use semi-formal documentations such as UML that may not completely and precisely describe the product, or they resort to ‘opening’ the actual product, the program code, when they have to provide detailed information about the product.

The mathematical documentation approaches introduced in this thesis are aimed at solving this problem. The interface specification approach is build upon the decades of effort pioneered by David Parnas, in finding appropriate ways of mathematically documenting the interface of a module. The various structural relations and connection matrices are developed or adapted by the author based on a close study of structural reliability theory and software design approaches.

Most of software quality measures that have been proposed over the years such as ‘kilo lines of code’ (KLOC), quality of service (QoS), various coupling measures, etc. are more subjective measures that mean different things for different situations. On the other hand, reliability is a precisely defined quality measure with specific meaning. When a software product is measured in terms of this quality, precisely, then one can say that software engineering has become closer to other engineering disciplines.

Traditional software design by and large, is still based on intuition, rule of thumb, experience and a few general principles such as information hiding, separation of concerns, loose coupling etc. that are usually overlapping and subject to interpretation. Structural reliability consideration can be a critical component in objectively assessing software design. Design for reliability predictability aims to change the status quo by combining design for change (DC) with focus on reliability together with precise behavioral documentation (PBD), and structural reliability prediction (SRP). i.e.

\[ \text{DRP} = \text{DC} + \text{PBD} + \text{SRP} \]
11.4 The Way Forward

Research on software reliability would be closed only if all the software systems that we continue to depend upon, in our every day life, critical or non-critical, become too dependable to worry upon. Unfortunately, this is not the case. Given the fact that software has to be constructed by human beings, involving dozens or hundreds of them, over a period of time and has to exist in multiple versions for a number of years, its dependability concerns are expected to persist for a time to come.

Therefore, the research in the area of identifying design methods and mathematical models that would allow improved reliability prediction of component based software systems can continue in various ways. Practical mathematical models, and techniques that can be used to understand the various aspects of software behavior can be developed or adapted based on well established classical theories. Methods for studying various structure functions are of critical important. Different structure functions can serve different purposes. The structures for decomposing systems into modules for ease of modifiability can be different from the structures required for analyzing the runtime behavior of systems for reliability prediction, which can still be different from the structures for analyzing system performance. Components that appear in parallel in terms of performance can be in series connection with respect to reliability and vice versa. Further research on various structure functions can possibly help us construct more trustworthy products.

The connections between design for change, mathematical documentation, and structure based reliability assessment can also be extended with the development of relevant tools to seamlessly move from requirements, through designs, implementations upto reliability prediction.
## 12. Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architecture Description Languages</td>
</tr>
<tr>
<td>ADT</td>
<td>Abstract Data Type</td>
</tr>
<tr>
<td>CLT</td>
<td>Central Limit Theorem</td>
</tr>
<tr>
<td>CN</td>
<td>Communication Networks</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off The Shelf</td>
</tr>
<tr>
<td>CTMC</td>
<td>Continuous Time Markov Chain</td>
</tr>
<tr>
<td>CTVR</td>
<td>Center for Telecommunication Value Chain Research</td>
</tr>
<tr>
<td>DC</td>
<td>Design for Change</td>
</tr>
<tr>
<td>DRP</td>
<td>Design for Reliability Predictability</td>
</tr>
<tr>
<td>DTMC</td>
<td>Discrete Time Markov Chain</td>
</tr>
<tr>
<td>EFG</td>
<td>Event-Flow-Graph</td>
</tr>
<tr>
<td>GAO</td>
<td>General Accounting Office</td>
</tr>
<tr>
<td>GPRS</td>
<td>General Packet Radio Service</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IC</td>
<td>Input Channel</td>
</tr>
<tr>
<td>KLOC</td>
<td>Kilo Lines of Code</td>
</tr>
<tr>
<td>LD</td>
<td>Local Device</td>
</tr>
<tr>
<td>LRCL</td>
<td>Long Range Communication Link</td>
</tr>
<tr>
<td>M&amp;C</td>
<td>Monitoring and Control</td>
</tr>
<tr>
<td>MD</td>
<td>Mobile Device</td>
</tr>
<tr>
<td>MG</td>
<td>Module Guide</td>
</tr>
<tr>
<td>MGF</td>
<td>Moment Generating Function</td>
</tr>
<tr>
<td>MID</td>
<td>Module Interface Specification</td>
</tr>
<tr>
<td>MIDD</td>
<td>Module Internal Design Document</td>
</tr>
<tr>
<td>MIL</td>
<td>Module Interconnection Languages</td>
</tr>
<tr>
<td>MSC</td>
<td>Mobile Side Components</td>
</tr>
<tr>
<td>MSS</td>
<td>Mobile Streaming System</td>
</tr>
<tr>
<td>MTTF</td>
<td>Mean Time to Failure</td>
</tr>
<tr>
<td>MTTR</td>
<td>Mean Time to Repair</td>
</tr>
<tr>
<td>MVD</td>
<td>Multi-version Design</td>
</tr>
<tr>
<td>NATO</td>
<td>North Atlantic Treaty Organization</td>
</tr>
<tr>
<td>NHPP</td>
<td>Non-Homogenous Poison Process</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>NVP</td>
<td>N-Version Programming</td>
</tr>
<tr>
<td>OC</td>
<td>Output Channel</td>
</tr>
<tr>
<td>OES</td>
<td>Other Elementary Systems</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OODB</td>
<td>Object-oriented database</td>
</tr>
<tr>
<td>PA</td>
<td>Parent Application</td>
</tr>
<tr>
<td>PBD</td>
<td>Precise Behavioral Documentation</td>
</tr>
<tr>
<td>RB</td>
<td>Recovery Blocks</td>
</tr>
</tbody>
</table>

NATO: North Atlantic Treaty Organization
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RS</td>
<td>Remote Server</td>
</tr>
<tr>
<td>S&amp;D</td>
<td>Specification and Documentation</td>
</tr>
<tr>
<td>SCR</td>
<td>Software Cost Reduction</td>
</tr>
<tr>
<td>SDC</td>
<td>Software Design and Construction</td>
</tr>
<tr>
<td>SDI</td>
<td>Strategic Defense Initiative</td>
</tr>
<tr>
<td>SFI</td>
<td>Science Foundation of Ireland</td>
</tr>
<tr>
<td>SMP</td>
<td>Semi-Markov Process</td>
</tr>
<tr>
<td>SQRL</td>
<td>Software Quality Research Laboratory</td>
</tr>
<tr>
<td>SRCL</td>
<td>Short Range Communication Link</td>
</tr>
<tr>
<td>SRD</td>
<td>Service/System Requirement Document</td>
</tr>
<tr>
<td>SRE</td>
<td>Software Reliability Engineering</td>
</tr>
<tr>
<td>SSS</td>
<td>Server Side Services</td>
</tr>
<tr>
<td>THE.</td>
<td>Technische Hogeschool Eindhoven</td>
</tr>
<tr>
<td>TAM</td>
<td>Trace Assertion Method</td>
</tr>
<tr>
<td>TFM</td>
<td>Trace Function Method</td>
</tr>
<tr>
<td>UA</td>
<td>User Application</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
13. Glossary

1 Relation: is a mapping between two sets of elements, called the domain and range. A function is a special case of a relation where every element in the domain is mapped to exactly one element in the range.

2 Characteristic predicate: for a binary relation R, from domain X to range Y, R(X,Y) denotes a predicate, the characteristic predicate of the set R; R(a,b) is true if and only if (a,b) is a member of R.

3 Functional Document: A functional document is a mathematical document that describes an aspect of a system using a specific function and/or relation that defines the domain and range elements of the system and the mapping from the domain to the range (Parnas and Jan, 1995). A functional document can be used to write specification or description of a product.

3.1 Description: a description states properties of a product; it may include both incidental and required properties

3.2 Specification: is a description that states only required properties

3.3 Full specification: is a specification that states all required properties.

4 Structure: “a partial description of a system showing it as a collection of parts and showing some relations between the parts” (Parnas, 2002).

5 Software architecture: a set of structures each of which describing different aspects of a software system.

6 Resource: An entity that is necessary for the successful functioning of a computer system.

7 Variable: A resource identification mechanism in computer programs.

8 State: The condition of (or the information stored at) a particular resource at any time t.

9 Program: Any sequence of statements executable by a processor, possibly with the help of another program, to transform one state to another.
**Module**: A “work assignment” given to a programmer or group of programmers (Parnas, 1972a). Generally, this consists of a group of programs and a data structure they share.

**Component**: A software component is a collection of programs that is distributed as a unit for use in larger systems. A component may be part of a module or include (parts of) several modules.

**Connector**: a runtime entity or data structure at a specific location and time created by one or more modules through which modules communicate with each other and provide their specified functionality.

**Composite**: a network of components that are connected through connectors and behave as a unit to perform state transformations.

**Interface**: The set of assumptions modules can make about each other in relation to their input/output variables, communication protocols and input-output transfer functions that may be assumed to be satisfied by those who produce modules that use the module being described.

**Abstract Interface**: The set of assumptions that is common to more than one interface and may be limited to the input/output variables and communication protocols.

**Type**: a classification mechanism of variables, connectors, component instances and modules according to their externally observable properties, permissible operations, data representations and functionalities. A type identifies a class of entities that have the same set of externally observable properties and permissible operations. This can then be used to define a relation between entity classifiers - which consist of names of (abstract) interfaces, modules, and data structures - and resource identifiers – consisting of variables and connectors.

**Subtype**: a class of entities that have a common base type, where every property and operation are identical with entities of the same base type, and have additional sets of properties and/or permissible operations that makes them different from their base type. Subtypes can be used to create type hierarchies by defining a subtype relation.
between types. A relation $R_{st}(T,T)$ is a subtype relation if for two type elements $A$ and $B$, we have, $(A,B) \in R_{st} \iff A$ is a subtype of $B$

18 **Substitutability**: an entity, $A$, may be substituted in places where another entity, $B$, is expected, if either both $A$ and $B$ are of the same type or $A$ is a subtype of $B$, i.e. $A$ may be substituted in places where $B$ is expected if $(A,B) \in R_{st}$.

19 **Polymorphism**: “Polymorphism is the ability of something to appear in multiple forms, depending on context, and the ability of different things to appear the same in a certain context.” (Szyperski, 2002).

20 **Event**: An occurrence at any time $t$ that changes the state of one or more variables/resources in a system. In a computer system, an event is usually related to the execution of a program but it may also be caused by some other reason such as passing of time, failure of a resource, etc.

21 **Undesired Event**: An event that can possibly result in partial or complete failure of a service provided by one or more components or an event that can possibly change a resource state from good to bad.

22 **Thread (thread of control)**: A thread may be considered as a subset of the event sets (especially those that arise due to execution of programs) which may be organized as one scheduling unit to be used by a scheduler. A thread is a sequence of program executions or sequence of events $T=\{e\}$ that are controlled as a unit by a single program counter (or event counter).

23 **Process**: a set of threads run by a composite structure $P=\{T_1, \ldots, T_n\}$.

24 **Sequential process**: a process with a single thread of control. $P=\{T\}$ where $\text{Card}(P)=1$. Events in a sequential process occur in a pre-defined order.

25 **Concurrent process**: a process with multiple threads of control that may be scheduled and executed in parallel (or pseudo-parallel for single processor systems). $P=\{T_1, \ldots, T_n\}$ where $\text{Card}(P)>1$. Events from different processes or threads can occur in arbitrary time order (interleaved or even simultaneously) except at synchronization time points.
26 **Event driven process**: a structure in which events are not partitioned into process sets but rather each event is independently scheduled or allocated resource. This can be thought to be a special case of concurrent process where each event acts as a thread of control.

27 **Service**: A *service* is a set of independently usable systems functions with distinctly identifiable input-output variables and specified sets of input-output mappings.

28 **System**: A set of composites providing a set of services.

29 **Software testing**: is the execution of software in a real or simulated environment by providing various inputs (or test sets) and comparing the response with the expected result.

30 **Formal program verification**: is the process of proving programs to be correct.

31 **N-version programming** - is the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification (Avizienis, 1995).

32 **Graph**: a graph $G=(V,E)$ consists of a set of objects $V=\{v_1, v_2, \ldots\}$ called vertices, and another set $E=\{e_1, e_2, \ldots\}$ whose elements are called edges, such that each edge $e_k$ is identified with an unordered pair $(v_i, v_j)$ of vertices (Deo, 1974, p 1).

33 **Incidence Matrix**: is an $n \times e$ matrix $A=[a_{ij}]$, defined on a graph $G$ with $n$ vertices and $e$, as follows:

$$p_{ij} = \begin{cases} 1, & \text{if } j^{th} \text{ edge } e_j \text{ is incident on } i^{th} \text{ vertex } v_i, \text{ and} \\ 0, & \text{otherwise} \end{cases}$$

(Deo, 1974, p 137)

34 **Path Matrix**: a path matrix is defined for a specific pair of vertices in a graph, say $(x,y)$, and is written as $P(x,y)$. The rows in $P(x,y)$ correspond to different paths between vertices $x$ and $y$, and the columns correspond to the edges in $G$. i.e. the path matrix for $P(x,y)=[p_{ij}]$, where

$$p_{ij} = \begin{cases} 1, & \text{if } j^{th} \text{ edge lies in } i^{th} \text{ path, and} \\ 0, & \text{otherwise} \end{cases}$$

(Deo, 1974, p 156)

35 **Failure**: the termination of the ability of an entity to perform a required function.

(Villemeur, 1992b)
36 **Primary failure**: Failure of an entity not caused either directly or indirectly by the failure of another entity.

37 **Secondary failure**: Failure of an entity caused either directly or indirectly by the failure of another entity.

38 **Software bug**: is a deviation of software characteristics from its required characteristics.

39 **Fault**: A fault is the inability of an entity to perform a required function. When a failure occurs, the entity is said to be faulty.

40 **Failure Modes**: A failure mode is the effect by which a failure is observed.

41 **Defect**: A defect is any departure of a characteristic of an entity from requirements, the extent to this departure exceeding acceptable limits under given conditions. Thus, a defect is rather defined as non-compliance with objectives and specification clauses. A defect detected in a system or in a component of the said system may not affect the ability of the system to perform a required function. Conversely, failure of an entity results in defect as it related to a departure of the actual characteristics.

42 **Component Failure**: A component is said to have failed if it receives inputs in its domain but either does not produce output within some specified time, produces outputs that do not satisfy the specified output functions, or produces an error signal indicating that it could not function.

43 **Service Failure**: A service is said to have failed if one or more of its primary outputs fail to satisfy the required properties.

44 **Service Degradation**: A service is in degraded state if its primary outputs are in good state but one or more of its secondary outputs are in bad state.

45 **System Failure**: A system is said to have failed if one of its primary services has failed.

46 **System Degradation**: A system is said to be in degraded state if all of its primary services are operational but any one of its secondary services has failed.
47 **Reliability**: is the ability of an entity to perform its intended function for a specified time period when operating under stated environmental conditions.

48 **Availability**: is the ability of an entity to be in a state to perform a required function under given conditions at a given instant of time.

49 **Maintainability**: is the ability of an entity to be maintained in, or restored to, a state in which it can perform a required function, when maintenance is performed under given conditions and using stated procedures and resources.

50 **Safety**: is the ability of an entity not to cause, under given conditions, critical or catastrophic events.

51 **Random variable**: is a function that reflects the result of a random experiment.

52 **Discrete random variable**: – a random variable that can only assume discrete values, where the discrete values are often non-negative integers

53 **Continuous random variable**: – a random variable $X$ that can assume all values in the interval, where $-\infty < a < b < +\infty$

54 **Stochastic process**: is a mathematical model that relates the realizations of a random variable $X$ to the realizations of an explanatory random variable $T$.

55 **Markov process**: a stochastic process where the state of the system in the future depends only on its present state and is unaffected by its past history (set of its previous states), $P[E(t_n)=E_1, E(t_1)=E_1, E(t_2)=E_2, \ldots, E(t_{n-1})=E_{n-1}] = P[E(t_n)=E_1, E(t_{n-1})=E_{n-1}], \forall t_n, \forall E_n$,

56 **CTMC**: Continuous Time Markov Chain is a Markov process where the transition between states is characterized by having exponential time distribution.

57 **MTTF**: Mean time to (first) failure of an entity, that is the up time of entity before the first failure

58 **MTTR**: Mean time to repair

59 **Failure rate**: this is the limit, if it exists, of the ratio of the conditional probability that the instant of time $T$ of a failure of an entity falls within a given time interval $[t,$
t+ΔT] to the length of this time interval when ΔT tends to zero given that the entity has not failed over [0,t] (Villemeur, 1992b).

60 **Repair rate**: this is the limit, if it exists, of the ratio of the conditional probability that instant T corresponding to the completion of the entity repair be included within a given time interval [t, t+Δt] to the length of this time interval when Δt approaches zero assuming that the entity was failed over time period [0,t] (Villemeur, 1992b).

61 **Asymptotic failure rate**: for modeling purpose, the limit, if any, of the instantaneous failure rate when the time tends to infinity. It is denoted by Λ(∞) (Villemeur, 1992b).

62 **Asymptotic availability**: the limit, if any, of the instantaneous availability when the time tends to infinity. It is denoted by A(∞) (Villemeur, 1992b).

63 **Failure rate upon demand**: This parameter expresses the probability that entity E fails to change state whenever it is required to do so.

64 **Active redundancy**: All components normally are permanently operating

65 **Standby redundancy**: At least one operable component should be ready to replace the on-line components if it fails
14. Bibliography:

53. IEEE (2007) IEEE COMPUTER SOCIETY 60TH ANNIVERSARY AWARDS FOR A CENTURY’S COMPUTER CONTRIBUTION.


63. KNIGHT, J. C. & LEVESON, N. G. A Reply to the Criticisms of the Knight & Leveson Experiment.


80. NATO (1968) *NATO Software Engineering Conference*. Garmisch, Germany.


84. Framingham, MA, USA.


86. Framingham, MA, USA.


DRP


121. VILLEMEUR, A. (1992a) *Reliability, Availability, Maintainability, and Safety Assessment,* Chichester, John Wiley & Sons Ltd.


135.
Appendix A. Mathematical Documents of MSS (Partial)

A.1 TFM Specification of Requirements

A.1.1 BP Service in MSS

Output variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>{p, po, pc, jr}</td>
</tr>
<tr>
<td>Transported</td>
<td>Real [0:100]</td>
</tr>
<tr>
<td>Streamed</td>
<td>Real [0:100]</td>
</tr>
<tr>
<td>Status</td>
<td>{SUBMITTING, ORDERED, REORDERED, CANCELED, STREAM_START, EXEC_START, STREAM_END, EXEC_END, ERROR}</td>
</tr>
<tr>
<td>Log</td>
<td>String</td>
</tr>
<tr>
<td>Chk</td>
<td>String</td>
</tr>
</tbody>
</table>

BP represents an output obtained from an attached local device (LD) and the different symbols assigned in the type set represent the different state conditions of the output. (e.g. p-print in progress, po-print out, pc-cancelled print, jr- job removed)

Input variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Event&gt;</td>
<td>{j, c, r, rj}</td>
</tr>
<tr>
<td>&lt;id&gt;</td>
<td>&lt;XObject&gt;</td>
</tr>
<tr>
<td>Job.MaxInactiveLife</td>
<td>&lt;Integer&gt;</td>
</tr>
</tbody>
</table>

Hardware Resources:

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Resource Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>&lt;LocalDevice&gt;</td>
</tr>
<tr>
<td>MD</td>
<td>&lt;MobileDevice&gt;</td>
</tr>
<tr>
<td>RS</td>
<td>&lt;RemoteServer&gt;</td>
</tr>
</tbody>
</table>

Communication Links:

<table>
<thead>
<tr>
<th>Communication Link</th>
<th>Communication Type</th>
<th>Connected Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRCL</td>
<td>&lt;Long Range Communication Link&gt;</td>
<td>RS → MD</td>
</tr>
<tr>
<td>SRCL</td>
<td>&lt;Short Range Communication Link&gt;</td>
<td>LD ↔ MD</td>
</tr>
</tbody>
</table>
### Abbreviated Event Descriptors:

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>(EVT:j, 'id', 'BP', 'Status', 'Transported', 'Streamed')</td>
</tr>
<tr>
<td>c</td>
<td>(EVT: c, 'id', 'BP', 'Status')</td>
</tr>
<tr>
<td>r</td>
<td>(EVT: r, 'id', 'BP', 'Status', 'Transported', 'Streamed')</td>
</tr>
<tr>
<td>rj</td>
<td>(EVT: rj, 'id', 'BP')</td>
</tr>
<tr>
<td>elt(BP(p(T_{\text{id}}))=po) &gt; Job.MaxInactiveLife</td>
<td>(EVT: elt(BP(p(T_{\text{id}}))=po) &gt; Job.MaxInactiveLife, 'id', 'BP')</td>
</tr>
<tr>
<td>elt(BP(p(T_{\text{id}}))=pc) &gt; Job.MaxInactiveLife</td>
<td>(EVT: elt(BP(p(T_{\text{id}}))=pc) &gt; Job.MaxInactiveLife, 'id', 'BP')</td>
</tr>
</tbody>
</table>

Dependent Events: are events generated inside the composite network (normally in response to externally induced events), that have externally observable effects.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Event Action</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>&lt;id&gt;.url(RS→MD)</td>
<td>(EVT: u, 'id', 'RS', 'MD', 'Status')</td>
</tr>
<tr>
<td>t</td>
<td>&lt;id&gt;.bytes(RS→MD)</td>
<td>(EVT: t, 'id', 'RS', 'MD', 'Status', 'Transported')</td>
</tr>
<tr>
<td>s</td>
<td>&lt;id&gt;.bytes(MD→LD)</td>
<td>(EVT: s, 'id', 'MD', 'LD', 'Status', 'Streamed')</td>
</tr>
<tr>
<td>ue</td>
<td></td>
<td>(EVT:ue, 'Status')</td>
</tr>
<tr>
<td>clear_ue</td>
<td></td>
<td>(EVT:clear_ue, 'Status')</td>
</tr>
</tbody>
</table>

**Local Dictionary:**

For e an event where

\[
e = x(y \rightarrow z) \equiv e = x @ y \land e' = x @ z
\]

**Auxiliary Functions:**

Before(Var, Val)(T) – Value of variable Var before taking Val where Val is last.

\[
(<\text{Var}>x<\text{State}>x<\text{trace}> \rightarrow <\text{State}>)
\]

Before(Var, val)(T)=

\[
\begin{array}{c|c|c}
T= _ \land \neg (T= _) & Var(T) = val & Before(Var, val)(p(T)) \\
\neg (Var(T) = val) & Var(T) &
\end{array}
\]

Next Events: events that are generated following occurrence of other events. In some cases, especially with feedbacks (that are equivalent to loops) the occurrence of an event following another event could be probabilistic.
\[(n(T^{<id>}), p(n(T^{<id>}))) =
\]

| \(-\neg(T^{<id>} = _)\) | \(r(T^{<id>}) = j\) | \(u\) | \(1\) | 
| \(\land\) | \(r(T^{<id>}) = u\) | \(t\) | \(1\) | 
| \(\land\) | \(r(T^{<id>}) = t\) | \(s\) | \(1\) | 
| \(\land\) | \(r(T^{<id>}) = s\) | \(t\) | \(p_t\) | 
| \(\land\) | \((\text{elt}(BP(p(T^{<id>}))=po) > \text{Job.MaxInactiveLife})\) \lor (elt(BP(p(T^{<id>}))=pc) > \text{Job.MaxInactiveLife})\) | \(s\) | \(p_s\) | 
| \(\land\) | \((r(T^{<id>}) = r)\) \land \((BP(p(T^{<id>}))=po \lor BP(p(T^{<id>}))=pc)\) | \(rj\) | \(1\) | 
| \(\land\) | \((\neg(BP(p(T^{<id>}))=po \lor BP(p(T^{<id>}))=pc)\) | \(t\) | \(1\) | 

\(p(e) \rightarrow \) probability of occurrence of \(e\)

**Output Functions:**

\[BP(T^{<id>}) =
\]

| \(-\neg(T^{<id>} = _)\) | \(r(T^{<id>}) = j\) | \(BP(n(T^{<id>}))\) | 
| \(\land\) | \(BP(n(T^{<id>}))\) | \(P\) | 
| \(\land\) | \(0<\text{Streamed}(T^{<id>})<100\) | \(po\) | 
| \(\land\) | \(\text{Streamed}(T^{<id>})=100\) | \(pc\) | 
| \(\land\) | \(\text{Streamed}(T^{<id>})<100\) | \(\text{Streamed}(T^{<id>})=100\) | 
| \(\land\) | \((BP(p(T^{<id>}))=po \lor BP(p(T^{<id>}))=pc)\) | \(\neg(BP(p(T^{<id>}))=po \lor BP(p(T^{<id>}))=pc)\) | 
| \(\land\) | \((\neg(BP(p(T^{<id>}))=po \lor BP(p(T^{<id>}))=pc)\) | \(jr\) | 
| \(\land\) | \((\neg((r(T^{<id>}) = j) \lor (r(T^{<id>}) = s) \lor (r(T^{<id>}) = c) \lor (r(T^{<id>}) = r) \lor (r(T^{<id>}) = rj))))\) | |
(Transported, Streamed) \((T_{<id>})=\)

<table>
<thead>
<tr>
<th>(T_{&lt;id&gt;}=_)</th>
<th>(\neg(T=_)\wedge)</th>
<th>%Transported</th>
<th>% Streamed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r(T_{&lt;id&gt;})=j)</td>
<td>(\neg(BP(p(T_{&lt;id&gt;}))=po \lor BP(p(T_{&lt;id&gt;}))=pc))</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(BP(T_{&lt;id&gt;})=po \lor BP(p(T_{&lt;id&gt;}))=pc)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=t)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=s)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
<tr>
<td>(Transported(p(T_{&lt;id&gt;}))=0)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
<tr>
<td>(Streamed(T_{&lt;id&gt;})&lt;100)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
</tbody>
</table>

**Status** \((T_{<id>})=\)

<table>
<thead>
<tr>
<th>(T_{&lt;id&gt;}=_)</th>
<th>(\neg(T=_)\wedge)</th>
<th>%Transported</th>
<th>% Streamed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r(T_{&lt;id&gt;})=j)</td>
<td>(\neg(BP(p(T_{&lt;id&gt;}))=po \lor BP(p(T_{&lt;id&gt;}))=pc))</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(BP(T_{&lt;id&gt;})=po \lor BP(p(T_{&lt;id&gt;}))=pc)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=u)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=t\wedge)</td>
<td>(Transported(T_{&lt;id&gt;})=0)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
</tr>
<tr>
<td>(Streamed(T_{&lt;id&gt;})=100)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
<tr>
<td>(0&lt;Transported(T_{&lt;id&gt;})&lt;100)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=s\wedge)</td>
<td>(Streamed(T_{&lt;id&gt;})=0)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
</tr>
<tr>
<td>(Streamed(T_{&lt;id&gt;})=100)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
<tr>
<td>(0&lt;Streamed(T_{&lt;id&gt;})&lt;100)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=c\wedge)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
<tr>
<td>(Streamed(T_{&lt;id&gt;})=100)</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td>Streamed(p(T_{&lt;id&gt;}))</td>
<td></td>
</tr>
<tr>
<td>(sd(T_{&lt;id&gt;})=Status(T_{&lt;id&gt;})\wedge Transported(T_{&lt;id&gt;})\wedge Streamed(T_{&lt;id&gt;}))</td>
<td>SUBMITTING</td>
<td>ORDERED</td>
<td>REORDERED</td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=UE)</td>
<td>STREAM_START</td>
<td>STREAM_END</td>
<td></td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=clear(UW))</td>
<td>Status(p(T_{&lt;id&gt;}))</td>
<td>EXEC_START</td>
<td>EXEC_END</td>
</tr>
<tr>
<td>(\neg(BP(p(T_{&lt;id&gt;}))=po \lor BP(p(T_{&lt;id&gt;}))=pc))</td>
<td>Status(p(T_{&lt;id&gt;}))</td>
<td>EXEC_END</td>
<td></td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=UE)</td>
<td>CANCELLED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(r(T_{&lt;id&gt;})=clear(UW))</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Before(Status, UNDESIRABLE EVEN T) \((T_{<id>})=\)

sd(T_{<id>})=Status(T_{<id>}) \wedge Transported(T_{<id>}) \wedge Streamed(T_{<id>})
## A.2 Module Guide (Summary)

Table A-1: Module Guide Summary of MSS

<table>
<thead>
<tr>
<th>1st Level Module</th>
<th>2nd Level Decomposition</th>
<th>3rd Level Decomposition</th>
<th>4th Level Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Support Module</td>
<td>CNS</td>
<td>RemoteLink</td>
<td>RLinkAgent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MFileStreamer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RLMessenger</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MessageManager</td>
</tr>
<tr>
<td></td>
<td>MailLink</td>
<td></td>
<td>AMLAgent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mailer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LocalLink</td>
<td></td>
<td>LLAgent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LLink</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LLStreamer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LLLListener</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HR</td>
<td></td>
<td>LLink</td>
</tr>
<tr>
<td></td>
<td>MD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Streams</td>
<td>IOStreams</td>
<td>FileStream</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NetworkStream</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ByteArrayStream</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IComDecom</td>
<td>GZIPComDecom</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZipComDecom</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ComDecomFactory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ICrypto</td>
<td>DESCrypto</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TripleDESCrypto</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AESCrypto</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RSACrypto</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CryptoFactory</td>
<td></td>
</tr>
<tr>
<td>Software Decision Module</td>
<td>Containers</td>
<td>Hashtable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ResourceManager</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Syncn_Queue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Connectors</td>
<td>Exec</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EStatus</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MException</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IStatus</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IProgress</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Synchronizer</td>
<td></td>
</tr>
<tr>
<td>User Requirements Modules</td>
<td>JobMonitor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MailView</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ObjectEditor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DirectoryServices</td>
<td>Searcher,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1st Level</td>
<td>2nd Level Decomposition</td>
<td>3rd Level Decomposition</td>
<td>4th Level Decomposition</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------</td>
<td>-------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td></td>
<td>Explorer</td>
<td>IMapStream</td>
<td>IMapStream</td>
</tr>
<tr>
<td></td>
<td>SecurityService</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StreamMapServices</td>
<td></td>
<td>IMapStream</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IMapStream</td>
<td>IMapStream</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPrinterService</td>
<td>IPrinterService</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPrinter</td>
<td>IPrinter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ImagPrinter</td>
<td>ImagPrinter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ExcelPrinter</td>
<td>ExcelPrinter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PptPrinter</td>
<td>PptPrinter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DocPrinter</td>
<td>DocPrinter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PdfPrinter</td>
<td>PdfPrinter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PrintFactory</td>
<td>PrintFactory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IMailService</td>
<td>IMailService</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IMail2File</td>
<td>IMail2File</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IFaxService</td>
<td>IFaxService</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IFile2Mail</td>
<td>IFile2Mail</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUtils</td>
<td></td>
<td>MasterMapFactory</td>
<td>MasterMapFactory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Interface Module</td>
<td>Localization</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resources</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UiApplication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Screen</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MenuItem</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Menu</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Controls</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.3 TFM Specification of Interfaces

A.3.1 Short Range Communication Link (SRCL)

Trace function table for communication channel - SRCL

Output variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>&lt;SRCL_MODE&gt;</td>
</tr>
<tr>
<td>MessageReceived</td>
<td>&lt;bytes[]&gt;</td>
</tr>
<tr>
<td>ResponseReceived</td>
<td>&lt;bytes[]&gt;</td>
</tr>
<tr>
<td>BaudRate</td>
<td>&lt;REAL&gt;</td>
</tr>
<tr>
<td>ue</td>
<td>&lt;UndesiredEvent&gt;</td>
</tr>
</tbody>
</table>

Input variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>&lt;SRCL_DEVICE&gt;</td>
</tr>
<tr>
<td>Receiver</td>
<td>&lt;SRCL_DEVICE&gt;</td>
</tr>
<tr>
<td>Noise</td>
<td>&lt;Noise&gt;</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>REAL</td>
</tr>
<tr>
<td>MessageSent</td>
<td>&lt;bytes[]&gt;</td>
</tr>
<tr>
<td>ResponseSent</td>
<td>&lt;bytes[]&gt;</td>
</tr>
<tr>
<td>SRCL</td>
<td>&lt;COM_LINK&gt;</td>
</tr>
<tr>
<td>EVT</td>
<td>{OPEN, MVD_OUT_RANGE, INTERFER, CLOSE, SEND, RESPOND}</td>
</tr>
</tbody>
</table>

Local Type Dictionary:

SRCL_MODE = {Initializing, Opened, Fail, Lost, Closed}

<Noise> = Environmental disturbance that may interfere in the normal functioning of a communication link.

<COM_LINK> - abstract type for communication links.

SRCL – a class of communication link type for short-range communications

<SRCL_DEVICE> - abstract data type for representing devices that are able to establish a communication link – SRCL_LINK

SRCL_DEVICE.ID – identification mechanism for SRCL_DEVICES
**Event Descriptors:**

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Abbreviated Event Descriptor</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>(EVT: OPEN, ‘sender, ‘receiver, sender’, receiver’, srcl.Mode’)</td>
<td>OPEN occurs when sender requests connection with receiver, receiver is expected to acknowledge for the connection to be established</td>
</tr>
<tr>
<td>INTERFER</td>
<td>(EVT:INTERFER, ‘noise, sender’, receiver’, srcl.Mode’)</td>
<td>INTERFER is said to occur at t if Level(_noise) at t &gt; SRCL.MAX_TOLERABLE_NOISE</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>(EVT:TIMEOUT)</td>
<td>TIMEOUT occurs when connection is left idle but open for a time more than a given timeout time, i.e., when – elt(r(T))&gt;TIMEOUT.</td>
</tr>
<tr>
<td>CLOSE</td>
<td>(EVT:CLOSE, _srcl.Mode’)</td>
<td>CLOSE event occurs when sender closes the connection.</td>
</tr>
<tr>
<td>SEND</td>
<td>(EVT:SEND, ‘MessageSent, ‘MessageReceived’, BaudRate’)</td>
<td>SEND occurs when sender sends data to receiver</td>
</tr>
<tr>
<td>RESPOND</td>
<td>(EVT:RESPOND, ‘ResponseSent, ResponseReceived’, sender’)</td>
<td>RESPOND event occurs when receiver responds to the SEND message from sender</td>
</tr>
</tbody>
</table>

**Local Dictionary:**

OUT_RANGE(sender, receiver) = Dist(Loc(sender),Loc(receiver))

>Min(sender.RANGE, receiver.RANGE)
IN_RANGE(sender, receiver) = Dist(Loc(sender),Loc(receiver)) ≤ Min(sender.RANGE, receiver.RANGE)

Loc(item) = Physical location of an item in space and time (4 dimensions) i.e. (item.x, item.y, item.z, t)

Dist(loc1, loc2): distance function between two space points as a function of time.

Dist(loc1, loc2) = \sqrt{(loc1.x-loc2.x)^2 + (loc1.y-loc2.y)^2 + (loc1.z-loc2.z)^2})

### Output Functions

**Mode(T)=**

In the following table, T ≡ T_{srcl}

| \(~(T=\_\) ^ | EVT(r(T))=OPEN | elt(r(T))<SRCL.R | receiver.RES= _ | receiver.RES= ACK | receiver.RES=DENY | \(~(Mode(p(T))=Opened) | Initailizing^5 | Opened | Fail | Fail | Lost | Mode(p(T)) | Closed | Mode(p(T)) |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| \(~(T=\_\) ^ | EVT(r(T))=OPEN | elt(r(T))<SRCL.R | receiver.RES= _ | receiver.RES= ACK | receiver.RES=DENY | \(~(Mode(p(T))=Opened) | Initailizing^5 | Opened | Fail | Fail | Lost | Mode(p(T)) | Closed | Mode(p(T)) |

**MessageReceived(T)=**

<table>
<thead>
<tr>
<th>T= _</th>
<th>EVT(r(T))=SEND</th>
<th>Mode(T)=OPEN</th>
<th>MessageSent(In(r(T)))</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>(~(T=_) ^</td>
<td>EVT(r(T))=SEND</td>
<td>Mode(T)=OPEN</td>
<td>MessageSent(In(r(T)))</td>
<td>Null</td>
</tr>
</tbody>
</table>

**ResponseRecieved(Tl)=**

<table>
<thead>
<tr>
<th>T= _</th>
<th>EVT(r(T))=RESPOND</th>
<th>Mode(T)=OPEN</th>
<th>ResponseSent(In(r(T)))</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>(~(T=_) ^</td>
<td>EVT(r(T))=RESPOND</td>
<td>Mode(T)=OPEN</td>
<td>ResponseSent(In(r(T)))</td>
<td>Null</td>
</tr>
</tbody>
</table>

---

^5 The state of the communication channel between open request from client and server response may be considered to be undefined if the deadline for response is too short to allow any further operation attempt on communication. If the duration is not negligible, the state may be assigned a value – say Initializing.
\textbf{BaudRate}(T)=

<table>
<thead>
<tr>
<th>T=_</th>
<th>(\neg(T=_)\land)</th>
<th>(\neg(EVT(r(T))=SEND \lor EVT(r(T))=RESPOND))</th>
<th>Mode(T)=OPEN</th>
<th>Length(in(r(T)))/(Time(r(T))’-Time(‘r(T)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg(T=_) \land)</td>
<td>EVT(r(T))=SEND \lor EVT(r(T))=RESPOND</td>
<td>Mode(T)=OPEN</td>
<td>Length(in(r(T)))/(Time(r(T))’-Time(‘r(T)))</td>
<td></td>
</tr>
<tr>
<td>(\neg(T=_) \land)</td>
<td>(\neg(Mode(T)=OPEN))</td>
<td>(\neg(EVT(r(T))=SEND \lor EVT(r(T))=RESPOND))</td>
<td>(\neg(Mode(T)=OPEN))</td>
<td>(\neg(EVT(r(T))=SEND \lor EVT(r(T))=RESPOND))</td>
</tr>
</tbody>
</table>

\textbf{ue}(T)=

<table>
<thead>
<tr>
<th>T=_</th>
<th>(\neg(T=_)\land)</th>
<th>(\neg(EVT(r(T))=SEND \lor EVT(r(T))=RESPOND))</th>
<th>Mode(T)=OPEN</th>
<th>(\neg(Mode(T)=OPEN))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg(T=_) \land)</td>
<td>EVT(r(T))=SEND \lor EVT(r(T))=RESPOND</td>
<td>Mode(T)=OPEN</td>
<td>(\neg(Mode(T)=OPEN))</td>
<td></td>
</tr>
<tr>
<td>(\neg(T=_) \land)</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL)) \lor (\neg(Mode(T)=FAIL))</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL))</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL))</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{ue}(T)=

<table>
<thead>
<tr>
<th>T=_</th>
<th>(\neg(T=_)\land)</th>
<th>(\neg(EVT(r(T))=SEND \lor EVT(r(T))=RESPOND))</th>
<th>Mode(T)=OPEN</th>
<th>(\neg(Mode(T)=OPEN))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg(T=_) \land)</td>
<td>EVT(r(T))=SEND \lor EVT(r(T))=RESPOND</td>
<td>Mode(T)=OPEN</td>
<td>(\neg(Mode(T)=OPEN))</td>
<td></td>
</tr>
<tr>
<td>(\neg(T=_) \land)</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL)) \lor (\neg(Mode(T)=FAIL))</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL))</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL))</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{ue}(T)=

<table>
<thead>
<tr>
<th>T=_</th>
<th>(\neg(T=_)\land)</th>
<th>(\neg(EVT(r(T))=SEND \lor EVT(r(T))=RESPOND))</th>
<th>Mode(T)=OPEN</th>
<th>(\neg(Mode(T)=OPEN))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg(T=_) \land)</td>
<td>EVT(r(T))=SEND \lor EVT(r(T))=RESPOND</td>
<td>Mode(T)=OPEN</td>
<td>(\neg(Mode(T)=OPEN))</td>
<td></td>
</tr>
<tr>
<td>(\neg(T=_) \land)</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL)) \lor (\neg(Mode(T)=FAIL))</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL))</td>
<td>(\neg(Mode(T)=OPEN)) \lor (\neg(Mode(T)=FAIL))</td>
<td></td>
</tr>
</tbody>
</table>
A.3.2 Long Range Communication Link (LRCL)

Output variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>&lt;LRCL_MODE&gt;</td>
</tr>
<tr>
<td>BaudRate</td>
<td>&lt;REAL&gt;</td>
</tr>
<tr>
<td>ue</td>
<td>&lt;UNDESIFIED_EVENT&gt;</td>
</tr>
<tr>
<td>MessageReceived</td>
<td>&lt;bytes[]&gt;</td>
</tr>
<tr>
<td>ResponseReceived</td>
<td>&lt;bytes[]&gt;</td>
</tr>
</tbody>
</table>

Input variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender</td>
<td>&lt;LRCLDEVICE&gt;</td>
</tr>
<tr>
<td>receiver</td>
<td>&lt;LRCLDEVICE&gt;</td>
</tr>
<tr>
<td>_carrier</td>
<td>&lt;LRCL_NETWORK&gt;</td>
</tr>
<tr>
<td>_gateWay</td>
<td>&lt;GATEWAY&gt;</td>
</tr>
<tr>
<td>_noise</td>
<td>&lt;Noise&gt;</td>
</tr>
<tr>
<td>LRCL</td>
<td>&lt;ILRCL&gt;</td>
</tr>
<tr>
<td>MessageSent</td>
<td>&lt;bytes[]&gt;</td>
</tr>
<tr>
<td>ResponseSent</td>
<td>&lt;bytes[]&gt;</td>
</tr>
<tr>
<td>EVT</td>
<td>{OPEN, INTERRUPT, CLOSE, RESPOND, SEND}</td>
</tr>
</tbody>
</table>

Local Type Dictionary

LRCL_MODE = {OPENED, FAIL, INITIALIZING, LOST, CLOSED}
GATEWay = {MDS, PublicMDS, IPGateway}
<Noise> = Environmental disturbance that may interfere in the normal functioning of a communication link.
<ILRCL>- abstract type for communication links.
LRCL – a class of communication link type for long-range communications
<LRCLDEVICE>- abstract data type for representing devices that are able to establish a communication link – LRCL_LINK
## Event Descriptors

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Abbreviated Event Descriptor</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>(EVT: OPEN, 'sender, '_internet, '_carrier, '_gateWay, 'receiver, 'sender',receiver', _lrcl.Mode')</td>
<td>OPEN occurs when either the sender or receiver requests communication with the other one and the appropriate acknowledgement is received.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>(EVT: INTERRUPT, sender',receiver', _lrcl.Mode')</td>
<td>INTERRUPT is said to occur at t if an OPEN connection is closed due to an increase in the noise level of the environment beyond the channel’s noise handling capacity</td>
</tr>
<tr>
<td>CLOSE</td>
<td>(EVT:CLOSE, sender',receiver', _lrcl.Mode')</td>
<td>CLOSE event occurs when the device that opened the connection closes it.</td>
</tr>
<tr>
<td>RESPOND</td>
<td>(EVT: RESPOND, 'receiver, '_response, sender')</td>
<td>RESPOND is an event when a receiver responds to a SEND event</td>
</tr>
<tr>
<td>SEND</td>
<td>(EVT:SEND, sender', receiver', '_message)</td>
<td>SEND – is the event of sending a message from sender to receiver</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>(EVT:TIMEOUT)</td>
<td>TIMEOUT occurs when connection is left idle but open for a time more than a given timeout time, i.e., when (-\text{elt}(r(T))&gt;\text{TIMEOUT}).</td>
</tr>
</tbody>
</table>

$$\text{Mode}(T) =$$

<table>
<thead>
<tr>
<th>$T = _ \land \neg(T=_)$</th>
<th>$\neg\text{Available}('\text{In}(r(T)))$</th>
<th>$\text{Available}('\text{In}(r(T)))$</th>
<th>$\neg\text{Available}('\text{In}(r(T)))$</th>
<th>$\text{elt}(r(T))&lt;_\text{gateWay}.\text{TimeOUT}$</th>
<th>$\text{elt}(r(T))\geq_\text{gateWay}.\text{TimeOUT}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\neg\text{(EVT}(r(T))=\text{OPEN})\land\neg\text{Available}('\text{In}(r(T))))</td>
<td>$\text{Mode}(p(T))=\text{OPENED}$</td>
<td>$\neg\text{(Mode}(p(T))=\text{OPENED})$</td>
<td>$\text{Mode}(p(T))=\text{OPENED}$</td>
<td>$\text{Mode}(p(T))=\text{OPENED}$</td>
<td>$\text{Mode}(p(T))=\text{OPENED}$</td>
</tr>
<tr>
<td>$\text{EVT}(r(T))=\text{CLOSE}$</td>
<td>$\neg((\text{EVT}(r(T))=\text{INTERRUPT})\land(\text{elt}(r(T))&gt;\text{TIMEOUT})$</td>
<td>$\text{Mode}(p(T))=\text{INITIALIZING}$</td>
<td>$\text{Mode}(p(T))=\text{INITIALIZING}$</td>
<td>$\text{Mode}(p(T))=\text{INITIALIZING}$</td>
<td>$\text{Mode}(p(T))=\text{INITIALIZING}$</td>
</tr>
<tr>
<td>$\neg((\text{EVT}(r(T))=\text{INTERRUPT})\lor(\text{elt}(r(T))&gt;\text{TIMEOUT})$</td>
<td>$\lor(\text{EVT}(r(T))=\text{OPEN}))$</td>
<td>$\text{Mode}(p(T))=\text{FAIL}$</td>
<td>$\text{Mode}(p(T))=\text{FAIL}$</td>
<td>$\text{Mode}(p(T))=\text{FAIL}$</td>
<td>$\text{Mode}(p(T))=\text{FAIL}$</td>
</tr>
<tr>
<td>$\text{Mode}(p(T))=\text{CLOSE}$</td>
<td>$\text{Mode}(p(T))=\text{CLOSE}$</td>
<td>$\text{Mode}(p(T))=\text{CLOSE}$</td>
<td>$\text{Mode}(p(T))=\text{CLOSE}$</td>
<td>$\text{Mode}(p(T))=\text{CLOSE}$</td>
<td>$\text{Mode}(p(T))=\text{CLOSE}$</td>
</tr>
</tbody>
</table>
Available('In(r(T))) = Available('sender(t)) ^ Available('carrier) ^ Available('getWay) ^ Available('receiver)
Where Available(_object) = _object is either connected or ready for connection at the requested moment.

**MessageRecieved(T)=**

<table>
<thead>
<tr>
<th>T= _</th>
<th>¬(T=_) ^</th>
<th>EVT(r(T))=SEND</th>
<th>Mode(T)=OPEN</th>
<th>¬(Mode(T)=OPEN)</th>
<th>MessageSent(In(r(T)))</th>
<th>Null</th>
</tr>
</thead>
</table>

**ResponseRecieved(T)=**

<table>
<thead>
<tr>
<th>T= _</th>
<th>¬(T=_) ^</th>
<th>EVT(r(T))=RESPOND</th>
<th>Mode(T)=OPEN</th>
<th>¬(Mode(T)=OPEN)</th>
<th>ResponseSent(In(r(T)))</th>
<th>Null</th>
</tr>
</thead>
</table>

**BaudRate(T)=**

<table>
<thead>
<tr>
<th>T= _</th>
<th>¬(T=_) ^</th>
<th>EVT(r(T))=SEND ∨ EVT(r(T))=RESPOND</th>
<th>Mode(T)=OPEN</th>
<th>¬(Mode(T)=OPEN)</th>
<th>Length(In(r(T)))/(Time(r(T)')-Time('r(T)))</th>
<th>0</th>
</tr>
</thead>
</table>

**UE(T)=**

<table>
<thead>
<tr>
<th>T= _</th>
<th>¬(T=_) ^</th>
<th>EVT(r(T))=OPEN^</th>
<th>Available ('In(r(T)))</th>
<th>¬Available ('In(r(T)))</th>
<th>elt(r(T))&lt;_gateWay.TimeOUT</th>
<th>elt(r(T))≥_gateWay.TimeOUT</th>
<th>Mode(p(T))=Opened</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>(EVT(r(T))=INTERRUPT ∨ elt(r(T))&gt;TIMEOUT)</th>
<th>Mode(p(T))=Opened</th>
</tr>
</thead>
</table>

| EVT(r(T))=CLOSE | ¬(EVT(r(T))=OPEN ∨ (EVT(r(T))=INTERRUPT) ∨ (elt(r(T))>TIMEOUT) ∨ (EVT(r(T))=CLOSE)) |

| | | | NONE | NONE | CONNECTION_FAIL | CONNECTION_LOST | NONE | UE(p(T)) |
A.3.3 LocalDevice (LD)

Output variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>printOut</td>
<td>&lt;PrintOut&gt;</td>
</tr>
<tr>
<td>Mode</td>
<td>&lt;LD_MODE&gt;</td>
</tr>
<tr>
<td>RES</td>
<td>&lt;RESPONSE&gt;</td>
</tr>
<tr>
<td>_length</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>dtr (data terminal ready)</td>
<td>&lt;bool&gt;</td>
</tr>
<tr>
<td>llink</td>
<td>&lt;LLink&gt;</td>
</tr>
<tr>
<td>ldPortID</td>
<td>&lt;SRCL_DEVICE.ID&gt;</td>
</tr>
<tr>
<td>ue</td>
<td>&lt;UndesiredEvent&gt;</td>
</tr>
</tbody>
</table>

Input variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVT</td>
<td>{INITIALIZE, DISCOVER, OPEN, WRITE, RUNOUTOFSUPPLY, REFIL, DSR (Data Set Ready), CONSUMED}</td>
</tr>
<tr>
<td>sender</td>
<td>&lt;SRCL_DEVICE&gt;</td>
</tr>
<tr>
<td>srcl</td>
<td>&lt;SRCL_LINK&gt;</td>
</tr>
<tr>
<td>data</td>
<td>&lt;byte[]&gt;</td>
</tr>
<tr>
<td>supply</td>
<td>{Paper, Ink}</td>
</tr>
</tbody>
</table>
## Event Descriptors

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Value</th>
<th>Abbreviated Event Descriptor</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZE</td>
<td></td>
<td>(EVT:INITIALIZE, _ld.Mode’)</td>
<td>INITIALIZE is the event that sets up the device to be ready for use</td>
</tr>
<tr>
<td>DISCOVER</td>
<td>&lt;ID&gt;</td>
<td>(EVT:DISCOVER, Value’)</td>
<td>DISCOVER is signal sent by sender requesting the address of the local device</td>
</tr>
<tr>
<td>OPEN</td>
<td></td>
<td>(EVT: OPEN, ‘sender, srcl, RES’, srcl’, dtr’, ld.Mode’</td>
<td>OPEN occurs when sender requests connection with the local device as server. The response is to be given by the variable RES for the connection to be established. This may be followed by raising of DTR value of the local device if is ready</td>
</tr>
<tr>
<td>DSR</td>
<td></td>
<td>(EVT:DSR, ‘sender, srcl, dtr’, ld.Mode’</td>
<td>Data Set Ready is a control signal that may be sent by sender to receiver. Receiver may respond by raising DTR – which represents Data Terminal Ready</td>
</tr>
<tr>
<td>WRITE</td>
<td>&lt;Integer&gt;</td>
<td>(EVT:WRITE, ‘data, srcl, Value’, printOut’</td>
<td>Write event converts the bytes sent to the local device in digital form into a printout form on paper</td>
</tr>
<tr>
<td>CONSUMED</td>
<td></td>
<td>(EVT: CONSUMED, ‘data, llink’, ld.Mode’)</td>
<td>CONSUMED occurs when data given as input in the mrcall Write event is used by the device</td>
</tr>
<tr>
<td>RUNOUTOFSUPPLY</td>
<td></td>
<td>(EVT: RUNOUTOFSUPPLY, ld.Mode’)</td>
<td>Device needs supply, paper or ink, to continue to operate i.e. (Number(paper in LD)=0) ∨ (Level(ld.ink)&lt;ld.MinmumInk)</td>
</tr>
<tr>
<td>REFILL</td>
<td></td>
<td>(EVT: REFILL, ld.Mode’)</td>
<td>Device refilled with input supply.</td>
</tr>
<tr>
<td>At(srcl.Lost)</td>
<td></td>
<td>(EVT: At(_srcl.Lost), ld.Mode’)</td>
<td></td>
</tr>
</tbody>
</table>
Trace Function Dictionary

1. Consume: data[] → data[]. Consume(dat) – returns the part of the data that is already consumed by the device from the given data, dat.
2. isOperational: device → bool, isOperational(Device) returns a true if the given device is working properly and has sufficient supplies.
3. at(M) – The event generated at (or due to) the change of mode to M. at(M)=_ if there has not been any change on the mode. (<Mode>→<event_descriptor>)
4. after(M) – The event generated at the end of mode M, i.e. when it is changed to some other Mode.
5. LD_MODE = { READY4CONNECT, CONNECTED, READY4WRITE, CONSUME, WAIT4SUPPLY, FAULTY }

Auxiliary Functions

Mode(T)=

<table>
<thead>
<tr>
<th>T= _</th>
<th>EVT(τ(T))=INITIALIZE</th>
<th>isOperational(_ld)</th>
<th>¬isOperational(_ld)</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬(T= _) ∧</td>
<td>EVT(τ(T))=OPEN</td>
<td>(Mode(p(T))= READY4CONNECT)</td>
<td>¬(Mode(p(T))= READY4CONNECT)</td>
</tr>
<tr>
<td>EVT(τ(T))=OPEN</td>
<td>EVT(τ(p(T)))= RUNOUTOF_SUPPLY</td>
<td>¬(EVT(τ(p(T)))= RUNOUTOF_SUPPLY)</td>
<td></td>
</tr>
<tr>
<td>¬(T= _) ∧</td>
<td>EVT(τ(T))=DSR</td>
<td>(Mode(p(T))= CONNECTED)</td>
<td>¬(Mode(p(T))= CONNECTED)</td>
</tr>
<tr>
<td>EVT(τ(T))=DSR</td>
<td>EVT(τ(p(T)))= RUNOUTOF_SUPPLY</td>
<td>¬(EVT(τ(p(T)))= RUNOUTOF_SUPPLY)</td>
<td></td>
</tr>
<tr>
<td>¬(T= _) ∧</td>
<td>EVT(τ(T))=WRITE</td>
<td>(Mode(p(T))= READY4WRITE)</td>
<td>¬(Mode(p(T))= READY4WRITE)</td>
</tr>
<tr>
<td>EVT(τ(T))=WRITE</td>
<td>EVT(τ(p(T)))= RUNOUTOF_SUPPLY</td>
<td>¬(EVT(τ(p(T)))= RUNOUTOF_SUPPLY)</td>
<td></td>
</tr>
<tr>
<td>¬(T= _) ∧</td>
<td>EVT(τ(T))=CONSUME D</td>
<td>Length(Consume('_data'))=Length('_data(i n(mrcall(WRITE, T))))</td>
<td>Length(Consume('_data'))&lt;Length('_data(i n(mrcall(WRITE, T))))</td>
</tr>
<tr>
<td>EVT(τ(T))=CONSUME D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬(T= _) ∧</td>
<td>EVT(τ(T))=CLOSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVT(τ(T))=CLOSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬(T= _) ∧</td>
<td>EVT(τ(T))=At(_srcl.Lost)</td>
<td>¬(EVT(τ(T))=INITIALIZE ∨ EVT(τ(T))=OPEN ∨ EVT(τ(T))=WRITE ∨ At(CONNECTED) ∨ EVT(τ(T))=CLOSE ∨ EVT(τ(T))=REFILL ∨ EVT(τ(T))= RUNOUTOF_SUPPLY ∨ EVT(τ(T))=CONSUMED ∨ EVT(τ(T))=CLOSE EVT(τ(T)) =At(_srcl.Lost)</td>
<td></td>
</tr>
</tbody>
</table>

Output Functions
\textbf{printOut}(T) =

\[
\begin{array}{|c|c|c|}
\hline
\text{T=\_} & \neg(T=\_) & \neg(\text{EVT}(\tau(T))=\text{Write}) \\
\text{\^} & \text{\^} & \neg(\neg(\text{Mode}(p(T))=\text{READY4WRITE})) \\
\text{\^} & \text{\^} & \neg(\text{EVT}(\tau(T))=\text{Write}) \\
\hline
\end{array}
\]

\text{LDPrint} : \text{is a function that converts data from digital representation to print out document on paper : LDPrint} : \text{bytes[]} \rightarrow \text{PaperDocument}

\textbf{RES}(T) \ (ld.RES)=

\[
\begin{array}{|c|c|c|}
\hline
\text{T=\_} & \neg(T=\_) & \neg(\text{EVT}(\tau(T))=\text{Write}) \\
\text{\^} & \text{\^} & \neg(\neg(\text{Mode}(p(T))=\text{READY4WRITE})) \\
\text{\^} & \text{\^} & \neg(\text{EVT}(\tau(T))=\text{Write}) \\
\hline
\end{array}
\]

\text{length}(T)=

\[
\begin{array}{|c|c|c|}
\hline
\text{T=\_} & \neg(T=\_) & \neg(\text{EVT}(\tau(T))=\text{CONSUMED}) \\
\text{\^} & \text{\^} & \neg(\neg(\text{Mode}(p(T))=\text{READY4WRITE})) \\
\text{\^} & \text{\^} & \neg(\text{EVT}(\tau(T))=\text{CONSUMED}) \\
\hline
\end{array}
\]

\text{\text{dtr}}(T)=

\[
\begin{array}{|c|c|}
\hline
\text{T=\_} & \neg(T=\_) \\
\text{\^} & \neg(\text{Mode}(T)=\text{READY4WRITE}) \\
\hline
\end{array}
\]

\text{\text{ldPortID}}(T)=

\[
\begin{array}{|c|c|}
\hline
\text{T=\_} & \neg(T=\_) \\
\text{\^} & \neg(\text{EVT}(\tau(T))=\text{DISCOVER}) \\
\hline
\end{array}
\]

\text{LDPrint} (\text{data}(\text{in}(\tau(T))))
\text{llink}(T) =
\text{Events generated and communicated to } _\text{llink} \text{ on sender.}

\begin{align*}
T &= _ \\
\neg (T = _) & ^\wedge \\
\text{At(READY4WRITE)} & \\
\text{After(READY4WRITE)} & \\
\text{EVT}(\tau(T)) & = \text{CONSUMED} \\
\text{At(CONNECTED)} & \neg (\text{RES}(T) = \text{ACK}) \\
\wedge & \\
\neg (\text{RES}(T) = \text{ACK}) & \\
\text{EVT}(\tau(T)) & = \text{CLOSE} \\
\neg (\text{At(READY4WRITE)} \lor \\
\text{After(READY4WRITE)} \lor \\
\text{After(CONSUME)} \lor \\
\text{At(CONNECTED)} \lor \\
\text{EVT}(\tau(T)) & = \text{CLOSE})
\end{align*}

\text{UE}(T) =

\begin{align*}
T &= _ \\
\neg (T = _) & ^\wedge \\
\text{EVT}(\tau(T)) & = \text{INITIALIZE} \\
\neg \text{isOperational}(_\text{ld}) & \\
\neg \text{isOperational}(_\text{ld}) & \\
\text{EVT}(\tau(T)) & = \text{OPEN} \\
\neg (\text{Mode}(p(T)) = \text{READY4CONNECT}) & \\
\text{EVT}(\tau(T)) & = \text{DSR} \\
\neg (\text{Mode}(p(T)) = \text{CONNECTED}) & \\
\text{EVT}(\tau(T)) & = \text{WRITE} \\
\neg (\text{Mode}(p(T)) = \text{READY4WRITE}) & \\
\text{EVT}(\tau(T)) & = \text{RUNOUTOFSUPPLY} \\
\text{EVT}(\tau(T)) & = \text{REFILL} \\
\text{EVT}(\tau(p(T))) & = \text{RUNOUTOFSUPPLY} \\
\neg (\text{EVT}(\tau(p(T)))) & = \text{RUNOUTOFSUPPLY} \\
(\text{EVT}(\tau(T)) & = \text{CONSUMED} ) ^\wedge \\
\text{Length}(\text{Consume}(_\text{data})) & = \text{Length}(_\text{data} \\
in(\text{mrcall}(\text{WRITE}, T))) \\
\text{Length}(\text{Consume}(_\text{data})) & < \text{Length}(_\text{data} \\
in(\text{mrcall}(\text{WRITE}, T))) \\
\text{EVT}(\tau(T)) & = \text{CLOSE} \\
(\text{EVT}(\tau(T)) & = \text{At(_srcl.Lost)}) ^\wedge \\
((\text{Mode}(p(T)) = \text{CONSUME}) \lor \\
(\text{Mode}(p(T)) = \text{WAIT4SUPPLY}) \\
(\neg (\text{Mode}(p(T)) = \text{CONSUME})) \lor \\
(\text{Mode}(p(T)) = \text{WAIT4SUPPLY}) \\
(\neg (\text{Mode}(p(T)) = \text{CONSUME})))
\end{align*}
A.3.4 MFileStreamer

Output Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ds[]</td>
<td>&lt;InputStream&gt;</td>
</tr>
<tr>
<td>Progress</td>
<td>&lt;IProgress&gt;</td>
</tr>
<tr>
<td>ue</td>
<td>&lt;UndesiredEvent&gt;</td>
</tr>
</tbody>
</table>

Input Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM</td>
<td>{ MFileStreamer, getStream, refresh, stop, cancel}</td>
</tr>
<tr>
<td>urls[]</td>
<td>&lt;URL&gt;</td>
</tr>
<tr>
<td>uec</td>
<td>&lt;IUEChannel&gt;</td>
</tr>
<tr>
<td>strmCmpse</td>
<td>List&lt;STREAMKEY&gt;</td>
</tr>
<tr>
<td>maxSplitSize</td>
<td>Integer</td>
</tr>
</tbody>
</table>

Lower face Components:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pa</td>
<td>&lt;IPA&gt;</td>
</tr>
<tr>
<td>rlmessenger</td>
<td>&lt;RLMessenger&gt;</td>
</tr>
<tr>
<td>comDecom</td>
<td>&lt;IComDecom&gt;</td>
</tr>
</tbody>
</table>

Constraints:

maxSplitSize ≤ MDS.UploadLimit

Access Programs by Users of MFileStreamer:

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>getStream</td>
<td>&lt;InputStream&gt;</td>
<td></td>
<td>(PGM:openStream, ‘Value’)</td>
</tr>
<tr>
<td>refresh</td>
<td>&lt;InputStream&gt;</td>
<td></td>
<td>(PGM:refresh, ‘Value’)</td>
</tr>
<tr>
<td>seek</td>
<td>&lt;Integer&gt;:pos</td>
<td></td>
<td>(PGM:seek, ‘pos’)</td>
</tr>
<tr>
<td>cancel</td>
<td></td>
<td></td>
<td>(PGM:cancel)</td>
</tr>
<tr>
<td>stop</td>
<td></td>
<td></td>
<td>(PGM:stop)</td>
</tr>
</tbody>
</table>

Auxiliary functions:
\( scount(T) = \)

<table>
<thead>
<tr>
<th>( \lnot(T = _) )</th>
<th>PGM(r(T))=MFileStreamer</th>
<th>( -1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM(r(T))=next</td>
<td>( (scount(p(T))=-1) \lor (scount(p(T)=0) )</td>
<td>( %\text{streamCount} )</td>
</tr>
<tr>
<td>PGM(r(T))=previous^</td>
<td>( (scount(p(T))=-1) \lor (scount(p(T)=0) )</td>
<td>streamCount-1</td>
</tr>
<tr>
<td>PGM(r(T))=refresh^</td>
<td>( \lnot(scount(p(T))=-1) )</td>
<td>scount(p(T))-1</td>
</tr>
<tr>
<td>PGM(r(T))=seek^</td>
<td>( 0 \leq (\text{in(r(T)}) &lt; \text{streamCount} )</td>
<td>scount(p(T))</td>
</tr>
<tr>
<td></td>
<td>( \lnot(0 \leq (\text{in(r(T)}) &lt; \text{streamCount} )</td>
<td>0</td>
</tr>
<tr>
<td>( \lnot((\text{PGM(r(T))=MFileStreamer}) \lor(\text{PGM(r(T))=previous}) \lor(\text{PGM(r(T))=refresh}) \lor(\text{PGM(r(T))=seek}) )</td>
<td>scount(p(T))</td>
<td></td>
</tr>
</tbody>
</table>
**noStreams**

Vector Function Table for vector variables given by

[Index(scount), Offset(scount), streamLen(scount)]

<table>
<thead>
<tr>
<th><code>scount</code></th>
<th><code>maxSplitSiz &gt; size(urls[0]) - offset(scount)</code></th>
<th><code>maxSplitSiz &lt;= size(urls[0]) - offset(scount)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scount</code></td>
<td><code>scount * maxSplitSize</code></td>
<td><code>maxSplitSize</code></td>
</tr>
<tr>
<td><code>i : \sum_{j=0}^{i-1} (\text{size(urls}[j])/\text{maxSplitSize}) &lt; scount</code></td>
<td><code>maxSplitSiz &gt; size(urls[i]) - offset(scount)</code></td>
<td><code>maxSplitSiz &lt;= size(urls[i]) - offset(scount)</code></td>
</tr>
<tr>
<td><code>\wedge scount \leq \sum_{j=0}^{i-1} (\text{size(urls}[j])/\text{maxSplitSize})</code></td>
<td><code>maxSplitSiz &gt; size(urls[i]) - offset(scount)</code></td>
<td><code>maxSplitSiz &lt;= size(urls[i]) - offset(scount)</code></td>
</tr>
<tr>
<td><code>\wedge scount \leq (\sum_{j=0}^{i}(\text{size(urls}[j])/\text{maxSplitSize}) \leq \text{maxSplitSize}</code></td>
<td><code>maxSplitSiz &gt; size(urls[i]) - offset(scount)</code></td>
<td><code>maxSplitSiz &lt;= size(urls[i]) - offset(scount)</code></td>
</tr>
<tr>
<td><code>\wedge scount \leq (\sum_{j=0}^{i}(\text{size(urls}[j])/\text{maxSplitSize}) \leq \text{maxSplitSize}</code></td>
<td><code>maxSplitSiz &gt; size(urls[i]) - offset(scount)</code></td>
<td><code>maxSplitSiz &lt;= size(urls[i]) - offset(scount)</code></td>
</tr>
<tr>
<td><code>\wedge scount \leq \text{maxSplitSize}</code></td>
<td><code>maxSplitSiz &gt; size(urls[i]) - offset(scount)</code></td>
<td><code>maxSplitSiz &lt;= size(urls[i]) - offset(scount)</code></td>
</tr>
<tr>
<td><code>\wedge scount \leq \text{maxSplitSize}</code></td>
<td><code>maxSplitSiz &gt; size(urls[i]) - offset(scount)</code></td>
<td><code>maxSplitSiz &lt;= size(urls[i]) - offset(scount)</code></td>
</tr>
<tr>
<td><code>\wedge scount \leq \text{maxSplitSize}</code></td>
<td><code>maxSplitSiz &gt; size(urls[i]) - offset(scount)</code></td>
<td><code>maxSplitSiz &lt;= size(urls[i]) - offset(scount)</code></td>
</tr>
</tbody>
</table>
Output Functions:

$$ds(T) =$$

<table>
<thead>
<tr>
<th>$(T=_)$ $\lor$ $(\text{PGM}(r(T))=\text{MFileStream})$</th>
<th>$(\text{PGM}(r(T))=\text{next'})$ $\lor$ $(\text{PGM}(r(T))=\text{previous'})$ $\lor$ $(\text{PGM}(r(T))=\text{refresh})$ $\lor$ $(0 \leq \text{scount'} &lt; \text{noStreams})$</th>
<th>$(\text{PGM}(r(T))=\text{next'})$ $\lor$ $(\text{PGM}(r(T))=\text{previous'})$ $\lor$ $(\text{PGM}(r(T))=\text{refresh})$ $\lor$ $(0 \leq \text{scount'} &lt; \text{noStreams})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg(T=_)$ $\land$</td>
<td>OpenStream(urls[index(i)]).bytes[offset(scount')]: offset(scount')+$\text{streamLen}(scount')$</td>
<td>Null</td>
</tr>
</tbody>
</table>

Inner function:

$$\text{ds} = \text{StreamFactory}.\text{getInputStream}(\text{strmComp},\text{ds}(\text{rlmessenger}(\text{StreamFactory}.\text{getOutputStream}(\text{strmComp}))))$$

$$\text{ue}(T) =$$

<table>
<thead>
<tr>
<th>$(T=_)$ $\lor$ $(\text{validConfig}(<em>\text{pa}(\text{in}(r(T))))\ \text{None})$ $\lor$ $(\text{validConfig}(</em>\text{pa}(\text{in}(r(T))))\ \text{CONFIG.ERR})$</th>
<th>None $\lor$ $\text{mexception}(_\text{message}(\text{rlmessage.ue}))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg(T=_)$ $\land$</td>
<td>$\neg\text{PGM}(r(T)&lt;\text{rlmessenger})=\text{ue}$ $\lor$ $\text{PGM}(r(T)=\text{next'})$ $\lor$ $\text{PGM}(r(T)=\text{previous'})$ $\lor$ $\text{PGM}(r(T)=\text{refresh})$ $\lor$ $\text{PGM}(r(T)&lt;\text{rlmessenger})=\text{ue}$</td>
</tr>
</tbody>
</table>

$$\text{progress}(T) =$$

<table>
<thead>
<tr>
<th>$(T=_)$</th>
<th>NewProgress(noStream) $\lor$ Progress.set(scount) $\lor$ Progress(p(T))</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg(T=_)$ $\land$</td>
<td>$\neg\text{PGM}(r(T)=\text{MFileStream})$ $\lor$ $(\text{PGM}(r(T))=\text{next'})$ $\lor$ $(\text{PGM}(r(T))=\text{previous'})$ $\lor$ $(\text{PGM}(r(T))=\text{refresh})$</td>
</tr>
</tbody>
</table>
A.3.5 Sync_Queue<Param>

Output Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>outObject</td>
<td>&lt;Param&gt;</td>
</tr>
<tr>
<td>Size</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>Idle</td>
<td>&lt;Synchronizer&gt;</td>
</tr>
<tr>
<td>Value</td>
<td></td>
</tr>
</tbody>
</table>

Input Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM</td>
<td>{Sync_Queue, put, get, Size, isEmpty}</td>
</tr>
</tbody>
</table>

Parameter:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxSize</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync_Queue</td>
<td>&lt;ID&gt;</td>
<td></td>
<td>(PGM:Sync_Queue)</td>
</tr>
<tr>
<td>Put</td>
<td>&lt;Param&gt;</td>
<td></td>
<td>(PGM:put, 'in, _size')</td>
</tr>
<tr>
<td>Get</td>
<td>&lt;Param&gt;</td>
<td></td>
<td>(PGM:get, Value’, _size’)</td>
</tr>
<tr>
<td>Size</td>
<td>&lt;int&gt;</td>
<td></td>
<td>(PGM:size, Value’)</td>
</tr>
<tr>
<td>isEmpty</td>
<td>&lt;boolean&gt;</td>
<td></td>
<td>(PGM:isEmpty, Value’)</td>
</tr>
</tbody>
</table>
**Auxiliary functions:**

noeffect(e) ≡ (PGM(e)=size)∨(PGM(e)=isEmpty)

qs(T1, T2)=

<table>
<thead>
<tr>
<th>T2=_</th>
<th>(T2_=_)∧noeffect(o(T2))</th>
<th>¬(T2=_)^PGM(o(T2))=’get^ isEmpty(T1) ¬isEmpty(T1)</th>
<th>¬PGM(o(T2))=put size(T1)&lt;MaxSize ¬(T1=_)^¬(PGM(o(T1))=put) ¬(size(T1))&lt;MaxSize</th>
<th>T1</th>
<th>qs(T1, s(T2)) qs(T1.o(T2), s(T2)) qs(s(T1), s(T2)) qs(T1.o(T2), s(T2)) qs(s(T1), s(T2)) qs(T1, s(T2))</th>
</tr>
</thead>
</table>

qstrip(T)=qs(_, T)

**idle(T) =**

(=Value(get'(T)))

| T= _ | ¬(T=_)^PGM(r(T))=’get isEmpty(p(T)) ¬isEmpty(p(T)) | ¬isEmpty(p(T)) | isEmpty(p(T)) |
|-----|------------------|------------------|------------------|-----|------------------|

Idle(p(T))

**size(T) =**

| T= _ | ¬(T=_)^PGM(r(T))=’get isEmpty(p(T)) ¬isEmpty(p(T)) | ¬isEmpty(p(T)) | isEmpty(p(T)) |
|-----|------------------|------------------|------------------|-----|------------------|

| T= _ | ¬(T=_)^PGM(r(T))=’get isEmpty(p(T)) ¬isEmpty(p(T)) | ¬isEmpty(p(T)) | isEmpty(p(T)) |
|-----|------------------|------------------|------------------|-----|------------------|

isEmpty(T) ⇔ size(T)=0
\[ \text{Value}(T) = \]

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T = _)</td>
<td>ID(this)</td>
</tr>
<tr>
<td>( \neg (T = _) \land )</td>
<td>outObject(T)</td>
</tr>
<tr>
<td>( \neg (T = _) \land )</td>
<td>Size(T)</td>
</tr>
<tr>
<td>( \neg (T = _) \land )</td>
<td>(size(T)=0)</td>
</tr>
<tr>
<td>( \neg (\text{PGM}(r(T)) = \text{get}) \lor ) ( \neg (\text{PGM}(r(T)) = \text{put}) )</td>
<td>(size(T)=0)</td>
</tr>
</tbody>
</table>
A.3.6 Synchronizer (EventChannel)

Output Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM pass’</td>
<td></td>
</tr>
<tr>
<td>shallWait</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>isWaiting</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>Value</td>
<td></td>
</tr>
</tbody>
</table>

Input Variables:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM</td>
<td>Synchronizer, set, ‘pass, shallWait, isWaiting</td>
</tr>
<tr>
<td>waitIf</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>Initial</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>Val</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>Name</td>
<td>&lt;String&gt;</td>
</tr>
<tr>
<td>timeout</td>
<td>&lt;Time_in_milliseconds&gt;</td>
</tr>
<tr>
<td>Thread</td>
<td>&lt;Thread&gt;</td>
</tr>
</tbody>
</table>

Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronizer</td>
<td>&lt;id&gt;</td>
<td></td>
<td>(PGM: Synchronizer, Value’)</td>
</tr>
<tr>
<td>Synchronizer</td>
<td>&lt;id&gt;</td>
<td>waitIf, initial</td>
<td>(PGM: Synchronizer, ‘waitIf, initial, Value’)</td>
</tr>
<tr>
<td>Synchronizer</td>
<td>&lt;id&gt;</td>
<td>waitIf, initial, name</td>
<td>(PGM: Synchronizer, ‘waitIf, initial, name, Value’)</td>
</tr>
<tr>
<td>Set</td>
<td>val</td>
<td></td>
<td>(PGM: set, ‘val, pass’)</td>
</tr>
<tr>
<td>Pass</td>
<td></td>
<td></td>
<td>(PGM:pass, pass’)</td>
</tr>
<tr>
<td>Pass</td>
<td></td>
<td>timeout</td>
<td>(PGM:pass, ‘timeout, pass’)</td>
</tr>
<tr>
<td>shallWait</td>
<td>&lt;bool&gt;</td>
<td></td>
<td>(PGM:shallWait, Value’)</td>
</tr>
<tr>
<td>isWaiting</td>
<td>&lt;bool&gt;</td>
<td></td>
<td>(PGM:isWaiting, Value’ )</td>
</tr>
</tbody>
</table>
Output Function Tables

| shallWait(T) = |  
|----------------|-----------------|
| \(T = _\)     |                 |
| \(\neg (T = _) \land \neg \text{Synchronizer} \land \neg \text{Card}(\text{in}(T) > 1) \land \neg \text{waitIf}(\text{In}(T)) = \text{initial}(\text{In}(T))\) | False |
| \(\text{PGM}(r(T)) = \text{set}\land \neg \text{In}(T) = \text{waitIf}\) | True |
| \(\neg \text{In}(T) = \text{waitIf}\) | False |
| \(\neg \text{isWaiting}(p(T)) \land \text{PGM}(r(T)) = \text{set}\land \neg \text{isWaiting}(p(T))\) | shallWait(p(T)) |

Card\(\text{in}(\text{T})\) = number of arguments in the parameter list

| pass'(T) = |  
|----------------|-----------------|
| \(T = _\)     |                 |
| \(\neg (T = _) \land \text{PGM}(r(T)) = \text{'pass} \land \neg \text{shallWait}(T)\) | \(\_ \) (BLOCK) |
| \(\neg \text{shallWait}(T)\) | pass' (REALSE) |
| \(\neg \text{In}(T) = \text{waitIf}\) | \(\text{isWaiting}(p(T))\) |
| \(\text{set}\land \neg \text{In}(T) = \text{waitIf}\) | pass' (REALSE) |
| \(\neg \text{isWaiting}(p(T))\) | \(\text{isWaiting}(p(T))\) |
| \(\neg \text{isWaiting}(p(T))\) | pass' (REALSE) |

BLOCK \(\equiv\) (The thread that called \text{'pass} program will be set to WAIT state)

REALSE \(\equiv\) (The thread that called \text{'pass} program will not be blocked if it is a new call or will be released from the wait state if it were in WAIT state.)

| pass(timeout)'(T) = |  
|----------------|-----------------|
| \(T = _\)     |                 |
| \(\neg (T = _) \land \text{PGM}(r(T)) = \text{'pass} \land \neg \text{shallWait}(T)\) | \(\_ \) (BLOCK) |
| \(\neg \text{shallWait}(T)\) | pass' (REALSE) |
| \(\text{elt}(r(T)) < \text{'pass} \land \text{elt}(r(T)) \geq \text{'pass}\) | \(\text{elt}(r(T)) < \text{'pass}\) |
| \(\text{elt}(r(T)) \geq \text{'pass}\) | pass' (REALSE) |
| \(\text{elt}(r(T)) = \text{'pass}\) | pass' (REALSE) |
| \(\text{elt}(r(T)) \neq \text{'pass}\) | pass' (REALSE) |
| \(\text{elt}(r(T)) \neq \text{'pass}\) | pass' (REALSE) |
| \(\text{EVT}(\text{elt}(\text{mrcall('pass(timeout),T}) \geq \text{'in(\text{mrcall('pass(timeout), T)}\) | pass' (REALSE) |
| \(\text{EVT}(\text{elt}(\text{mrcall('pass(timeout),T}) \geq \text{'in(\text{mrcall('pass(timeout), T)}\) | pass' (REALSE) |
| \(\neg \text{isWaiting}(p(T))\) | pass' (REALSE) |
| \(\neg \text{isWaiting}(p(T))\) | pass' (REALSE) |
| \(\neg \text{isWaiting}(p(T))\) | pass' (REALSE) |
| \(\neg \text{isWaiting}(p(T))\) | pass' (REALSE) |
| \(\neg \text{isWaiting}(p(T))\) | pass' (REALSE) |
isWaiting(T)=

<table>
<thead>
<tr>
<th>T= _</th>
<th>PGM(r(T))=Synchronizer</th>
<th>shallWait(T)</th>
<th>shallWait(T)</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬(T=_) ∧</td>
<td>PGM(r(T))=’pass’</td>
<td>shallWait(T)</td>
<td>¬shallWait(T)</td>
<td>True</td>
</tr>
<tr>
<td>¬(T=_) ∧</td>
<td>PGM(r(T))=set</td>
<td>In(r(T))=waitIf</td>
<td>¬(In(r(T))=waitIf)</td>
<td>False</td>
</tr>
<tr>
<td>¬((PGM(r(T))= ’pass’) ∨ (PGM(r(T))=set))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Value(T)=

<table>
<thead>
<tr>
<th>T= _</th>
<th>PGM(r(T))=Synchronizer</th>
<th>ID(this)</th>
<th>shallWait(T)</th>
<th>isWaiting(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬(T=_) ∧</td>
<td>PGM(r(T))=shallWait</td>
<td>shallWait(T)</td>
<td>isWaiting(T)</td>
<td></td>
</tr>
<tr>
<td>¬(T=_) ∧</td>
<td>PGM(r(T))=isWaiting</td>
<td>isWaiting(T)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬((PGM(r(T))= Synchronizer) ∨ (PGM(r(T))= shallWait) ∨ (PGM(r(T))= isWaiting))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A.4 MSS Composite Structure (Partial)

Figure A-1 shows a connection structure for a composite in MSS. Equation A-14-1 represents the composite structure in a connection matrix, whose equivalent connection graph is depicted in Figure A-2.
Figure A-1: Composite Structure for MSS (Partial)
Equation A-14-1: Connection Matrix for a Composite Structure in MSS
Legend:

- Component
- Events and input/output variables

Figure A-2: Connection Graph
A.5 Event-Flow-Equations and Event-Flow-Graph (EFG) of MSS

Given a connection matrix and a TFM description of each component, the outgoing events of each component may be given as a function of the incoming events in the form of Boolean difference equations. Equation A-14-2 shows the event-flow-equations for the MSS composite structure whereas Figure A-3 shows the equivalent flow-graph.

\[
\begin{align*}
  n_{n+1} &= j_n \\
  j_{c,n+1} &= j_n \\
  j_{d,n+1} &= j_{c,n} \\
  u_{n+1} &= j_{d,n} \quad \text{(in Design alternative 1, D1) or, } u_{n+1} = s_{d,c,n} \quad \text{(in Design alternative 2, D2)} \\
  d_{f,n+1} &= d_{f,n} \\
  d_{s,n+1} &= d_{f,n} \\
  d_{s,c,n+1} &= d_{f,n} \\
  c_{k,n+1} &= p_{e,b,n} \\
  r_{c,n+1} &= c_{l,n} \\
  n_{l,a,n+1} &= q_{c,n} n \quad \{q_c = 1 - p_c : \text{probability of instantiating a new executor}\} \\
  l_{a,n+1} &= n_{l,a,n} \\
  n_{e,n+1} &= n_{e,n} \quad \{n_{l,b}, n_{j,o}, n_{r,a}\} \\
  j_{o,n+1} &= n_{j,o,n} \\
  l_{b,n+1} &= n_{l,b,n} \\
  r_{l,n+1} &= n_{r,l,n} \\
  e_{n+1} &= r_{3,n} \lor p_{c,n} \quad \text{where } r_{3,n} = j_{o,n}.l_{b,n} \lor r_{l,n} \\
  i_{n+1} &= e_n \\
  x_{n+1} &= e_n \lor r_n \\
  \{\text{event x is generated after events e and u, which can come at any order for new jobs and if there is a repeat order on existing jobs}\} \\
  g_{x,n+1} &= i_n \lor \neg c_n.q_{f,l}.f_{b,p,n} \lor c_n.f_{b,p,n} = i_n \lor (\neg c_n.q_{f,l}.q_{l,b} \lor c_n).f_{b,p,n} \\
  \{\neg c_n \rightarrow \text{if a job is not cancelled at time n,}\} \\
  p_f \rightarrow \text{probability of task f for a single job x not completed } q_f = 1 - p_f \\
  \text{size(splitFile)}/\text{size(job)} \\
  q_{l,b} \rightarrow \text{probability of pfb (putting full buffer on lb) in lb completed.} \\
  q_{l,b} = \begin{cases} 
  \text{size(lb.buffer)}/\text{size(splitFile)}, & \text{for size(lb.buffer) < size(splitFile)} \\
  1, & \text{otherwise} 
\end{cases} \\
  x_{g,n+1} &= g_{x,n} \lor \neg e_{m,p}(J) \lor x_n.e_{m,p-1}(J).g_{x,n-1} \\
  e_{m,p+1}(J) &= n_{j,o,n} \lor (\text{count}(x_g, p(T_{<JO>})) = \text{count}(x, p(T_{<JO>})))^6 \\
\end{align*}
\]

1. \(^6\) count(e, T) (count the number of occurrence of event e in trace T) \((<\text{Event Descripotor}> x <\text{trace}> \rightarrow <\text{Integer}>))

\[
\begin{array}{|c|c|c|}
\hline
T = _ & r(T) = e & 0 \\
\hline
\neg(T = _) \land \neg (r(T) = e) & \text{count}(e, p(T)) + 1 \\
\hline
\end{array}
\]
\[ g_{n+1} = x_n \lor \neg c_n \lor p_n \lor q \lor p_{fb_n} \]
\[ p_{g_{n+1}} = g_n \lor \neg \text{LittleSpace}_n(LB) \]
\[ g_{eb_{n+1}} = d_n \lor \neg c_n \lor p_{fb_n} \]
\[ e_{eb_{n+1}} = g_n \lor \neg \text{emp}_n(LB, eb) \lor \neg c_n \lor p_{fb_n} \]
\[ \text{peb}_{n+1} = (c_n \lor q \lor p_{ld})a_n \]
\[ q_{ld} = \begin{cases} \frac{\text{size}(ld.buffer)}{\text{size}(lb.buffer)}, & \text{for } \text{size}(ld.buffer) < \text{size}(lb.buffer) \\ 1, & \text{otherwise} \end{cases} \]
\[ e_{bp_{n+1}} = \text{peb}_n \]
\[ g_{fb_{n+1}} = i_n \lor \text{peb}_n \]
\[ f_{eb_{n+1}} = g_{fb_{n+1}} \lor g_{fb_{n-1}} \lor p_{fb_{n-1}} \]
\[ \text{emp}_{n+1} = \text{nlb}_n \lor p(T_{<LB>}) \]
\[ f_{1,n+1} = \neg c_n \lor p_n \]
\[ f_{2,n+1} = f_{1,n} \lor p \lor d_{c,n} \]
\[ d_{n+1} = d_{c,n} \lor a_{n-1} \]
\[ t_{n+1} = d_n \]
\[ b_{n+1} = \text{fb}_n \lor \neg c_n \lor p_{ld} \]
\[ p_{n+1} = b_n \]
\[ p_{o_{n+1}} = q_{p_n}, \text{ where } q_j \text{ is the probability of completing the p output for a single job j.} \]
\[ a_{n+1} = \neg c_n \lor p_n \]
\[ s_{n+1} = s_n \lor t_n \]
\[ r_{n+1} = c_n \lor (x \lor xg \lor fdp \lor p) \]
\[ r_{1,n+1} = r_n \]
\[ r_{2,n+1} = c_n \]
\[ r_{3,n+1} = (pc_n \lor p_{o_n}) \]
\[ r_{j,n+1} = \text{Job.MaxInactiveLife} \lor (\text{elt}(\text{pc}_n) > \text{Job.MaxInactiveLife}) \]
\[ \text{UE}_{d,n+1} = \{ \text{elt}(\text{sd}_n, a_{n-1}) > \text{SLCL.TimeOUT} \} \lor \{ \text{elt}(q, f, d_{c,n-1}) > \text{LRCL.TimeOUT} \} \lor \{ \text{elt}(u_n, j_{n-1}) > \text{RLM.MaxWait} \} \]
\[ \text{UE}_{c,n+1} = \{ \text{sd} < c_n < f_2 \lor e_c : 'LRCL.UploadLimit(t(e_c)) < LRCL.UploadLimit(t(e_c))' \} \]
\[ \text{UE}_{m,n+1} = \{ M < 1 \text{MB} \} \]
\[ \text{UE}_{uo,n+1} = \{ \text{sd} = f_2 \lor u_{n+1} = j_{d,n} \} \]

\text{Design} is to fire event u after event sd, i.e. \( u_{n+1} = sd_{c,n} \).
Figure A-3: Event Flow Graph of MSS (Partial)
Missing Labels on Edges/Links:

\[ gp \rightarrow pg = \neg \text{LittleSpace}_n(LB) \]

\[ fbg \rightarrow pg = \text{LittleSpace}_{n-1}(LB), \text{ gp}_{n-1} = \neg \text{LittleSpace}_n(LB) \]

\[ fbp \rightarrow gp = \neg c_n, pf \]

\[ fbp \rightarrow gx = \neg c_n, qf \leftarrow qf \lor c_n \]

\[ fbp \rightarrow fbg = \neg \text{ emp}_n(LB, fb) \]

\[ gfb \rightarrow fbg = \text{ emp}_n(LB, fb) \]

\[ gx \rightarrow xg = \neg \text{ emp}_n(JO) \]

\[ x \rightarrow xg = g(n). \text{ emp}_n(JO) \]

\[ fbg \rightarrow fbp = \text{ full}_{n-1}(LB, fb), pf_{n-1} \]

\[ f_2 \rightarrow d_c = sd_c \lor \neg c_n, pf \]

\[ pf_{n-1} \rightarrow fbp = \text{ full}_{n}(LB, fb) \]

The event-flow-graph can be used to assess whether any required outputs can be obtained without being trapped into some undesired states due to occurrence of an undesired event. This may be obtained by finding the ‘transfer function’ between any two desired points in the flow-graph by traversing the graph according to the edge directions satisfying the given conditions. For instance, the transfer function between \( j \) and \( po \) can be obtained as:

\[ t(j \rightarrow po) = j. (n. (q_c.n.la.la.ne.n3.r3+p_e)e.u_n+j.c.j_d.u.e_n).i.gfb.x.xg.(gp.pg.f_1.f_c.f_2.d_c.d.(geb.ebgpf.fb)].{fbg.b.p}.{a.b.p}.{a.peb.gfb})^*.po \]

\[ u_n = j.n. (q_c.n.la.la.ne.n3.r3+p_e).e \]

\[ e_n = j.n. (q_c.n.la.la.ne.n3.r3+p_e).e \]

\[ t(j \rightarrow po) = j. (n. (q_c.n.la.la.ne.n3.r3+p_e)e.jc.j_d.u.e_jc.j_d.u.j.n.(q_c.n.la.la.ne.n3.r3+p_e).e).i.gfb.x.xg.(gp.pg.f_1.f_c.f_2.d_c.d.(geb.ebgpf.fb)].{fbg.b.p}.{a.b.p}.{a.peb.gfb})^*.po \]

\[ = j.n. (q_c.n.la.la.ne.n3.r3+p_e).j_c.j_d.u.e.i.gfb.x.xg.(gp.pg.f_1.f_c.f_2.d_c.d.(geb.ebgpf.fb)].{fbg.b.p}.{a.b.p}.{a.peb.gfb})^*.po \]

\[ t(j \rightarrow po) = j.n. (q_c.n.la.la.ne.n3.r3+p_e).j_c.j_d.u.e.i.gfb.x.xg.(gp.pg.f_1.f_c.f_2.d_c.d.(geb.ebgpf.fb)].{fbg.b.p}.{a.b.p}.{a.peb.gfb})^*.po \]

Equation A-1-3
Appendix B. Publications and Posters

In addition to the various at SQRL research seminars, the following publications and poster presentations are prepared for public domain, based on this research work.

Presentations and poster sessions


Draft papers ready for submission to relevant journals


Early draft papers being prepared for submission to relevant journals


Book

1. There is a plan to publish the thesis as a book with the 'Cambridge Scholars Publishing'