West Virginia University
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2002

Early component-based reliability assessment using UML based software models

William Black Smith V
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
Smith, William Black V, "Early component-based reliability assessment using UML based software models" (2002). *Graduate Theses, Dissertations, and Problem Reports*. 1291.
https://researchrepository.wvu.edu/etd/1291

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Early Component-Based Reliability Assessment using UML Based Software Models

William B. Smith V

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science in Electrical Engineering

Bojan Cukic, Ph.D., Chair
Roy S. Nutter Jr., Ph.D.
Hany H. Ammar, Ph.D.

Lane Department of
Computer Science and Electrical Engineering

Morgantown, West Virginia
2002

Keywords: component-based, software, engineering, reliability, prediction, UML, Rational Rose, early component-based reliability assessment, ECRA, architectural, Bayesian

ABSTRACT

Early Component-Based Reliability Assessment using UML Based Software Models

William B. Smith V

In the last decade, software has grown in complexity and size, while development timelines have diminished. As a result, component-based software engineering is becoming routine. Component-based software reliability assessment combines the architecture of the system with the reliability of the components to obtain the system reliability. This allows developers to produce a reliable system and testers to focus on the vulnerable areas.

This thesis discusses a tool developed to implement the methodology previously created for early reliability assessment of component-based systems. The tool, Early Component-based Reliability Assessment (ECRA), uses Rational Rose Unified Modeling Language (UML) diagrams to predict the reliability of component-based software. ECRA provides the user with an easy interface to annotate the UML diagrams and uses a Bayesian algorithm to predict the system reliability. This thesis presents the methodology of ECRA, the steps taken to develop it, and its applications.

Dedication

This paper is dedicated to my family and friends that have provided emotional support throughout the years. To my mother and father, who have always provided the best for me and have constantly encouraged me to do my best. To my sister who provides comic relief in my life. To my other "mom" who has always been willing to provide a helping hand. To my grandparents whose positive encouragement has provided light in the dark times. And last, but not least, to my fiancée, Alison who has had to endure the long nights and lonely dinners without me. I only hope that our future together will be twice as bright as our past.

Acknowledgments

I would like to acknowledge my graduate committee:

Dr. Bojan Cukic, Committee Chairperson
Dr. Roy Nutter
Dr. Hany Ammar

for their assistance and guidance throughout the development of my thesis. Additionally, I would like to thank the professors at West Virginia University for providing me with an education that has lead to a good start in life.

Table of Contents

Chapter 1: Introduction ... 1
  1.1 Introduction to Software Reliability Engineering ... 1
  1.2 Software Reliability Engineering Process ... 3
    1.2.1 Requirements ... 3
    1.2.2 Design ... 4
    1.2.3 Implementation ... 5
    1.2.4 Maintenance ... 5
  1.3 Contribution to Software Reliability Engineering ... 6
  1.4 Thesis Outline ... 7
Chapter 2: Related Work ... 8
  2.1 Software Reliability Growth Models ... 9
    2.1.1 Concave Models ... 10
    2.1.3 Software Reliability Growth Applications ... 11
  2.2 Component-Based Software Reliability Models ... 12
    2.2.1 State-Based Models ... 12
    2.2.2 Path-Based Models ... 14
    2.2.3 Additive Models ... 15
    2.2.4 Component-Based Reliability Applications ... 15
Chapter 3: ECRA Methodology ... 17
  3.1 Annotating Use Case Diagrams ... 18
  3.2 Annotating Sequence Diagrams ... 18
  3.3 Annotating Deployment Diagrams ... 19
  3.4 Combining Component and Connection Failures ... 19
Chapter 4 System Requirements for Tool ... 20
  4.1 Requirement Decisions and Tradeoffs ... 20
  4.2 Detailed Requirements ... 21
Chapter 5: Program Design ... 22
Chapter 6: Testing ... 23
Chapter 7: Summary and Future Work ... 25
References ... 26
Appendix A: User's Manual ... 28
Appendix B: Programmer's Manual ... 38
Curriculum Vitae ... 61

List of Figures

Figure 1 - Software Reliability Growth Model Types 9
Figure 2 - Annotated Sequence Diagram 18
Figure 3 - Overview of Process 22
Figure 4 - Rational Rose File Menu 30
Figure 5 - Export Character Set 31
Figure 6 - Export Completion 31
Figure 7 - ECRA Start Wizard Mode 31
Figure 8 - Opening Rational Rose XML File 32
Figure 9 - Saving Simulation Results 32
Figure 10 - Saving Simulation Settings 32
Figure 11 - Starting Simulation 33
Figure 12 - Viewing Results 33
Figure 13 - Histogram of Results 33
Figure 14 - Generating Script File 34
Figure 15 - Saving Script File 34
Figure 16 - Load Script File 35
Figure 17 - Loading Saved Results 36
Figure 18 - Overall Flow Diagram 42
Figure 19 - Use Case Flow Diagram 43
Figure 20 - Sequence Flow Diagram 44
Figure 21 - Deployment Flow Diagram 45
Figure 22 - Splash Screen 46
Figure 23 - Start Screen 46
Figure 24 - Use Case Actor Probability Screen 48
Figure 25 - Use Case Actor Connection Prob. Screen 49
Figure 26 - Use Case – Sequence Diagram Connection Screen 50
Figure 27 - Sequence Diagram – Deployment Diagram Connection Screen 51
Figure 28 - Deployment Diagram Connection Use Screen 52
Figure 29 - Deployment Diagram Connection Probability Screen 53
Figure 30 - Deployment Diagram Component Probability Screen 54
Figure 31 - Output Setting Screen 55
Figure 32 - Running Simulation Screen 55

List of Symbols, Abbreviations, or Nomenclature

1. CDG – Component Dependency Graph
2. COTS – Component off-the-shelf software
3. DD – Deployment Diagram
4. ECRA – Early Component-based Reliability Assessment
5. N.H.P.P – Non-Homogeneous Poison Process
6. PECT – Prediction-enabled Component Technology
7. SBRA – Scenario-Based Reliability Analysis
8. SD – Sequence Diagram
9. SRE – Software Reliability Engineering
10. SREPT – Software Reliability Estimation and Prediction Tool
11. UCD – Use Case Diagram
12. UML – Unified Modeling Language
13. XMI – Extensible Markup Interchange
14. XML – Extensible Markup Language

Chapter 1: Introduction

Software Reliability Engineering (SRE) started in the early 1970s thanks to pioneering works of Mora, Shoo, Cout, and many others [3]. Since its conception, the field of software reliability engineering has grown in importance. At the same time, so has the complexity and size of software projects.

In the last 30 years, software has gone from applications that require a large computer to applications that control the everyday conveniences that we know and love. As a result, we have come to depend on software to perform reliably and correctly. We expect the lights to turn on, the phone to ring, and email to be delivered. While it can be correctly argued that these items also depend on hardware to succeed, the fact remains that computer software has become a major source of outages in phone, communications, and email delivery in the last decade [4]. Software reliability engineering's goal is to prevent and/or reduce the chances of these outages from occurring. A short power failure due to a glitch in the software at the relay station may be a small inconvenience, but a glitch in the control system of a scram shutdown of a nuclear power plant or, as was the case with the Therac-25 radiation therapy machine [5], failure will result in loss of human life.

This chapter will provide an introduction into software reliability engineering. It will present a description of what software reliability engineering is, how it is performed, and the benefits of performing it. Additionally, this chapter will provide an insight into how the work and research presented here will contribute to the field of software reliability engineering. This chapter concludes with a short description into the content of the remaining chapters in this thesis.

1.1 Introduction to Software Reliability Engineering

Before we can define software reliability engineering, we first must define some key terms in the field. Software reliability is defined as "the probability of failure-free software operation for a specified period of time in a specified environment" [6]. Another term commonly used, ultrareliability, is used to refer to "failure intensities of less than 10^{-4} failures per execution hour" [7].

The definition of reliability and related ultrareliability, brings to question the difference between a failure and a fault. A failure is defined as "any departure of system behavior in execution from user needs," whereas a fault is defined as "a defect in system implementation that causes the failure when executed" [8]. To further elaborate, a software fault is "a defective, missing, or extra instruction or set of related instructions that is the cause of one or more actual or potential failures" [7]. This can be summarized to say that a failure is the result, and a fault is the cause.

When defining a failure and fault other key terms must be defined. Failures are grouped in two manners. The first is by the type of impact they have on the users. Failures with the same type of impact on the users are said to be part of the same failure category. The second manner is to group them by the magnitude the failure have to the users. Failures that exhibit the same degree of impact to the users are said to be part of the same failure severity class. This brings us to the final definition for failures, failure intensity, which is defined as "the number of failures per natural unit or unit of time" [7].

Faults provide the technical part of software reliability engineering with four key methods that assist in achieving reliability. The first is fault prevention, which is to avoid fault occurrences by construction. Fault removal is the act of detecting and eliminating faults with the help of verification and validation. Fault tolerance provides redundancy to handle faults without causing failures. Finally, fault forecasting estimates the presence of faults and the occurrences and consequences of the failure that the faults may cause through evaluation of the software [3].

The next set of terms that must be understood involve operational profiles and runs. First, an operation is defined as a "major system logical task of short duration, which returns control to system when complete" [7]. The operational profile is the set of operations and the probabilities of each operation occurring. This set of operations must have a total probability of occurrence that is equal to exactly one. A run, on the other hand, is the execution of a single operation and is characterized by an input state [7].

An input state is the complete set of all input variables for a single run and the possible values of each variable [7]. This should not be confused with the machine state, which is the set of all variables and their values that exist within the machine and may or may not be external to the particular program. The input variable itself is defined as a "variable that exists external to a run and affects its executions" [7]. As such, it can be seen that the input state is a subset of the machine state. The input states are also a subset of the input space, which contains the "set of all possible input states for a program" [7].

Through the previous definitions, we can now define what software reliability engineering is. It is "the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability" [9]. As such, software reliability engineering includes [3]:

1. Software reliability measurement, estimation, and prediction, which is accomplished through the various software reliability models discussed in the next chapter.
2. The attributes, metrics, and impact on reliability of product design, development process, system architecture, and software operational environment.
3. The application of this knowledge to assist in developing system software architecture definition, development, testing, use, and maintenance of the software.

This means that software reliability engineering enables both testers and developers to simultaneously [7]:

1. Ensure that product reliability meets the users' needs.
2. Speeds the product to market faster.
3. Reduces product development costs.
4. Improves customer satisfaction.
5. Increases productivity of testers and developers

In the next section, we will describe how software reliability engineering is performed. This section will describe in more detail the various aspects of software reliability engineering that is performed in the field.

1.2 Software Reliability Engineering Process

The process and cost of applying software reliability engineering varies from project to project. Depending on how well the company plans for and adapts to the new processes, the initial project may cost more money than it will save. As the company adapts and refines their application process, the benefits of software reliability engineering will become more evident. The specification of reliability requirements will allow "testers to concretely verify that the finished product meets the customers' needs before it is released" [3]. This verification will result in increased satisfaction of the customers since the product matches their needs more precisely than before. The expected knowledge of system use will also result in additional benefits. Developers will be able to coordinate resources to high-usage areas of the system. Additionally, testers will be able to use the expected system use and required reliability to control and direct testing to the sections of the system that need it. This will result in the product being delivered sooner and at a lower cost.

The remainder of this section will provide an overview of the software reliability engineering process. It will discuss how software reliability engineering is applied to the various phases of the waterfall model. Specifically, it will discuss software reliability engineering in the requirements/specifications, design, implementation, and maintenance phases of the waterfall model.

1.2.1 Requirements

In the early stages of development, the product is being considered and defined. Software reliability engineering activities in this phase vary from project to project. The first activity is to develop a functional profile. This profile defers from the operational profile that will be developed later. A functional profile includes the tasks performed by the product and the environment factors that the product must deal with (i.e. hardware, operating system, etc.). These tasks and factors are then assigned criticality values to weight their importance to the system.

Once the functional profile is completed, the next software reliability engineering activity is to define a failure with respect to the product and customer. Failures are commonly grouped by the amount of impact the failure will have on the product, user, and/or customer. These groups, called failure severity classes, are normally created with around three or four classes; however, more classes can be created. It will depend on the needs of the customer for the actual amount to be created. If the customer requires a fine resolution of failure effects on the user, more classes will need to be created, but at the cost of extra effort later in the life cycle.

The next step will be to initially define the customer's reliability needs. In some cases, it may be necessary to include both hardware and software reliabilities in their needs. Depending on the product, there are a number of options available. Some of the methods to generate reliability needs include analyzing a similar product from a competitor or an older version of the product to be developed. In defining the customer's needs, acceptable failure intensity should be assigned for each failure severity class. These values could range from 0.01 failures per hour to 10^{-6} failures per hour depending on the type of application to be developed. Once assigned, the reliability objective is then allocated between the hardware and software to produce the individual reliabilities.

With the functional profile and reliability information at hand, various trade-offs can be realized. These trade-offs can help minimize cost, maximize performance, and reduce the delivery date [10]. Thus the benefits of software reliability engineering can be realized at an early stage of development.

1.2.2 Design

In the design phase, the functional profile developed in the requirements is used to further guide the software reliability engineering process. Developers will convert the functional profile into an operational profile that will be used throughout the remainder of the software reliability engineering process. Development of the operational profile is perhaps the most expensive part of software reliability engineering process. Depending on the amount of detail placed in the functional profile, the goal will be to convert each function into one or more operations. In some cases, it may be necessary to create multiple operational profiles to accommodate different modes of operation.

Once the operational profile is completed, it can then be used to guide the development of the system architecture. The system architecture should be developed in a manner that will divide the system into multiple components. The next step will be to assign reliability values among the components. Using the reliability objectives defined in the requirements phase, the goal will be to define allocations that will satisfy the system reliability objective when allocations are combined.

With the operational profile, system architecture, and component reliabilities, developers and testers will be able to better allocate resources. Developers will be able to use the information generated to assign personnel and time lines to best accomplish development of the product. Testers will also use the information to guide test case generation. As

such, both the developers and testers will be able to move into implementation with a solid footing.

1.2.3 Implementation

In the implementation and maintenance phases, the software reliability engineering process varies the most. It is in these phases that most of the models presented in chapter two are applied. In general, the goal in implementation is to control the number of faults introduced into the system. This can be best accomplished by applying development practices that address fault management. These practices include [3]:

- Practicing a development methodology.
- Constructing modular systems.
- Employing reuse.
- Doing unit and integration testing
- Conducting inspections and reviews
- Controlling change

As implementation progresses, testers will be able to apply test cases as necessary. Testing data will allow developers to continue to remove faults and improve reliability of the product. Additionally, developers will be able to use the various models discussed in chapter two to assist in monitoring reliability of the product. Through the models and testing, developers will be provided with the necessary data to justifiably decide when to approve sections and when to rework sections of the product. By consistent monitoring of reliability testing, development of the product will move at an accelerated rate and the product will be released to the satisfaction of the customer.

1.2.4 Maintenance

As already stated in implementation, the software reliability engineering process in maintenance varies a great deal with a wide choice of software reliability models as discussed in chapter two. The primary goal of software reliability engineering process in maintenance is to monitor reliability in the field. This requires a method to track the failures and execution times across all sites. There are multiple approaches to gathering failure data. One method is to use a trouble tracking system. These systems are often database type applications that can be used to store information about the failures that have been encountered in the field and also the modifications users have requested. These systems assist in collecting failure data, but will never be able to gather 100% of the failures due to the fact that the all failures would have to be reported. A more costly approach is to interview users at various sites. This approach will assist in better understanding any dissatisfaction that the users may have with the product. A more thorough approach to failure data collection is to incorporate data collection tools within the product itself. The systems using the product would contact a remote system to report any failure data as it occurs. The data stored on the remote system can then be analyzed to gather the overall failure data.

Once the failure data is collected, the calculated field reliabilities can then be compared to the failure objectives of the system. If the case turns out to be one with significant differences between the field and test reliabilities, the reason may be one of many factors [3]:

- The definition of what the customer perceives as a failure is different from the definition used in testing the product.
- Inaccurate data collection during system test and/or field trial.
- The field and test operational profiles differ, the environment not accurately reflecting field conditions.

If it turns out that the operational profile is incorrect, it will be necessary to find the flaws in the profile before any further releases can be tested.

Additional releases can be accomplished by following the same approach already given throughout the life cycle. The functional and operational profile in most cases will still be applicable to the new release with only small changes required. The lessons learned in the maintenance phase can be applied to continually refine your company's software reliability engineering process and constantly increase the benefit software reliability engineering can present to projects.

1.3 Contribution to Software Reliability Engineering

The work presented in this thesis differs from most traditional software reliability models by the ability to apply it at an early stage of program development. The tool allows direct interaction with Rational Rose, a case tool often used in the early stages of development. Through the use of system architecture and operational profiles, it can estimate the reliability of the released system before code development even begins. Additionally, the tool could be used in cost and comparison exercises. It provides an easy to use method of comparing different components that perform the same task, but may have different reliabilities and costs. Thus, the impact of each component to the finished product can be assessed and a cost-based decision could be made.

On another side, the tool developed could also become applicable outside the field of software reliability engineering. If another field is able to model their system using the same types of software engineering diagrams, the system reliability calculations could be useful to their system. This is due to the fact that the tool provides a "diagram-based mixed with statistical methods" approach that may be applied to many fields of engineering as time progresses.

1.4 Thesis Outline

The remainder of this thesis is separated into six chapters. The next chapter, Chapter two discusses some of the other research options that have been investigated to address software reliability. These options include methodologies and models that relate to the many fields of software reliability engineering from component-based systems to reliability growth models. Chapter three continues by discussing the methodology of the approach we have taken. It covers the assumptions and calculations of the model that this thesis is based upon. The next three chapters, chapters four through six, respectively describe the requirements, design, and testing of the system we developed. These chapters present the process and implementation methodology behind the current version of our tool. Finally, Chapter seven concludes with a summary of the work we have done thus far and also discusses future work planned around our tool.

Chapter 2: Related Work

Software reliability engineering requires the use of specialized models and tools. These models take into account the principal factors that affect reliability fault introduction, fault removal, and system use [7]. Since each of these factors are "probabilistic in nature and operate over time" [7], software reliability engineering models are usually represented in the form of random processes. These random processes vary from model to model and, as such, are often used to classify the models into groups. Additionally, the definition of time may vary with the different models. Some models use calendar time as its input, which is not realistic unless the testing is constantly being performed. Other models define time as the number of tests that have been run. Again, this may not be realistic unless all the tests require the same amount of processing time. Finally, a majority of the models use execution time to represent the time parameter. Unlike calendar time and the number of tests, execution time best portraits the system being tested; unfortunately, it is the hardest to measure. A majority of all models, independent of the definition of time, assume that failures are independent of each other and these failures are often the "results of two processes: the introduction of faults and their activation through selection of input states" [7].

Since software reliability engineering practices vary from company to company and each project is unique in its goals, not all models available will be applicable to every project. The deciding factors include the input and output parameters of each good software reliability engineering model, along with the goals of the model. A good software reliability engineering model will have the following characteristics [7]:

1. gives good projections of future failure behavior
2. computes useful quantities
3. simple
4. widely applicable
5. based on sound assumptions

Additionally, the model must be able to handle program evolution and incomplete failure data, and be immune to hardware changes. Finally, the data collection for the model should be easily obtained and a program implementing the model is readily available.

The remainder of this chapter discusses two areas of software reliability engineering that are applicable in the field. The next section discusses software reliability growth models. It will give an overview of what they are and briefly discuss some of the models available. The remaining section concludes this chapter with a look into component-based software reliability engineering and the various models available in the component-based field.

2.1 Software Reliability Growth Models

Most easily applicable models in software reliability engineering are reliability growth models. These are mathematical models of system reliability change as testing and debugging occurs [11]. As such, software reliability growth models are often used as "an indication of the number of failures that may be encountered after the software has shipped and thus as an indication of whether the software is ready to ship" [12].

The process of using software reliability growth models is similar to normal development testing with some additional metrics gathered. The process begins with the testing of faulty software. During each phase of testing, a number of bugs are discovered. At the end of the phase, the amount of testing time and the number of failures encountered are recorded. The software is then repaired and the testing sequence starts over. The metrics gathered at the end of each testing phase is then processed into a reliability growth model. Depending on the model used, the results of the model will predict how many bugs will be discovered in the next round of testing and how long it will take to discover them. This information can then be used as an indicator of whether further testing will be cost efficient, and also the reliability of the current system.

Generally, software reliability growth models can be classified into two types: concave and s-shaped. This classification is determined by graphing the prediction results as shown in the figure below.

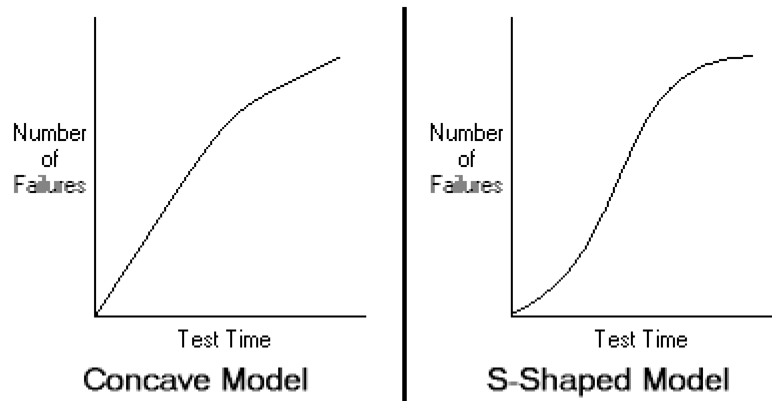

Figure 1 - Software Reliability Growth Model Types

Both concave and s-shaped models assume that as the number of failures detected and repaired increases, the failure detection rate decreases. This asymptotic behavior is based on the assumption that [12]:

1. A finite number of failures exist in a finite amount of code. In some cases, the repair of a failure may actually introduce an additional finite number of failures into the code. Depending on the model used, the new failures may be incorporated into the model or neglected all together.

2. The failure detection rate is proportional to the remaining failures in the code. As failures are repaired, fewer failures are left in the code. As such, the detection rate decreases and it requires more time to detect new failures.

2.1.1 Concave Models

Concave models are primarily defined by a defect detection rate that is proportional to the number of defects in the code, where this proportionality is assumed throughout the entire testing process [12]. These models have been used since the early years of software reliability engineering. Some of the more notable concave models include: Goel-Okumoto, Hossain-Dahiya/Goel-Okumoto, and Weibull models.

The Goel-Okumoto model was developed in 1979. Since its development, modifications of the Goel-Okumoto model have produced many other software reliability growth models. This model assumes that the defect detection rate per error is constant and the number of faults at the beginning of a test is a random variable [13]. The Goel-Okumoto model has two parameters: '*a*' and '*b*'. Parameter '*a*' represents the expected total number of defects in the code. Parameter '*b*' represents the shape factor, i.e. the rate at which the failure rate decreases. Through the combination of the two parameters the total expected number of defects at time *t* is: $u(t) = a(1 - e^{-bt})$ where $a \geq 0$ and $b > 0$.

Hossain-Dahiya/Goel-Okumoto model builds on the Goel-Okumoto model. This model makes the same assumption that the Goel-Okumoto model does with respect to the defect detection rate per error being constant; however, the Hossain-Dahiya/ Goel-Okumoto model assumes that there are no failures at time 0 [13]. Additionally, the model has three parameters: '*a*', '*b*', and '*c*'. Parameters '*a*' and '*b*' are the same parameters used in the Goel-Okumoto model. Parameter '*c*' is a control parameter that can be calculated using '*a*' and '*b*'. These parameters are combined to calculate the total expected number of defects at time *t* using: $u(t) = \frac{a(1 - e^{bt})}{(1 + c * e^{-bt})}$ where $a \geq 0$, $b > 0$, and $0 \leq c < 1$ [12]. This calculation is equivalent to the Goel-Okumoto model as '*c*' approaches zero.

The Weibull model is similar to the Goel-Okumoto model. This model states that even with an infinite amount of time, not all failures can be discovered [14]. The model will calculate the mean number of undetectable errors and is often used to predict and evaluate test effort. The model has three parameters: '*a*', '*b*', and '*c*'. Parameters '*a*' and '*b*' are the same parameters used in the Goel-Okumoto model and parameter '*c*' is a scale parameter. The Weibull calculation is then $u(t) = a(1 - e^{-x})$ where $x = -bt^{c}$, $a \geq 0$, $b > 0$, and $c > 0$. This calculation is equivalent to the Goel-Okumoto model when [12] $c=1$.

2.1.2 S-Shaped Models

S-Shaped models, unlike concave models, assume that the initial phase of testing is ineffective, but as testing progresses, the defect detection rate increases and the model starts to behave similar to the concave model for the remainder of testing. Two notable S-Shaped models are Gompertz [15] and Goel-Okumoto S-Shaped [12] models.

The Gompertz model assumes that "the testing/debugging effort is homogeneous throughout the entire test phase effort and that fixes are not accumulated in batches before being implemented" [15]. As such, a solid test plan and rapid repair of defects is required for use of this model. Use of the Gompertz model allows for modeling of reliability, mean time to failure, and cumulative failure count [15]. Calculation of the Gompertz model is expressed by: $R = AB^{C^{T_p}}$ where R is the software reliability as a function of Tp, which represents the test period, i.e. $Tp = 0, 1, 2, 3...$ etc. Parameter A, B, and C all represent Gompertz constants that will require additional calculations to compute where $A \geq 0$, $0 \leq B \leq 1$, and $0 < C < 1$. [15] suggests a spreadsheet approach to performing these calculations.

The Goel-Okumoto S-Shaped model is an extension of the concave Goel-Okumoto model presented in the previous section. This model makes the same assumption of bugs fixed when found as the Gompertz model. The calculation is somewhat simpler: $u(t) = a(1-(1+bt)*e^{-bt})$ where $a \geq 0$, $b > 0$. With this model, $u(t)$ represents the number of defects at time t, parameter a represents the expected total number of defects in the code, and parameter b represents the shape factor, which is the rate at which the failure rate decreases [16]. As a result of parameters 'a' and 'b', it may be necessary to have knowledge of previous development experience for an accurate fit.

2.1.3 Software Reliability Growth Applications

There are a number of applications available both professionally and academically that perform software reliability growth calculations. In some cases, these programs allow use of multiple software reliability growth models for possible calculations. Among the applications available are: CASRE and ReliaSoft's RG.

CASRE is built on another program called SMERFS, in which CASRE provides a graphical user interface to extend the usefulness of SMERFS. The tool, which is in its third version, estimates failure intensity from failure data. CASRE accepts the number of failures that occur during a certain amount of test time. It handles both execution and calendar time. CASRE will then apply the data to either a logarithmic or an exponential model to calculate the failure intensity of the software.

ReliaSoft's RG is a commercially available package designed for reliability growth analysis. The tool works with Microsoft Windows operating systems and uses Non-Homogeneous Poison Processes (N.H.P.P), Duane, Lloyd, Lipow, Gompertz, Modified Gompertz, and Logistic models to perform analysis calculations [17]. The tool provides Maximum Likelihood Estimation and Risk Regression methods to produce Reliability Growth Plots, MTBF Growth Plots, and Failure Rate Plots. RG provides multiple wizard interfaces to allow for easy creation of reliability growth analysis.

2.2 Component-Based Software Reliability Models

Component-based software reliability models are used to predict reliability of component-based systems. Component-based systems differ from traditional systems in the fact that the system is divided into separate logical units. These units may consist of commercial off-the-shelf programs or parts of previous programs being reused. The characteristics of component-based systems can include [18]:

- Systems that have significant aggregate functionality and complexity.
- Components are self-contained and possibly execute independently.
- Components will be used "as is" rather than modified.
- Components must be integrated with other components to achieve required system functionality.

Due to the unique architecture of component-based systems, the old methodology of black-box testing can be replaced with one that takes into consideration the individual components and structures of the system. Component-based software reliability models must then be able to predict reliability based on the reliability of each individual component and usage patterns in the given application.

One of the best guides to component-based software reliability models can be found in [19]. In this paper, the authors separate the various models in this field into three categories: state-based, path-based, and additive models. Each category has a wide range of models available. The next three sections will discuss each individual category. Additionally, a final section will be included to discuss some models that have been implemented.

2.2.1 State-Based Models

State-based models use control flow graphs, created through the use of Discrete Time Markov Chains, Continuous Time Markov Chains, or Semi-Markov Process to represent the architecture of the system [19]. These models assume that the future behavior of the software system is conditionally independent of the past behavior given the knowledge of the controlling module at any point in time. State-based models can be further classified as either composite or hierarchical. Composite state-based models use the given architectural model and failure behavior to predict reliability, while the hierarchical

approach solves for the architectural model and uses the solved model along with given failure behavior to predict reliability. Among some of the models classified in this area are the Littlewood model, the Laprie model, and the Gokhale et al. model [19].

The Littlewood model uses an irreducible semi-markov process to model the architecture of the component-based system. This model assumes that the software system consists of a finite number of modules and the transfer of control between modules can be specified by a probability p_{ij}. Additionally, the time spent in a module can be described as a general distribution function $F_{ij}(t)$ with a finite mean of m_{ij} [19]. The failure behavior in the Littlewood model consists of two types: failure during execution of the module represented as λ_i, and failure during transfer of control between two modules represented as v_{ij}. By combining this data, the failure rate according to the Littlewood model is [19]:

$$\lambda_s = \sum_i a_i * \lambda_i + \sum_{i,j} b_{ij} * V_{ij} \quad \text{where } a_i = \frac{\pi_i * \sum_j p_{ij} * m_{ij}}{\sum_i \pi_i * \sum_j p_{ij} * m_{ij}} \text{ and } b_{ij} = \frac{\pi_i * p_{ij}}{\sum_i \pi_i * \sum_j p_{ij} * m_{ij}}.$$

The Laprie model, like the Littlewood model, assumes the system is made up of n components. Transfer of control between the individual components is described through the use of a continuous time Markov chain. Additionally, the time spent within each component is represented by μ_i. The failure behavior in the Laprie model assumes the components fail with a constant failure rate λ_i. The Laprie model then assumes that the "failure rates are much smaller than execution rates" [19] resulting in the exchange of control occurring before the failure occurs. As a result, the system failure rate is modeled as: $\lambda_s = \sum_{i=1}^{n} \pi_i * \lambda_i$ where n is the number of components, π_i is the proportion of time spent in component i and λ_i is the failure rate for component i [19].

The Gokhale et. al. model is a hierarchical state-based model. It uses an absorbing discrete time Markov chain to describe the software system. To determine the transition probabilities p_{ij}, the model requires the use of a coverage analysis tool called ATAC [20]. The time spent in each component is "computed as a product of the expected execution time of each block and the number of blocks in the module" [19]. The failure behavior in the Gokhale et al. model is modeled through the use of an "enhanced non-homogeneous Poisson process using a time-dependent failure intensity $\lambda_i(t)$ determined by block coverage measurements" [19] acquired by ATAC. The overall reliability in the model is given by:

$R = \prod_{i=1}^{n} Ri$, where $R_i = e^{-\int_0^{Xi} \lambda_i(t)dt}$ and x_i represents the total expected time per execution spent in module i [19].

2.2.2 Path-Based Models

Path-based models use reliability with respect to the various execution paths that software can take. These models tend to require a completed program for testing to generate the required path information about the program. The path information is then combined with the failure behavior to predict reliability. Two notable path-based models are the Shooman model, and Scenario-Based Reliability Analysis model.

The Shooman model is one of the first models to estimate reliability of component-based systems using a path-based approach. This model assumes that the execution paths of the system are known along with the frequencies of occurrence for each path, denoted by f_i. The Shooman model characterizes failure behavior by calculating the failure probability on each path, denoted by q_i. This information is combined to produce the system probability of failure on any test run to be:

$q_o = \sum_{i=1}^{n} f_i * q_i$ where n is the number of components in the system [19].

The Scenario-Based Reliability Analysis (SBRA) model, unlike other path-based models, allows for reliability calculations in the early stages of development before an executable of the system is available. It uses an analysis technique that is based strictly on execution scenarios. The SBRA model can be used "to identify critical components and critical component interfaces and to investigate the sensitivity of the application reliability to changes in the reliabilities of component and their interfaces" [21]. The model begins with the creation of scenarios, which are sets "of component interactions triggered by specific input stimulus" [21]. To define the scenarios, SBRA model uses sequence diagrams similar to these used in the unified modeling language (UML). These diagrams provide the necessary information to calculate "the average execution time of a component in a scenario, the average execution time of a scenario, and possible interactions among components" [21]. Once the scenarios are completed, additional information about the probability of a scenario occurring, the individual component and transition reliabilities are used to create a "Component Dependency Graph (CDG)" [21]. The CDG is a modified control flow graph that has been adapted to apply to component-based applications. Creation of the component dependency graph is accomplished by following an outlined protocol that uses the information gathered from the previously defined scenarios. An analysis algorithm defined in [21] is then applied to the completed CDG to obtain the reliability analysis results.

2.2.3 Additive Models

Additive models, which could also be used for software reliability growth modeling, estimate reliability by combining the reliability data of the individual components. These models assume that the "components' reliability can be modeled by a non-homogeneous Poisson process, allowing for the system reliability to be expressed as the sum of its components' reliability" [19]. Additive models focus more on the failure data of the individual components than the architecture of the system. An additive model example is the Xie and Wohlin model.

The Xie and Wohlin model assumes that each component is a single system that is combined in series to produce the total system. Using the philosophy of a series circuit, it can then be said that failure of a component will result in failure of the entire system. Thus, the Xie and Wohlin model examines the individual component failure intensity, denoted by $\lambda_i(t)$, to obtain the system failure intensity at time t as:

$\lambda_s(t) = \sum_{i=1}^{n} \lambda_i(t)$ where n is the number of components in the system.

2.2.4 Component-Based Reliability Applications

There are two applications that are currently being developed that will provide an implementable solution to modeling reliability of component-based systems. Unlike the models discussed in the previous three sections, these applications can address reliability prediction at an earlier stage of development. These applications include: PECT [22] and SREPT [23], both of which are the result of projects in academia.

PECT, which stands for Prediction-Enabled Component Technology, is an application currently being developed as part of the PACC[2] project at the Software Engineering Institute [22]. The goal of PECT is to develop an add-on for another tool called ComTek. ComTek is a component design and modeling environment, in which users can create and execute component-based programs. PECT will interface with ComTek to obtain the architecture of the software system. It will then take a path-based approach to calculate the average contribution of each component to the system. At this point, PECT can perform one of two actions. Depending on the knowledge about the components and system reliabilities available, PECT can calculate system reliability or reliability of the individual components. Knowledge of the system reliability will allow PECT to calculate the individual component reliabilities by proportioning the system reliability with the average contribution of each component. On the other hand, knowledge of the individual components reliabilities will allow PECT to extrapolate the total system reliabilities by adding the proportional reliability of each component with respect to the average contribution of each component to the system.

SREPT [23], which stands for Software Reliability Estimation and Prediction Tool, is an application developed at Duke University. Its primary goal is to “track the quality of a software product during the software life-cycle, right from the architectural phase all the way up to the operational phase of the software” [23]. As such, SREPT is a software reliability growth tool, but its models are designed to account for component-based systems. SREPT uses a state-based approach and provides multiple techniques to model the software system. Its architecture modeling list includes using Discrete Time Markov Chains, Semi-Markov, Stochastic Petri Nets, and also Directed Acyclic graphs. SREPT executes its model of the system to calculate the time for and the average number of visits to each component. It then associates the architecture of the system with the user-defined failure probabilities for each component to produce the overall system reliability. Additionally, SREPT can provide release time information and, unlike most software reliability growth models, allows for reliability predictions to take into account finite fault removal times.

Chapter 3: ECRA Methodology

The Early Component-based Reliability Assessment (ECRA) tool was developed as an implementation of the methodology originally discussed in [1, 2]. The goals of the ECRA tool are as follows:

- The development of a probabilistic technique for reliability prediction that is applicable in the early phases of development, before an executable version of the system is available.
- The ability to study the impact of individual components and interfaces to the reliability of the application, thus allowing a quantifiable method in selecting components when alternative reusable assets are available to result in maximum system reliability.

The tool makes three assumptions. First, the tool assumes the existence of knowledge about failure rates for components. This information traditionally is not available in component libraries, but we speculate that over time, information about failure histories could become available in the form of specification sheets provided with the purchase of the component. Second, the methodology of the tool is simplified by the assumption of independence of failures among different components. To help realize this assumption, there are some proposals to build applications that include component wrappers to isolate each component [24]. The final assumption we made further simplifies our tool. We assume that component failure follows the principle of regularity, in which components are expected to fail at the same rate whenever the component is invoked.

To work with these goals and assumptions, the ECRA tool models the software system by annotating UML diagrams that can be created using Rational Rose from Rational, Inc. The ECRA tool uses three distinct diagrams available in UML: Use Case, Sequence, and Deployment Diagrams. Each diagram's annotation will be described in the following sections and this chapter will conclude with an explanation of how the annotations are combined to produce system reliability.

3.1 Annotating Use Case Diagrams

A use case diagram (UCD) describes system behavior (use cases) and provides a high level view of how the system interacts with external entities (actors). In our case, this diagram will be used to define the operational profile of the system. In annotating a UCD, we look at two parameters. The first is the probability that an actor will use the system denoted by q_i, where $\sum q_i = 1$ ($i = 1$ to the number of actors). The second parameter is the probability of an actor using a selected system behavior, denoted by P_{ix}. Again, $\sum P_{ix} = 1$ for actor q_i where x represents each actor/use case connection. This data is then combined to produce the probability of a system behavior occurring during execution of the software system. The probability of a system behavior x occurring is given by:

$$P(x) = \sum_{i=1}^{m} qi * Pix\,, \tag{1}$$

where m represents the number of actors that use the given system behavior. This value is used to predict the probability of a sequence diagram occurring.

3.2 Annotating Sequence Diagrams

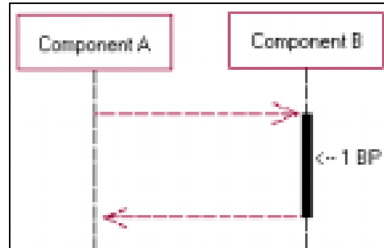

Figure 2 - Annotated Sequence Diagram

A sequence diagram (SD) depicts a time-based sequence of how groups of components interact to accomplish a given system behavior. There exists at least one SD per system behavior (use case). There are some instances in which more than one SD may exist for a single system behavior. In this case our methodology divides the probability $P(x)$ by the number of sequence diagrams used to represent the given system behavior.

Annotations of an SD in ECRA depict the amount of time each component is in a busy state or period. A busy period, shown in Figure 2, is defined as the interval of time that starts with an entering interaction and ends with the corresponding exit interaction [2]. We denote busy periods bp_{ij} as the number of busy periods that the component C_i shows in the Sequence Diagram j. By representing the failure probability for component C_i as θ_i, we can estimate the probability of component i in the scenario j failing as θ_{ij} using the following equation [2]:

$$\theta_{ij} = Prob(failure\ of\ C_{ij}) = 1 - (1- \theta_i)^{bpij} \tag{2}$$

3.3 Annotating Deployment Diagrams

A deployment diagram (DD) shows the physical configuration of the software application in terms of the various processors and connections that the application will be targeted to run. This diagram is the central point of the ECRA methodology. When combined with the information about the sequence diagrams, the system architecture can be defined. The DD allows annotations of both component and connection failure probabilities. These probabilities, denoted as θ_i and ψ_i respectfully, are represented by a mean failure probability and the 95% confidence interval of the failure probability to model the beta probability distribution of the component or connection. Our model includes information about connection reliability to take into account communication failures. To represent communication reliability between component *l* and *m*, we must first count the number of interactions that the two components exchange in the SD *j*, denoted as $|Interact(l,m,j)|$. Then we estimate the reliability ψ_{lmj} using the failure probability of the connection, denoted ψ_i as follows [1]:

$$\psi_{lmj} = (1-\theta_i)^{|Interact(l,m,j)|} \tag{3}$$

3.4 Combining Component and Connection Failures

Using the data from the UCD, SD, and DD, the ECRA tool can combine the information to produce an equation for the reliability of the whole system. By combining equations 1, 2, and 3, we obtain the following equation [1]:

$$\theta s = 1 - \sum_{j=1}^{k} P_j \left(\prod_{i=1}^{N} (1-\theta_i)^{bpij} * \prod_{(l,i)} (1-\psi_{lij})^{|Interact(l,i,j)|} \right) \tag{4}$$

This equation is used in the ECRA tool with the Bayesian reliability prediction algorithm, which uses random variables for θ_i and ψ_i to produce values for θs. By using multiple simulations, ECRA can produce a histogram of the results to represent the predicted system reliability. ECRA will also calculate the mean and 95% confidence interval of the simulation results to produce a beta curve for validation of the Bayesian model.

Chapter 4 System Requirements for Tool

4.1 Requirement Decisions and Tradeoffs

In the development of the requirements for the ECRA tool, described in section 4.2, various design decisions and alternative options presented themselves. These decisions and options included integration options to file formats. This section will focus on the benefits and alternative options that were available.

Decision: Interface with an existing tool for UML diagram creation or provide diagram creation within ECRA tool.
Results: Choose to interface with Rational Rose from Rational, Inc. Rational Rose is a well established UML design tool in the software engineering world. It provides the ability to create the necessary UML diagrams required to perform the reliability calculations. Also, by interfacing with Rational Rose, development time will be dramatically reduced.

Decision: Provide a method to specify UML annotations.
Results: Choose to use a dialog-type user interface. This allows for a smaller learning curve for the user and a reduced development time. The other option was to provide a method to graphically annotate the UML diagrams. The graphical option could not be completed within the development timeframe and may be useful in the future.

Decision: Provide seamless integration with a mathematical and graphical tool
Results: Choose to use MatLab from Mathworks as both the mathematical tool and graphical tool. MatLab provides easy integration into most major computer languages and has a large library of mathical and graphical functions. Another option available was Mathmatica, but the developer was not as familiar with that tool for it to be useful.

Decision: Provide a method to run script files containing the UML annotation data.
Results: Choose to use an XML-Based file format to use scripts. This choice does have the downfall of it not being user friendly, and a better option may have been to use Excel files. The excel files would have provided the user with a better interface, but would have required additional parsers. The decision to use the XML files does cost user friendliness, but provides but integration with the ECRA tool.

Decision: Provide a method to load and save simulation data.
Results: The choice to use XML-based files for running scripts also applied to loading and saving data. In this case, the use of xml files over a different format provides the ability for future integration with other tools.

4.2 Detailed Requirements

The ECRA tool provides a method to perform reliability assessment of a software system using Rational Rose diagrams and the methodology discussed in Chapter three. The requirements for the tool are separated into three sections: Rational Rose extraction, system inputs, and system outputs.

The ECRA tool need to interface with three types of diagrams in Rational Rose: use case, sequence, and deployment diagrams. The use case diagram requires abstraction of the names of each actor and use case, along with knowledge of each connection that may exist between the two. The sequence diagram involves the name of each diagram and modules within the diagram. Additionally, the tool is required to calculate the number of busy periods for each module in each diagram. Finally, the deployment diagram requires abstraction of the name of each processors and processes, along with knowledge of each connection between the individual processors.

The inputs into the ECRA tool consist primarily of the Rational Rose diagrams and user input. With respect to the use case diagram, the ECRA tool requires the user to specify the probability of an actor using the system, where the total probability of all actors using the system will equal exactly one. It will also require the user to assign probabilities to each of the connections an actor may have, where the total probability of all connections for a single actor equals exactly one. With respect to the sequence diagrams, the ECRA tool requires the user to link each sequence diagram to the use case procedure that it is supposed to represent. There is at least one sequence diagram for each use case procedure. Additionally, the user is required to specify the failure probability of confidence interval of each module in each diagram. Finally, with respect to the deployment diagram, the ECRA tool allows users to link processes to modules within each sequence diagram. There is at least one process for every module in the sequence diagrams. The tool also requires the user to specify the failure probability and confidence interval of each connection in the deployment diagram.

The outputs of the ECRA tool involve calculation results and graphs. The tool should calculate the parameters of prior beta distributions of each process and connector. This includes:

- θ for all components
- ψ for all connectors
- a_i and b_i for all components and connectors

The tool also produces a histogram plot of the calculation results, along with a file containing these results. Additionally, ECRA will produce a plot comparing the prior probability density function of the system failure probability θ_s and the normalized histogram from simulation observations. The ECRA tool also calculates the failure probability and the 95% confidence interval of the system.

Chapter 5: Program Design

The ECRA tool was developed using Visual Basic 6 programming language. Through the use of Visual Basic, generation of user interfaces, communication with Matlab, and parsing of XML code was made possible. The program consists of a main interface that will allow the user to load saved simulations, create and load script files, and start new simulations. The process to start a new simulation will work similar to a setup program, where the user will be guided through a series of steps and questions to produce the simulation results. The user will also be able to save the simulation settings for later modification and execution. The goal of this design is to produce a user friendly interface that requires minimal training to use.

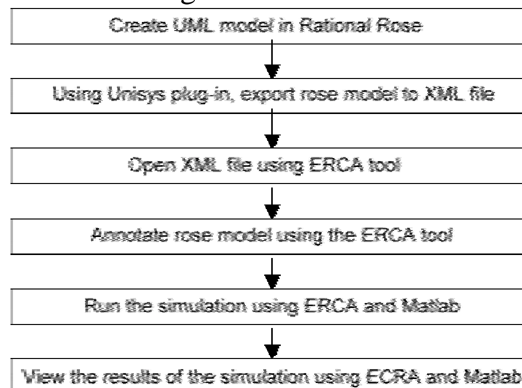

Figure 3 - Overview of Process

The overall process of running the simulation is shown in Figure 3. It begins with the user creating UML diagrams in Rational Rose. The user will create an overall use case diagram to represent the system to be developed. For each use case bubble, the user creates one or more sequence diagrams. The sequence diagrams represent the various actions that are performed during execution of a single use case bubble. Finally, the user creates a deployment diagram. The deployment diagram contains processors with processes and connectors connecting the processors. There will be one process for each unique node represented in the sequence diagrams. Once the user has completed and verified the diagrams, he or she uses the Unisys XML plug-in for Rational Rose to export the UML model to an XML file. Once the XML file is generated, the user can then open the ECRA tool to start the UML annotations. By selecting the XML file from Rational Rose, the user will be presented with a series of forms that will allow the user to annotate the UML diagrams. Once completed with the annotations, the user specifies the length of the simulation, and where to save the results. The ECRA tool will then open Matlab and run the simulation. Once the simulation has finished, ECRA will display two graphs of the results. The first graph will be a histogram showing the failure probability and 95% confidence interval of the system simulated. The second graph will display the curve of the histogram and the beta curve generated to represent the system. This graph will allow the user to judge the performance of the system. For more information on the design of the ECRA tool, please read Appendix B: Programmer's Manual.

Chapter 6: Testing

The ECRA tool was tested using the web application presented in [1]. The application was first modeled in Rational Rose, and then exported to an XML file. Testing of the ECRA tool was separated into three sections: functionality, robustness, and user-friendliness. Each section consisted of several tests that are described below.

To test the functionality of the ECRA tool required testing three branches. The first branch was to annotate the file using the series of questions. This branch was executed and using the information about the web application presented in [1], the UML model was annotated. The annotations were then saved to a file for later testing purposes. The second branch to test was creating and loading of script files. This test began with the creation of script file using the XML version of the Rose model. The created script file was then opened using a standard text editor. Next, the script file was filled in with the necessary information and saved. Once saved, the file was opened in the ECRA tool. The ECRA tool then allowed the user to verify the contents of the script file. The third and final branch was to load a saved simulation file. The file created in the first test was loaded and its contents were verified by the ECRA tool. At the conclusion of each of the three tests, the ECRA tool opened Matlab and ran the simulation. The simulation itself could take a great deal of time, depending on the number of trails to run and the processor speed and memory of the computer being run on. Once the simulation was completed, ECRA displayed the graphical results of the test.

To test the ECRA tool's robustness, various tests were used. The first involved the ECRA tool's ability to handle corrupt files. To perform this test, the XML file created in Rational Rose was edited in a standard text editor. Random sections of the file were removed. The edited file was then loaded into ECRA. The ECRA tool parsed the file and informed the user that the file was not correctly formatted. This test was also repeated for loading of script and simulation files. In each case, the ECRA tool informed the user that the file was not correctly formatted. The next test of robustness was to test invalid user input. This test consisted of testing each form where the user could enter text. In the section of ECRA involving annotation of the diagrams, the user would specify number values within a certain range. To test the error control of the input fields, each field was tested by entering a number within the range, a number on the upper and lower bound of the range, a negative and positive number out of the range, and a series of characters. The tests that involved a valid number which included entering a number within the range and on the upper and lower bound of the range were allowed to be entered. While with the tests that involved an invalid number, the ECRA tool informed the user that the number was invalid and defined valid input. The final test of robustness was to test the areas where the user could select information from a list. In this case, the test consisted of selecting the same item multiple times or selecting no items at all. Depending on which form the user was in during the test, the selection of matching items would only affect the simulation results, while selecting no items at all would either result in the user being required to select an item or the simulation ignoring the input.

The final test of the ECRA tool was to gain an initial understanding of the user-friendliness of the program. The test was performed using an inexperienced computer user. This user had no prior experience with the ECRA tool or software development in general. The hope was by using a user outside of the software development field, a worse case scenario could be realized about the ECRA tool. To assist the user in using the ECRA tool, the generation of the UML model and XML file in Rational Rose was presented to them up front. The user was then given a brief overview of the goal of the program and a sheet of paper with the answers to the series of questions. The user was then given free run of the program. The user was able to create the simulation and view the results. The user found that the entering of information was not difficult. Next, the user was told to create and edit a script file. Again, the user had the XML file of the rose model to start with and the sheet of paper with the answers to the series of questions. The user was immediately able to create the script file. To edit the file, the user required some instruction on how to open the file for editing. Once the file was open, the user then tried to work through the file and enter the information. After some time, the user was able to finish the file and run the simulation. The user found that using the script file method took more time, and liked the series of questions method much better. Overall, the user found using the ECRA tool took some time to understand, but was not overly difficult. Through the evaluation of this inexperienced user, it is believed that the ECRA tool will prove easier to user within a software development environment.

By testing the functionality, robustness, and user-friendliness of the ECRA tool, it was found to meet its specifications and perform correctly under the simulation conditions it was presented to. Future testing may prove useful with the use of a larger and more complicated UML models.

Chapter 7: Summary and Future Work

The ECRA tool discussed in this thesis has been successfully implemented and meets its requirements. The tool has already been demonstrated at two locations. Since its release, users of the tool have encountered few problems and have successfully produced reliability assessments of component-based systems. It is our hope that through the development of this tool and model, practitioners will be able to apply software reliability assessment at earlier stages of the life cycle model than currently available. Additionally, ECRA will hopefully assist practitioners in developing cost-effective reliable software system.

Through the use of Rational Rose and ECRA, a methodology to apply reliability assessment in the early stages of development can be realized. This early assessment will allow developers to predict and prevent problems before the problems escalate in cost and significance. Additionally, Rational Rose is a widely acceptable software development tool. As such, the incorporation of Rational Rose into the ECRA tool assists in an easy transition for developers in use the ECRA methodology. At the same rate, the interface between Rational Rose and ECRA is accomplished through the use of the XMI standard, which is to be applied to all UML development tools allowing for possible interfaces between ECRA and tools using the XMI standard.

The current version of the ECRA tool is compatible with Windows 98/ME, NT, 2000, and XP operating systems. This version requires MatLab 6.1+ and Rational Rose 2000 with the Unisys XMI plug-in for full capability. Future versions of ECRA will integrate MatLab into the program. Also, future versions of Rational Rose are to include the capability to export directly to XML. In time, the ECRA tool may even be integrated into Rational Rose.

We are currently searching for practical test cases to demonstrate the feasibility of the ECRA approach. This executable implementation of our model will allow us to further test and refine the reliability approach we have taken. It is our hope that the research we have completed will provide a methodology for future development in the component-based reliability field.

References

1. V. Cortellessa et al., "Early Reliability Assessment of UML Based Software Models", *Proc. International Workshop on Software Performance*, Rome, Italy, July 2002.
2. H. Singh et al., "A Bayesian Approach to Reliability Prediction and Assessment of Component Based Systems", *Proc. 12th International Symposium on Software Reliability Engineering* (ISSRE'01), Hong Kong, November 2001.
3. Lyu, M., *Handbook of Software Reliability Engineering*, McGraw-Hill, New York, NY, 1996
4. Gary, J., "A Census of Tandem System Availablity Between 1985 and 1990," *IEEE Transactions on Reliability*, vol. 39, no. 4, October 1990, pp. 409-418
5. Lee, I., and Iyer, R.K., "Analysis of Software Halts in Tandem System," *Proceedings of the 3rd International Symposium on Software Reliability Engineering*, October 1992, pp. 227-236
6. ANSI/IEEE, "Standard Glossary of Software Engineering Terminology," STD-729-1991, ANSI/IEEE, 1991
7. Musa, J., *Software Reliability Engineering*, McGraw-Hill, New York, NY, 1999
8. Musa, J., "More Reliable Software Faster and Cheaper: An Overview of Software Reliability Engineering", http://members.aol.com/JohnDMusa/ARTweb.htm, 9/14/2002
9. IEEE, Charter and Organization of the Software Reliability Engineering Committee, 1995
10. Musa, J.D., Iannino, A., and Okumoto, K., *Software Reliability --- Measurement, Prediction, Application*, McGraw-Hill, New York, NY, 1987
11. Sommerville, Ian., "Reliability growth modeling", http://www.comp.lancs.ac.uk/computing/users/tam/CS231/slides-18/sld033.htm, 09/25/2002
12. Wood, Alan, "Software Reliability Growth Models", http://www.hpl.hp.com/techreports/tandem/TR-96.1.html, 09/25/2002
13. Hossian, S., Dahiya, R., "Estimating the Parameters of A Non-homogeneous Poisson-Process Model for Software Reliability," IEEE Transactions on Reliability, Vol. 42, No. 4, December, 1993.
14. Yamada, S., Hishitani, J., and Osaki, S., "Software-Reliability Growth with a Weibull Test-Effort: A Model & Application," IEEE Transactions on Reliability, Vol. 42, No. 4, December, 1993, pp. 100-106
15. Davis, Jack, "An Applied Spreadsheet Approach to Estimating Software Reliability Growth Using the Gompertz Growth Model", http://smaplab.ri.uah/ice/davis.pdf, 09/25/2002
16. Yamada, S., Ohba, J., and Osaki, S., "S-Shaped Reliability Growth Modeling for Software Error Detection," IEEE Transactions on Reliability, Vol. R-32, No. 4, December, 1983, pp. 475-484
17. ReliaSoft's RG, http://www.reliasoft.com/RG/RGOver.htm, 09/24/2002

18. Brown, A., Wallnau, K., “Engineering of Component-Based Systems,” *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, IEEE Computer Society, 1996, pp. 7-15
19. Goseva-Popstojanova, K., Trivedi, K.S., “Architecture Based Approaches to Software Reliability Prediction”, *International Journal Computers & Mathematics with Applications*
20. Horgan, J.R., London, S., “ATAC: A data flow coverage testing tool for C,” *Proc. 2nd Symp. Assessment of Quality Software Development Tools*, 1992, pp. 2-10
21. Yacoub, S., Cukic, B., and Ammar, H., “Scenario-Based Reliability Analysis of Component-Based Software,” *Proc. 10th Int’l Symp. Software Reliability Engineering (ISSRE’99)*, 1999, pp. 22-31
22. Stafford, J., McGregor, J. D., “Issues in Predicting the Reliability of Composed Components”, *Proc. 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, May 2002
23. Ramani, S. et al., “SREPT: Software Reliability Estimation and Prediction Tool”, Performance Evaluation Journal, Special issue on tools for performance evaluation, Elsevier Science 1999
24. J. M. Voas, “COTS and High Assurance: An Oxymoron?”, *Proc. Of the 4th IEEE International Symposium on High-Assurance System Engineering*, Bethesda, MD, 1998.

Appendix A: User's Manual

Table of Contents:

A.1.0 System Requirements and Limitations 29
A.2.0 Exporting UML Diagrams in Rational Rose 30
A.3.0 Running Simulations in ERA 31
A.3.1 Using Wizard Mode 31
A.3.2 Using Advanced Mode 34
A.3.3 Additional Features 36
A.3.3.1 Loading Saved Results 36
A.3.3.2 Exiting ECRA program 36
A.4.0 Troubleshooting 37

Table of Figures:

Figure 4 - Rational Rose File Menu 30
Figure 5 - Export Character Set 31
Figure 6 - Export Completion 31
Figure 7 - ECRA Start Wizard Mode 31
Figure 8 - Opening Rational Rose XML File 32
Figure 9 - Saving Simulation Results 32
Figure 10 - Saving Simulation Settings 32
Figure 11 - Starting Simulation 33
Figure 12 - Viewing Results 33
Figure 13 - Histogram of Results 33
Figure 14 - Generating Script File 34
Figure 15 - Saving Script File 34
Figure 16 - Load Script File 35
Figure 17 - Loading Saved Results 36

A.1.0 System Requirements and Limitations

The ECRA program requires the following minimum system features:

- MatLab 6.1.0.450
- Rational Rose Professional 2001 (version 7.5.0103.1920)
  Unisys Rose XML Tool (Plug-in for Rational Rose)
- Windows 98/ME, NT, 2000 or XP-based System
  1024x768 Screen Size
  Pentium II 400 MHz
  256 MB RAM
  20 MB of free hard drive space
  Internet Explorer 6.0

The current version of ECRA has the following limitations in the Rational Rose Diagrams:

- Maximum of one Use Case Diagram
  - Maximum of 21 Actors
  - Maximum of 21 Use Case Bubbles
- Maximum of one Deployment Diagram
  - Maximum of 15 Processors
  - Maximum of 15 Connections
- Maximum of 16 Sequence Diagrams
  - Maximum of 15 Nodes per Sequence Diagram

A.2.0 Exporting UML Diagrams in Rational Rose

The following screen shots were taken from Rational Rose Enterprise Edition 2001 from Rational Software Corporation. To export your UML diagrams perform the steps below.

1. Create Use Case, Sequence, and Deployment Diagrams in Rational Rose.
2. Upon completion of model, Choose File → Export UML 1.1 to XMI as shown in figure 3.

Figure 4 - Rational Rose File Menu

3. Choose ASCII/MBCS as the Character Set as shown in figure 4 and click Ok.

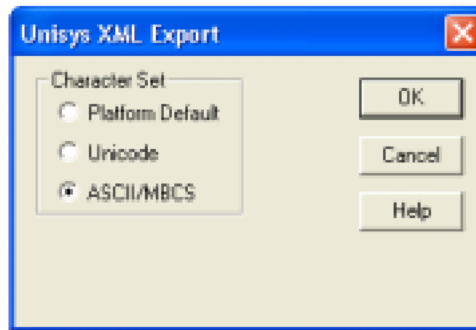

Figure 5 - Export Character Set

4. Upon completion, the screen shown in figure 5 will appear. A XML file will now exist in the same directory as your model file with the name: *model_name*.xml where *model_name* is the name of your UML model.

Figure 6 - Export Completion

A.3.0 Running Simulations in ERA

The ECRA tool provides two methods to perform annotation of the UML diagrams. The first method is the Wizard Mode. This mode is the default method of data entry, and is described in section A.3.1. The second mode is the Advanced Mode. This mode uses an ECRA generated XML file to allow saving and easy changing of values. The Advanced Mode is described in section A.3.2.

A.3.1 Using Wizard Mode

1. To begin data entry, click on the setup simulation button as shown in figure 6.

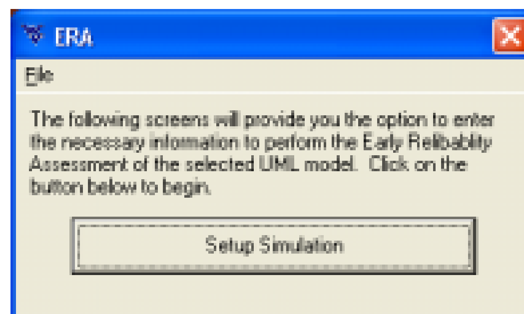

Figure 7 - ECRA Start Wizard Mode

2. Choose the XML file created by Rational Rose as shown in figure 7.

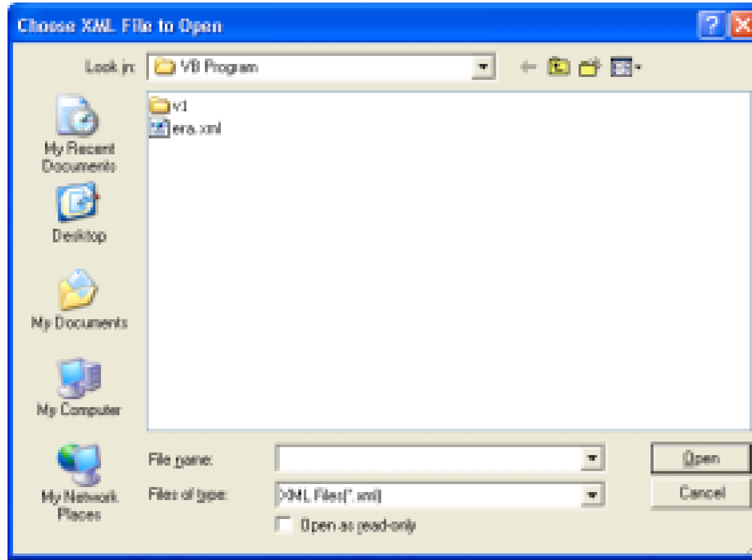

Figure 8 - Opening Rational Rose XML File

3. Complete each section, and click the next button to continue to the next section.
4. After completion of data entry the screen shown in figure 8 will appear. Enter the file name or select a location to save the results by clicking on the … button. Enter the number of trials to perform during the Matlab simulation. Click next to continue.

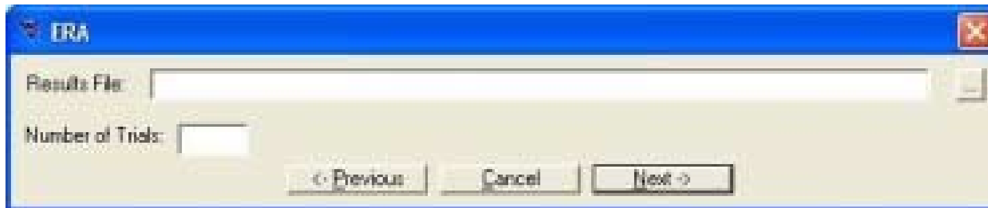

Figure 9 - Saving Simulation Results

5. After clicking the next button as shown in figure 9, you will be given the chance to save your simulation settings to an xml file. This will allow rerunning simulations by using the Advance Mode described in section A.3.2.

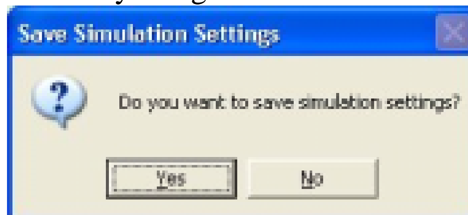

Figure 10 - Saving Simulation Settings

After saving simulation settings, click on the Start Simulation button shown in figure 10 to begin the simulation.

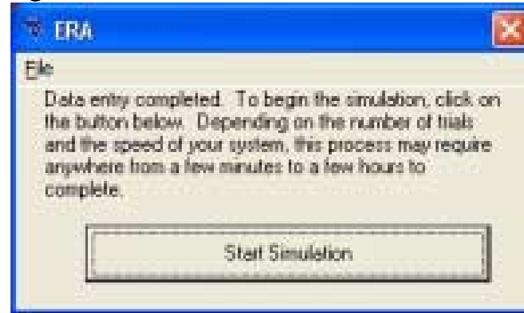

Figure 11 - Starting Simulation

6. Upon completion of the simulation, click the next button. Then click the view results button shown in figure 11 to view a histogram of the results.

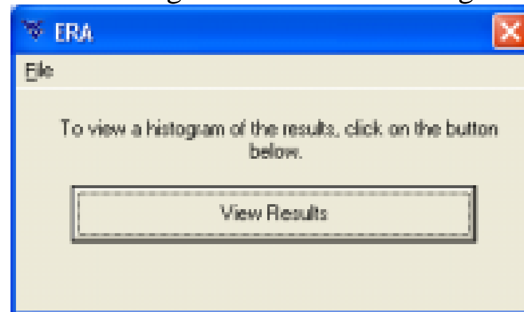

Figure 12 - Viewing Results

7. This will open a new window displaying the histogram of results along with a comparison of the beta curve and simulation results, as shown in figure 12.

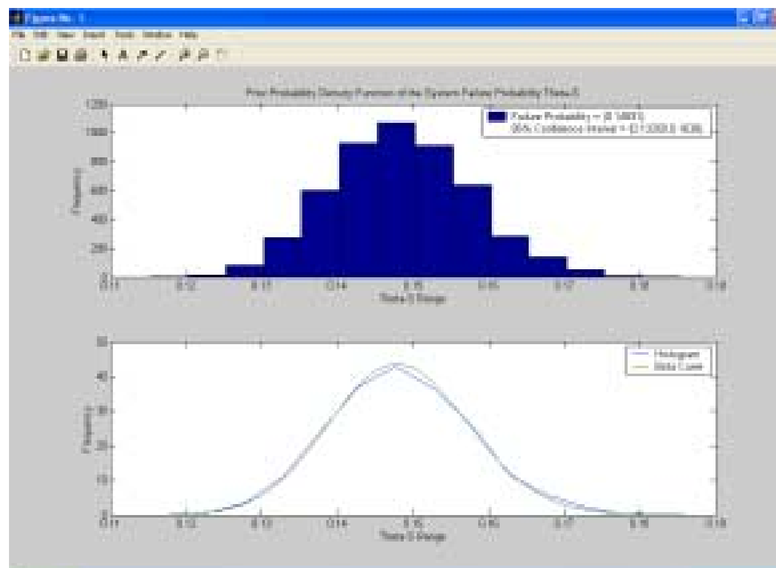

Figure 13 - Histogram of Results

A.3.2 Using Advanced Mode

1. Select File→Generate Script File as shown in figure 13.

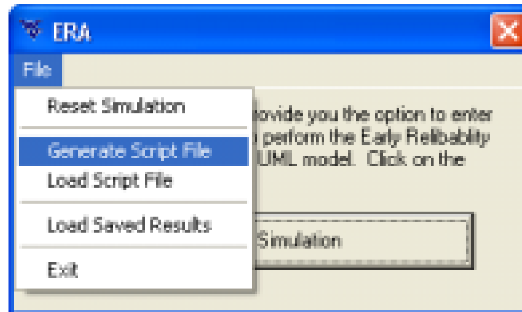

Figure 14 - Generating Script File

2. Choose the Rational Rose XML file to open as shown in figure 7 on page 19.
3. Choose where to save the script file as shown in figure 14.

Figure 15 - Saving Script File

4. Open the xml file created by the ECRA program using any text editor such as Notepad, or WordPad.

5. Fill in the missing data where requested.
   - For the use case diagram section, enter data where this line is: <Probability note = "Enter value between 0 and 1">###</Probability>
   - For the deployment diagram section, enter data where these lines are:

<FailureProbability note = "Enter value between 0 and 1">###</FailureProbability>

<ConfidenceInterval_Low note = "Enter value between 0 and 1">###
</ConfidenceInterval_Low>

<ConfidenceInterval_Hi note = "Enter value between 0 and 1">###
</ConfidenceInterval_Hi>

   - For the sequence diagram section, enter the processorID and processID for each process in each diagram. The processorID and processed can be found by view the deployment diagram section.

     Ex. The following process C1 has a processorID of 0 and processed of 0
     <Processor ID="0" Name="Client">
     <Process ID="0" Name="C1">

     These values are entered in the following lines:

<ProcessorID note = "Enter value corresponding to Processor id=?">###</ProcessorID>
<ProcessID note = "Enter value corresponding to Process id=?">###</ProcessID>

     In the case that the process in the sequence diagram does NOT correspond with a process in the deployment diagram, use -1 for both the processorID and processID.

     In addition to the processorID and processID, enter the number of times each connection is used in each diagram by placing the value in this line:

<ConnectionUse note = "Enter the number of times this connection is used">###
</ConnectionUse>

6. Save file. Open in Internet Explorer to easily navigate file.
7. Open the ECRA program.
8. Choose File->Load Script File as shown in figure 15.

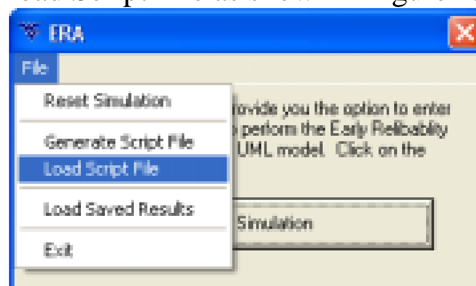

Figure 16 - Load Script File

9. Continue with steps 3 – 8 in section A.3.1 to finish the simulation.

A.3.3 Additional Features

A.3.3.1 Loading Saved Results

To load a previously run simulation, choose File→Load Saved Results as shown in figure 16. Select the *.dat file to open and follow step 8 in section A.3.1.

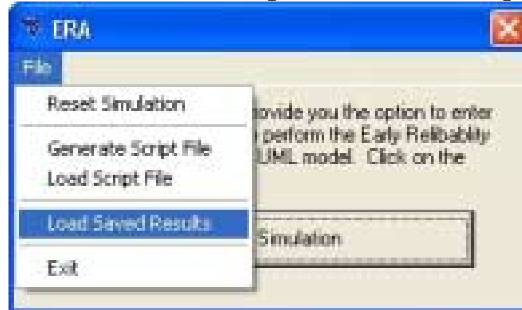

Figure 17 - Loading Saved Results

A.3.3.2 Exiting ECRA program

To exit the ECRA program, click the cancel button during data entry or click the File→Exit button when not in a data entry screen.

A.4.0 Troubleshooting

Problem: Unable to Export UML model in Rational Rose.
Solution: Verify Unisys Rose XML Tool is installed.

Problem: ECRA program does not load.
Solution: Verify Matlab 6.0 is installed and resolution is at least 1024x768

Problem: Rational Rose XML file does not open in ECRA.
Solution: Verify that UML diagrams do not exceed limitations discussed in section 1.0. Export XML file again.

Problem: Unable to load script file in ECRA.
Solution: Verify XML file is correctly formatted by loading in Internet Explorer. If unable to fix formatting, regenerate the file.

Problem: Click on View Results button and nothing happens.
Solution: Look on the taskbar for a window called Figure 1. Click on that window.

Appendix B: Programmer's Manual

Table of Contents:

B.1.0 System Requirements 40
  B.1.1 Production Requirements 40
  B.1.2 Development Requirements 40
B.2.0 Code Overview 41
  B.2.1 Forms 46
    B.2.1.1 frmSplash.frm 46
    B.2.1.2 frmChecklist.frm 46
    B.2.1.3 frmUCD.frm 48
    B.2.1.4 frmUCDActorProb.frm 48
    B.2.1.5 frmUCDActorConn.frm 49
    B.2.1.6 frmSDProc.frm 50
    B.2.1.7 frmSDUCconnection.frm 50
    B.2.1.8 frmSDProc.frm 51
    B.2.1.9 frmSDProcProc.frm 52
    B.2.1.10 frmDD.frm 53
    B.2.1.11 frmDDConnections.frm 53
    B.2.1.12 frmDDComponents.frm 54
    B.2.1.13 frmOutputSettings.frm 55
    B.2.1.14 frmRunSim.frm 55
  B.2.2 Generic Modules 56
  B.2.3 Class Modules 56
    B.2.3.1 ParseFile 57
    B.2.3.2 ReadFile 58
    B.2.3.3 RunSimulation 58
    B.2.3.4 WriteFile 59
  B.2.4 Matlab files 60

Table of Figures:

Figure 22 - Splash Screen.. 46
Figure 23 - Start Screen.. 46
Figure 24 - Use Case Actor Probability Screen 48
Figure 25 - Use Case Actor Connection Prob. Screen 49
Figure 26 - Use Case – Sequence Diagram Connection Screen 50
Figure 27 - Sequence Diagram – Deployment Diagram Connection Screen.................... 51
Figure 28 - Deployment Diagram Connection Use Screen... 52
Figure 29 - Deployment Diagram Connection Probability Screen 53
Figure 30 - Deployment Diagram Component Probability Screen................................... 54
Figure 31 - Output Setting Screen.. 55
Figure 32 - Running Simulation Screen... 55

B.1.0 System Requirements

B.1.1 Production Requirements

- MatLab 6.1.0.450
- Rational Rose Professional 2001 (version 7.5.0103.1920)
  Unisys Rose XML Tool (Plug-in for Rational Rose)

- Windows 98/ME, NT, 2000 or XP-based System
  1024x768 Screen Size
  Pentium II 400 MHz
  256 MB RAM
  20 MB of free hard drive space
  Internet Explorer 6.0

B.1.2 Development Requirements

- Above Production Requirements plus
- Microsoft Visual Basic 6.0 (SP5)
  References:
    - Visual Basic for Applications
    - Visual Basic runtime objects and procedures
    - Visual Basic objects and procedures
    - OLE Automation
    - Microsoft XML, version 2.0
    - Matlab Automation Server Type Library

  Controls:
    - Microsoft Windows Common Controls 5.0 (SP2)

B.2.0 Code Overview

The ECRA project consists of four types of files: forms, generic modules, class modules, and Matlab files. The section provides a description of these four types of files. The following diagrams (17 – 20) explain the flow of control throughout the ECRA tool.

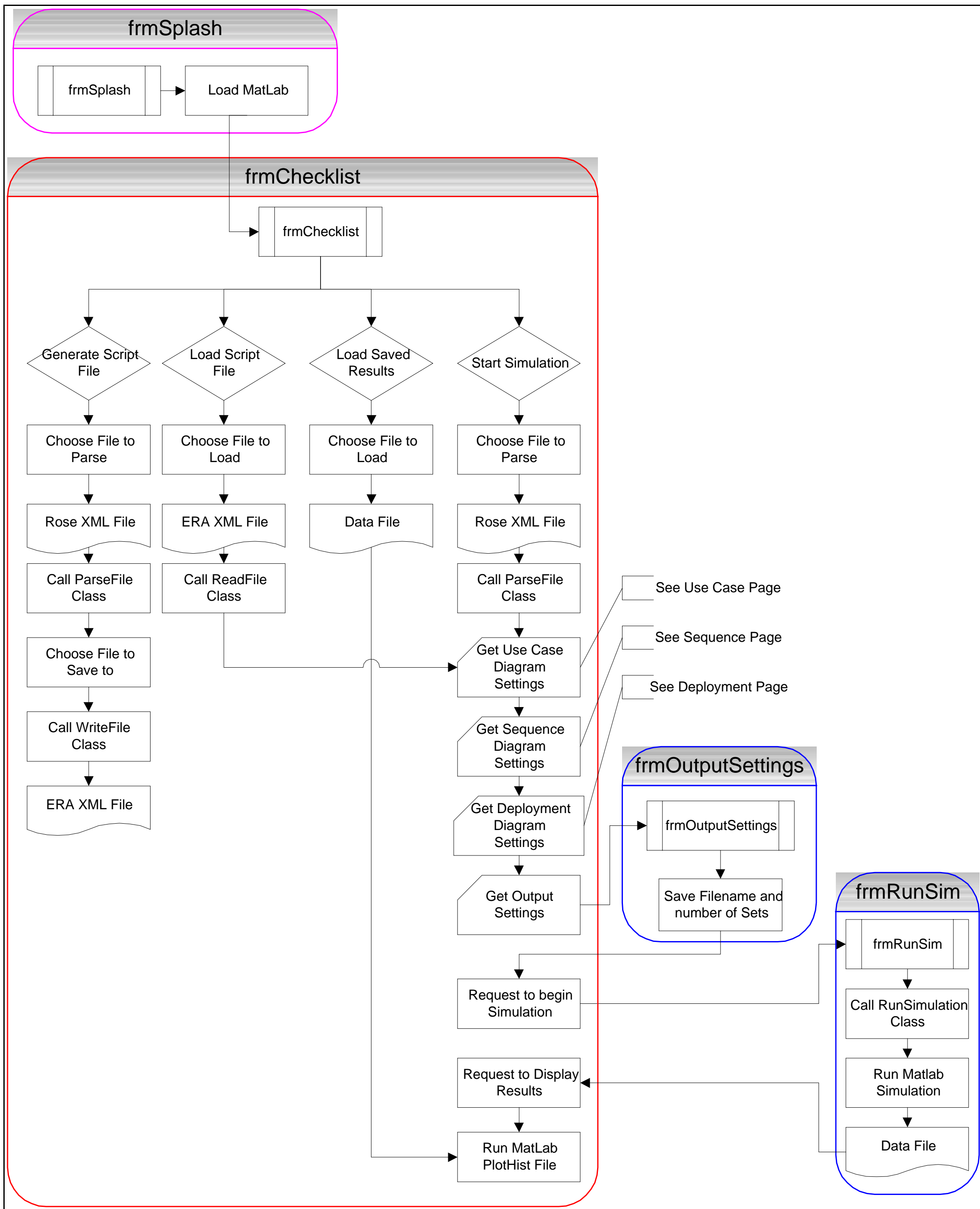

Figure 18 - Overall Flow Diagram

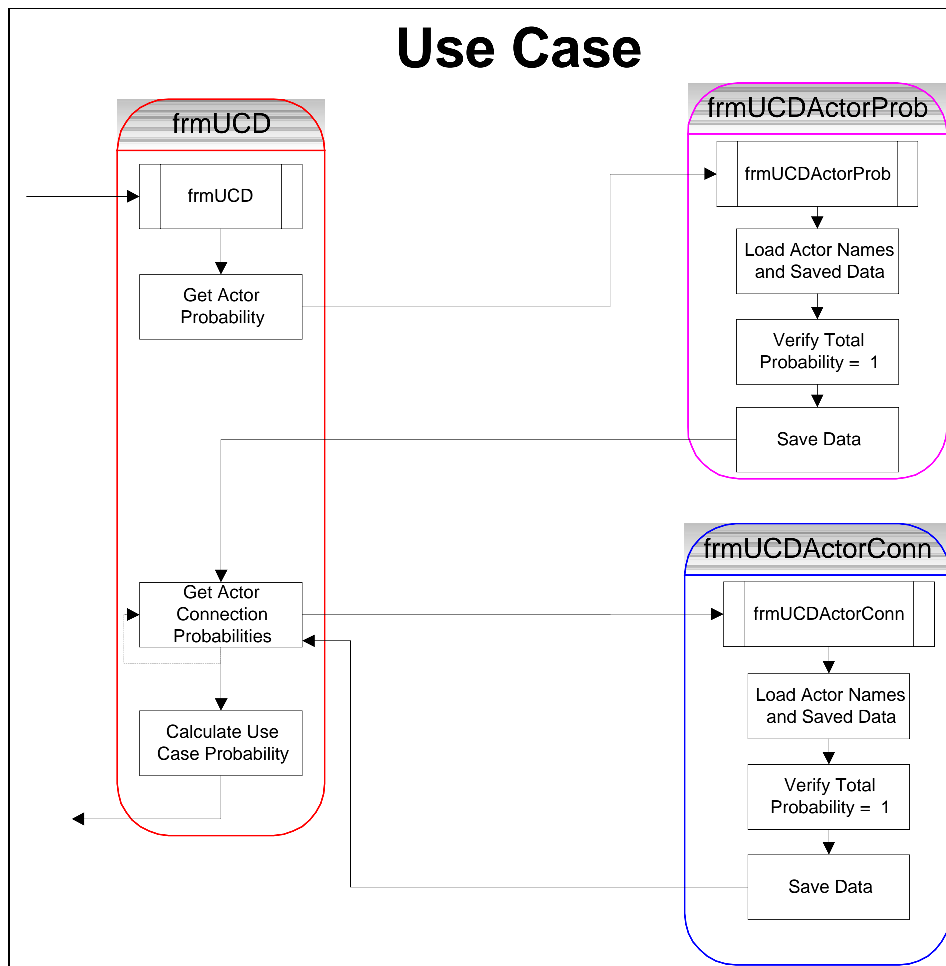

Figure 19 - Use Case Flow Diagram

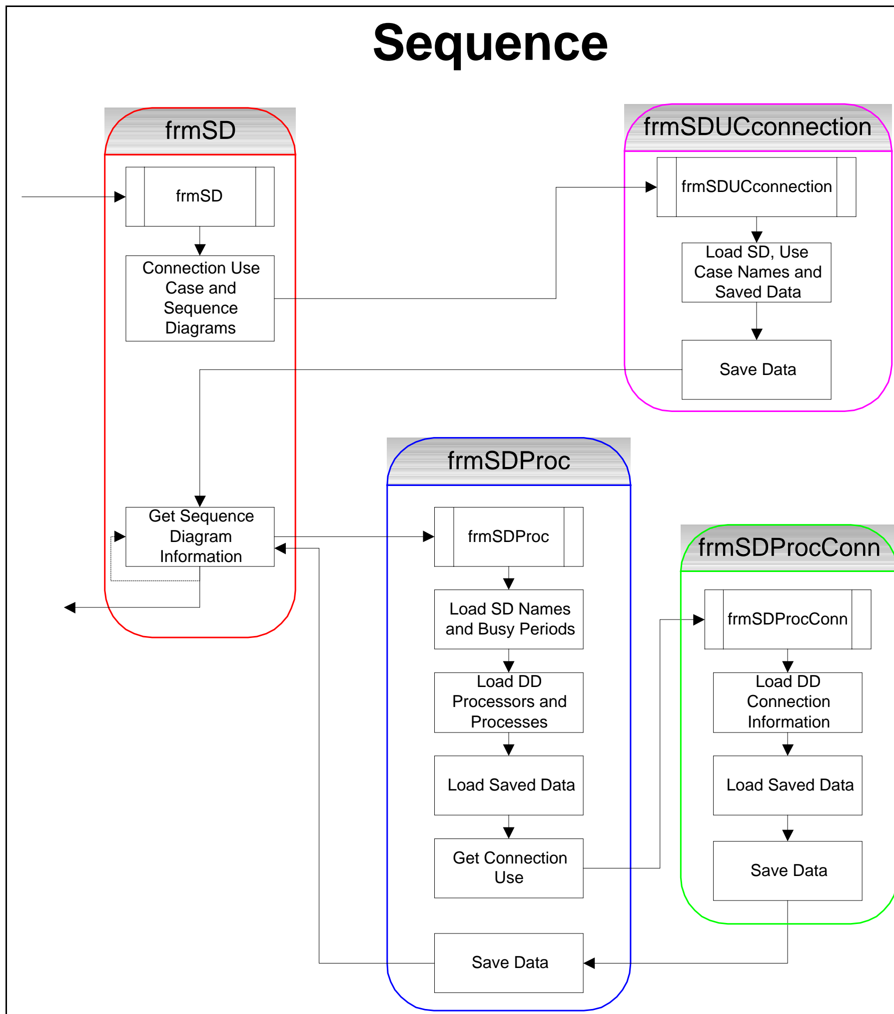

Figure 20 - Sequence Flow Diagram

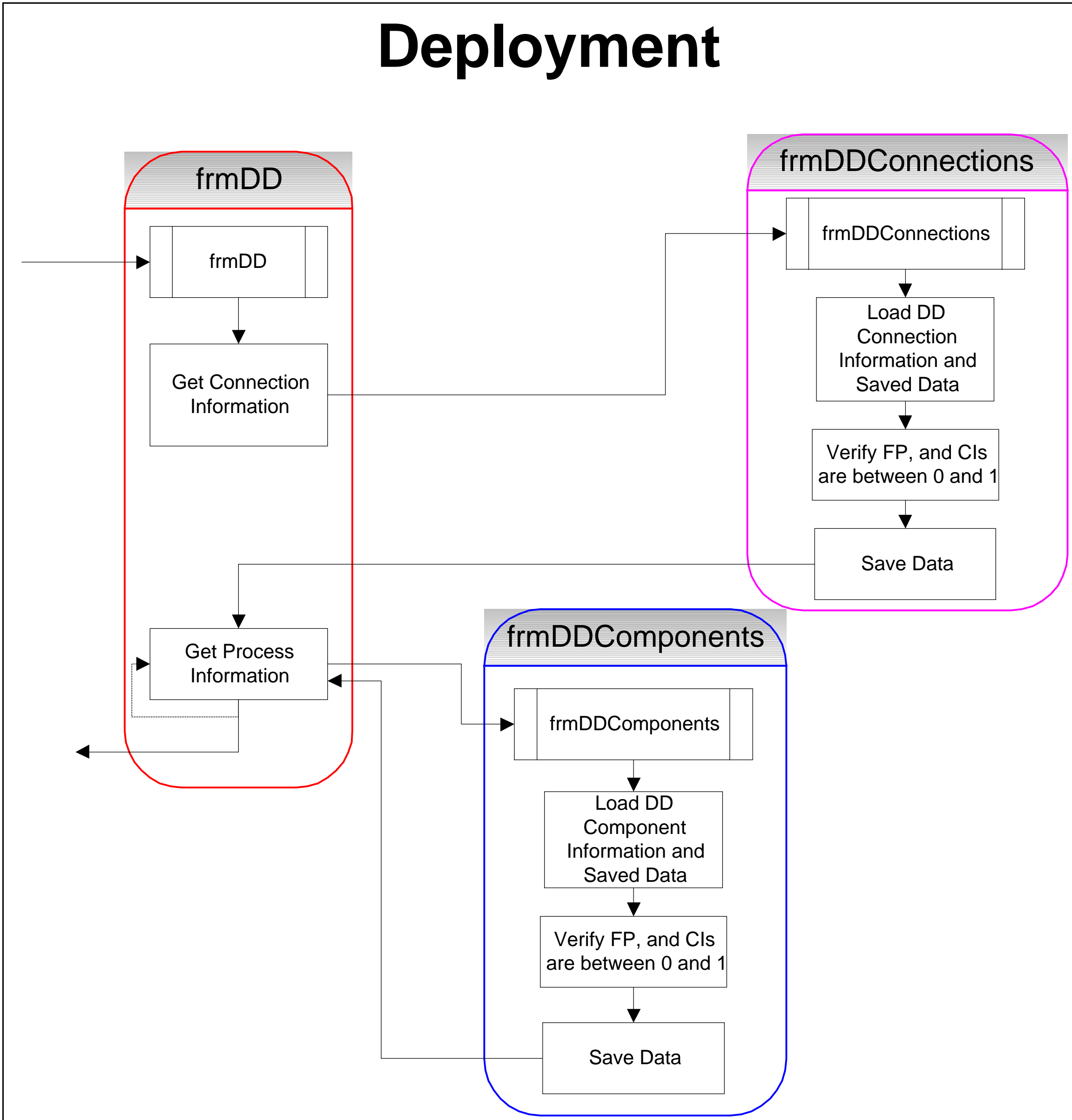

Figure 21 - Deployment Flow Diagram

B.2.1 Forms

This section contains screenshots and descriptions of the various events associated with each form. The forms are displayed in the order that the user will view them.

B.2.1.1 frmSplash.frm

Requirements Prior to Opening: None
Screen View:

Figure 22 - Splash Screen

Events:

Name:	Form_Load()
Description:	Loads form into memory, also sets version and product labels
Name:	Form_Unload(Cancel As Integer)
Description:	Verifies resolution is correct. Loads frmChecklist
Name:	Timer1_Timer()
Description:	Opens Matlab Program
Name:	Timer2_Timer()
Description:	Unloads Form

B.2.1.2 frmChecklist.frm

Requirements Prior to Opening: Matlab Loaded
Screen View:

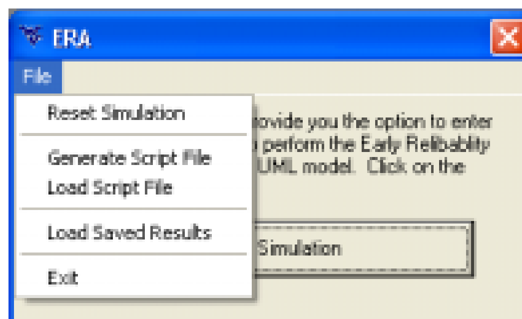

Figure 23 - Start Screen

Events:

Name:	Form_Load ()
Description:	Loads form into memory, also sets Position variable
Name:	Form_Activate()
Description:	Chooses next action to perform by looking at position variable. Hides form when needed and increments position variable
Name:	Form_Unload(Cancel As Integer)
Description:	Calls generic procedure CloseProgram
Name:	cmdStart_Click()
Description:	Updates position variable and calls form_activate
Name:	cmdRun_Click()
Description:	Updates position variable and calls form_activate
Name:	cmdEnd_Click()
Description:	Starts simulation and hides form
Name:	Function OpenFile() As Boolean
Description:	Runs Matlab file plotHist.m
Name:	mnuFileReset_Click
Description:	Updates position variables and calls form_activate
Name:	mnuFileGenerateScript_Click()
Description:	Lets user choose XML file to parse and also XML file to write settings to. Uses WriteFile Class to generate XML settings file.
Name:	mnuFileLoadScript_Click()
Description:	Lets user choose XML file to open. Uses ReadFile Class to load simulation settings.
Name:	mnuFileLoadResults_Click()
Description:	Lets user choose DAT file to open. Sets global variable FileName and Calls cmdEnd_Click
Name:	mnuFileExit_Click
Description:	Calls generic procedure CloseProgram
Name:	OpenFile() As Boolean
Description:	Lets user choose XML file to open. Uses ParseFile Class to parse XML file. Returns true if file successfully parsed.
Name:	SaveSettings()
Description:	Queries user to save simulation settings. If yes, uses WriteFile Class to save simulation settings. Will allow user to retry should WriteFile fail.
Name:	Reverse()
Description:	Updates position variable

B.2.1.3 frmUCD.frm

Requirements Prior to Opening: None
Screen View: Hidden Form
Events:

Name:	Form_Load()
Description:	Loads form into memory, updates local position variable using global UCDposition, and sizes form to 0.
Name:	Form_Activate()
Description:	Chooses next action to perform by looking at position variable. Unloads form when needed and increments local position variable.
Name:	Form_Unload(Cancel As Integer)
Description:	Calls calculateUCProb, shows frmChecklist, and updates global UCDposition
Name:	calculateUCProb
Description:	Uses variable arrays UseCaseActor(x), UseCaseUseCase(x), and UseCaseConnection(x) to calculate the total probability the given Use Case bubble will occur. Saves results to UseCaseUseCase(x).Prob
Name:	Reverse()
Description:	Updates position variable

B.2.1.4 frmUCDActorProb.frm

Requirements Prior to Opening: None
Screen View:

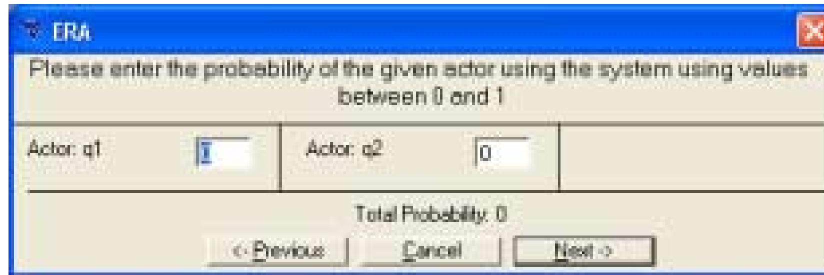

Figure 24 - Use Case Actor Probability Screen

Events:

Name:	Form_Load
Description:	Loads form into memory. Uses variable array UseCaseActor(x) to display actor names. Loads any saved data. Sizes and aligns form elements.
Name:	Form_Unload(Cancel As Integer)
Description:	Verifies to total probability = 1. Stores probabilities and shows frmUCD
Name:	cmdPrev_Click()
Description:	Calls frmUCD.Reverse() and Form_Unload()
Name:	cmdCancel_Click()

Description:	Calls generic procedure CloseProgram
Name:	cmdNext_Click
Description:	Calls Form_Unload()
Name:	txtProb_GotFocus(Index As Integer)
Description:	Highlights data when focused
Name:	txtProb_Change(Index As Integer)
Description:	Verifies value between 0 and 1 and recalculates total probability
Name:	txtProb_LostFocus(Index As Integer)
Description:	Calls txtProb_Change(Index)

B.2.1.5 frmUCDActorConn.frm

Requirements Prior to Opening: global variable CurrentActor to be set
Screen View:

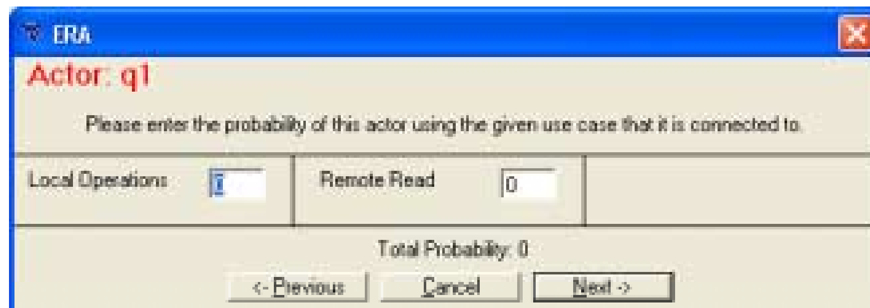

Figure 25 - Use Case Actor Connection Prob. Screen

Events:

Name:	Form_Load
Description:	Loads form into memory. Uses global variable CurrentActor, and arrays UseCaseUseCase(x), and UseCaseConnection(x) to display actor name, and use cases bubbles that actor is connected to. Loads any saved data. Sizes and aligns form elements.
Name:	Form_Unload(Cancel As Integer)
Description:	Verifies to total probability = 1. Stores probabilities in UseCaseConnection(counter_in).Prob. Uses CurrentActor, and arrays UseCaseUseCase(x), and UseCaseConnection(x) to save data. Shows frmUCD
Name:	cmdPrev_Click()
Description:	Calls frmUCD.Reverse() and Form_Unload()
Name:	cmdCancel_Click()
Description:	Calls generic procedure CloseProgram
Name:	cmdNext_Click
Description:	Calls Form_Unload()
Name:	txtProb_GotFocus(Index As Integer)
Description:	Highlights data when focused
Name:	txtProb_Change(Index As Integer)
Description:	Verifies value between 0 and 1 and recalculates total probability

Name:	txtProb_LostFocus(Index As Integer)
Description:	Calls txtProb_Change(Index)

B.2.1.6 frmSDProc.frm

Requirements Prior to Opening: None
Screen View: Hidden Form
Events:

Name:	Form_Load()
Description:	Loads form into memory, updates local position variable using global SDposition, and sizes form to 0.
Name:	Form_Activate()
Description:	Chooses next action to perform by looking at position variable. Unloads form when needed and increments local position variable.
Name:	Form_Unload(Cancel As Integer)
Description:	Shows frmChecklist, and updates global SDposition
Name:	Reverse()
Description:	Updates position variable

B.2.1.7 frmSDUCconnection.frm

Requirements Prior to Opening: None
Screen View:

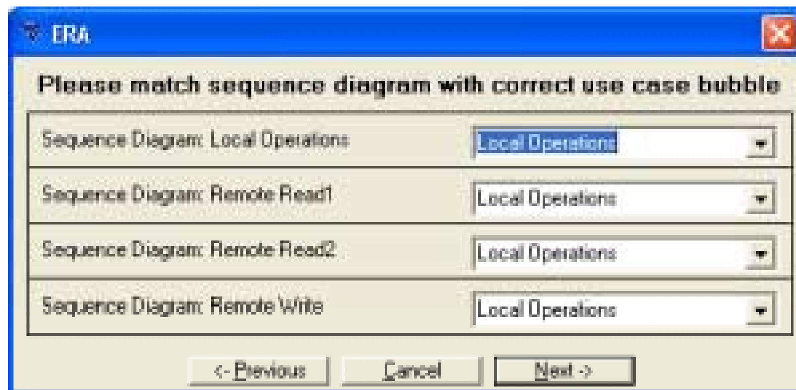

Figure 26 - Use Case – Sequence Diagram Connection Screen

Events:

Name:	Form_Load()
Description:	Loads form into memory. Uses global array SequenceDiagram(x) and UseCaseUseCase(x) to display Sequence Diagram and Use Case names. Loads any saved data. Sizes and aligns form elements.
Name:	Form_Unload(Cancel As Integer)
Description:	Compares choices for matches. Updates UseCaseUseCase(x).Prob if a use case is chosen more than once. Shows frmSD
Name:	cmdPrev_Click()
Description:	Calls frmSD.Reverse() and Form_Unload()

Name:	cmdCancel_Click()
Description:	Calls generic procedure CloseProgram
Name:	cmdNext_Click
Description:	Calls Form_Unload()

B.2.1.8 frmSDProc.frm

Requirements Prior to Opening: global variable CurrentSD to be set
Screen View:

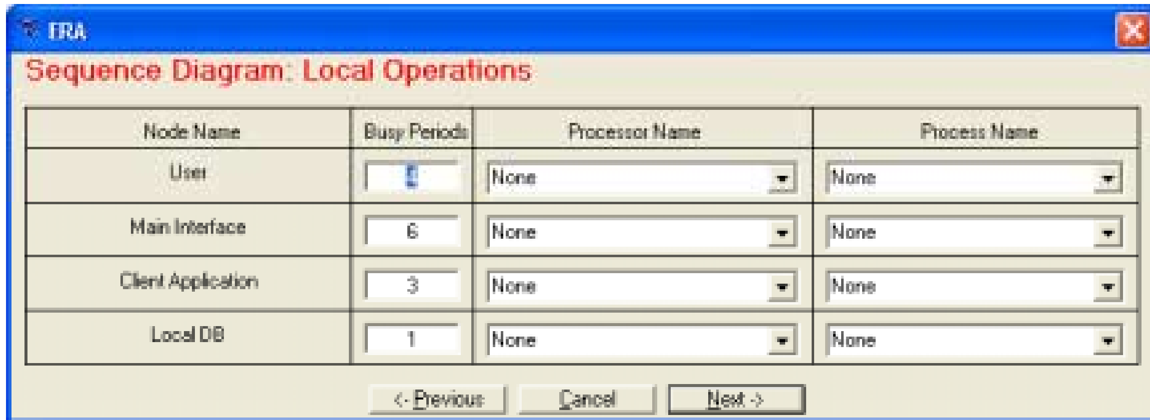

Figure 27 - Sequence Diagram – Deployment Diagram Connection Screen

Events:

Name:	Form_Load()
Description:	Loads form into memory. Uses global variable CurrentSD and array DeploymentProcessor(x) to display node names, and fill processor and process dropdown boxes. Loads any saved data. Sizes and aligns form elements.
Name:	Form_Activate
Description:	Looks at local variable finished to decide whether to unload.
Name:	Form_Unload(Cancel As Integer)
Description:	Stores results in variable CurrentSD, then finds correct location to store in global array SequenceDiagram(x). Shows frmSD
Name:	cmdPrev_Click()
Description:	Calls frmSD.Reverse() and Form_Unload()
Name:	cmdCancel_Click()
Description:	Calls generic procedure CloseProgram
Name:	cmdNext_Click
Description:	Calls Form_Unload()
Name:	cmbProcessor_Click(Index As Integer)
Description:	Updates processes dropdown when user changes the selection of processor.
Name:	txtBP_GotFocus(Index As Integer)
Description:	Highlights data when focused

Name:	txtBP_Change(Index As Integer)
Description:	Verifies value greater than 0
Name:	txtBP_LostFocus(Index As Integer)
Description:	Calls txtBP_Change(Index)
Name:	Reverse()
Description:	Sets local variable finished to false

B.2.1.9 frmSDProcProc.frm

Requirements Prior to Opening: global variable CurrentSD to be set
Screen View:

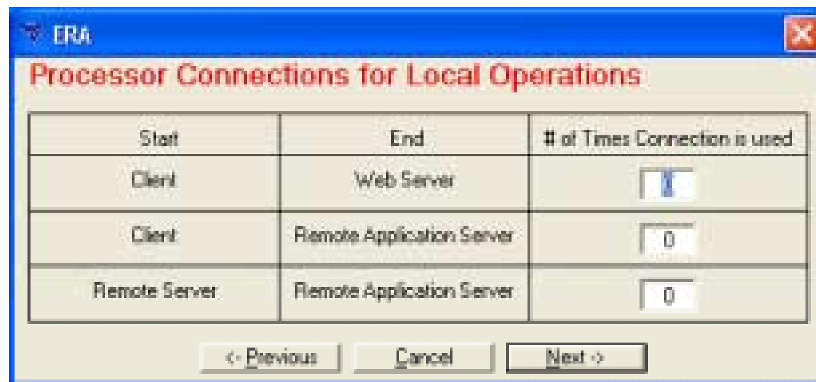

Figure 28 - Deployment Diagram Connection Use Screen

Events:

Name:	Form_Load()
Description:	Loads form into memory. Uses global variable CurrentSD and array DeploymentProcessor(x) to display connections. Loads any saved data. Sizes and aligns form elements.
Name:	Form_Unload(Cancel As Integer)
Description:	Stores results in variable CurrentSD.Connections(i). Shows frmSDProc
Name:	cmdPrev_Click()
Description:	Calls frmSDProc.Reverse() and Form_Unload()
Name:	cmdCancel_Click()
Description:	Calls generic procedure CloseProgram
Name:	cmdNext_Click
Description:	Calls Form_Unload()
Name:	txtCon_GotFocus(Index As Integer)
Description:	Highlights data when focused
Name:	txtCon_Change(Index As Integer)
Description:	Verifies value greater than 0
Name:	txtCon_LostFocus(Index As Integer)
Description:	Calls txtCon_Change(Index)

B.2.1.10 frmDD.frm

Requirements Prior to Opening: None
Screen View: Hidden Form
Events:

Name:	Form_Load()
Description:	Loads form into memory, updates local position variable using global DDposition, and sizes form to 0.
Name:	Form_Activate()
Description:	Chooses next action to perform by looking at position variable. Unloads form when needed and increments local position variable.
Name:	Form_Unload(Cancel As Integer)
Description:	Shows frmChecklist, and updates global DDposition
Name:	Reverse()
Description:	Updates position variable

B.2.1.11 frmDDConnections.frm

Requirements Prior to Opening: None
Screen View:

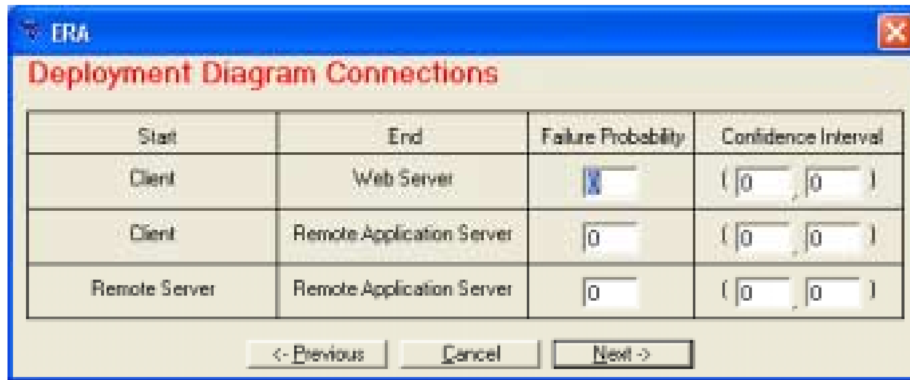

Figure 29 - Deployment Diagram Connection Probability Screen

Events:

Name:	Form_Load()
Description:	Loads form into memory. Uses global arrays ProcessorConArray(x) and DeploymentProcessor(x) to display connections. Loads any saved data. Sizes and aligns form elements.
Name:	Form_Unload(Cancel As Integer)
Description:	Stores results in global array ProcessorConArray(x). Shows frmDD.
Name:	cmdPrev_Click()
Description:	Calls frmDD.Reverse() and Form_Unload()
Name:	cmdCancel_Click()
Description:	Calls generic procedure CloseProgram
Name:	cmdNext_Click

Description:	Calls Form_Unload()
Name:	txtFP_GotFocus(Index As Integer) →*Same for txtCI1 and txtCI2*
Description:	Highlights data when focused
Name:	txtFP_Change(Index As Integer) →*Same for txtCI1 and txtCI2*
Description:	Verifies value between 0 and 1
Name:	txtFP_LostFocus(Index As Integer) →*Same for txtCI1 and txtCI2*
Description:	Calls txtFP_Change(Index)

B.2.1.12 frmDDComponents.frm

Requirements Prior to Opening: global variable CurrentDD to be set
Screen View:

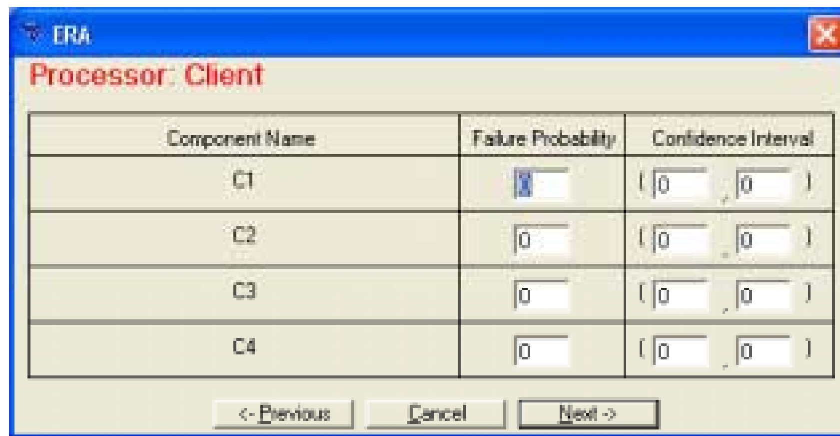

Figure 30 - Deployment Diagram Component Probability Screen

Events:

Name:	Form_Load()
Description:	Loads form into memory. Uses global variable CurrentDD to display processes. Loads any saved data. Sizes and aligns form elements.
Name:	Form_Unload(Cancel As Integer)
Description:	Stores results in global variable CurrentDD and then finds correct location in global array DeploymentProcessor (x) to store results. Shows frmDD.
Name:	cmdPrev_Click()
Description:	Calls frmDD.Reverse() and Form_Unload()
Name:	cmdCancel_Click()
Description:	Calls generic procedure CloseProgram
Name:	cmdNext_Click
Description:	Calls Form_Unload()
Name:	txtFP_GotFocus(Index As Integer) →*Same for txtCI1 and txtCI2*
Description:	Highlights data when focused
Name:	txtFP_Change(Index As Integer) →*Same for txtCI1 and txtCI2*
Description:	Verifies value between 0 and 1
Name:	txtFP_LostFocus(Index As Integer) →*Same for txtCI1 and txtCI2*
Description:	Calls txtFP_Change(Index)

B.2.1.13 frmOutputSettings.frm

Requirements Prior to Opening: None
Screen View:

Figure 31 - Output Setting Screen

Events:

Name:	Form_Load()
Description:	Loads form into memory. Sizes and aligns form elements.
Name:	Form_Unload(Cancel As Integer)
Description:	Validates number of trials and filename. Calculates number of samples and sets to run. Saves data to global variables FileName, Ssamples, and Ssets. Shows frmChecklist.
Name:	cmdSave_Click()
Description:	Lets user choose location to save results at. Displays file path in textbox.
Name:	cmdPrev_Click()
Description:	Calls frmChecklist.Reverse() and Form_Unload()
Name:	cmdCancel_Click()
Description:	Calls generic procedure CloseProgram
Name:	cmdNext_Click
Description:	Calls Form_Unload()
Name:	txtFilePath_GotFocus
Description:	Highlights data when focused
Name:	txtSamples_GotFocus
Description:	Highlights data when focused

B.2.1.14 frmRunSim.frm

Requirements Prior to Opening: None
Screen View:

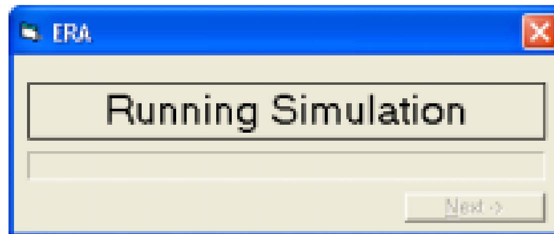

Figure 32 - Running Simulation Screen

Events:

Name:	Form_Load()
Description:	Loads form into memory. Disables next button.
Name:	Form_Unload(Cancel As Integer)
Description:	Shows frmChecklist.
Name:	cmdNext_Click
Description:	Calls Form_Unload()
Name:	timerStart_Timer()
Description:	Disables timers and calls Run_Sim(). This allows the form to completely load before simulation begins.
Name:	Run_Sim()
Description:	Uses RunSimulation Class to create MatLab simulation files. Updates status label and progress bar. Also sends MatLab execution commands. Enables cmdNext button at completion.

B.2.2 Generic Modules

The ECRA project contains two generic modules:

- modProcedures – contains CloseProgram, which asks the user if they really want to quit and closes ECRA if necessary.
- modVariables – contains type definitions and global variables used to store data and control user navigation.

B.2.3 Class Modules

The ECRA project contains four class modules:

- ParseFile – contains the procedures necessary to parse the XMI file produced from Rational Rose.
- ReadFile – contains the procedures necessary to parse the XML file produced by ECRA that contains the simulation settings.
- RunSimulation – contains the procedures necessary to produce the variable and equation file needed to run the MatLab simulation.
- WriteFile – contains the procedures necessary to produce or save the simulation settings to an XML file.

B.2.3.1 ParseFile

Methods:

Name:	Public Function ParseFile(FileName As String) As Boolean
Description:	Removes DTD information from XMI file by coping remaining information to app.path\temp.xml. Calls parse(New File path). Deletes file and returns true if no errors occurred during the parsing process.
Name:	Private Sub parse(Name As String)
Description:	Opens XML document and calls the procedures below.
Name:	Private Sub parseDeploymentD(oXMLDoc As MSXML.DOMDocument)
Description:	Parse XML document for Deployment diagram information. Saves information to global array DeploymentProcessor(x).
Name:	Private Sub parseUseCaseUC(oXMLDoc As MSXML.DOMDocument)
Description:	Parse XML document for Use Case diagram information about use case bubbles. Saves information to global array UseCaseUseCase (x).
Name:	Private Sub parseActors(oXMLDoc As MSXML.DOMDocument)
Description:	Parse XML document for Use Case diagram information about actors. Saves information to global array UseCaseActor(x).
Name:	Private Sub parseUCConnections(oXMLDoc As MSXML.DOMDocument)
Description:	Parse XML document for Use Case diagram information about connections. Saves information to global array UseCaseConnection(x).
Name:	Private Sub parseSDProcesses(oXMLDoc As MSXML.DOMDocument)
Description:	Parse XML document for Sequence diagram information. Calculates busy periods. Saves information to global array SequenceDiagram(x).
Name:	Private Sub parseProcessorConnection(oXMLDoc As MSXML.DOMDocument)
Description:	Analyzes global array DeploymentProcessor(x) to find unique connections. Saves information to global array ProcessorConArray(x).

B.2.3.2 ReadFile

Methods:

Name:	Public Function ReadFile(FName As String) As Boolean
Description:	Loads XML file FName, calls the procedures below, and returns true if no errors occurred while processing the file.
Name:	Private Sub readActors(oXMLDoc As MSXML.DOMDocument)
Description:	Parses XML file for Use Case actor information. Saves information to global array UseCaseActor(x).
Name:	Private Sub readUseCase(oXMLDoc As MSXML.DOMDocument)
Description:	Parses XML file for Use Case - use case bubble information. Saves information to global array UseCaseUseCase(x).
Name:	Private Sub readUCConnection(oXMLDoc As MSXML.DOMDocument)
Description:	Parses XML file for Use Case connection information. Saves information to global array UseCaseConnection(x).
Name:	Private Sub readDDProcessor(oXMLDoc As MSXML.DOMDocument)
Description:	Parses XML file for Deployment diagram information. Saves information to global array DeploymentProcessor(x).
Name:	Private Sub readDDConnection(oXMLDoc As MSXML.DOMDocument)
Description:	Parses XML file for Deployment diagram connection information. Saves information to global array ProcessorConArray(x).
Name:	Private Sub readSD(oXMLDoc As MSXML.DOMDocument)
Description:	Parses XML file for Sequence diagram connection information. Saves information to global array SequenceDiagram(x).

B.2.3.3 RunSimulation

Methods:

Name:	Public Sub Create_VariableFile()
Description:	Writes Matlab file called variables.m that contains the variables needed to run Matlab simulation. Uses global arrays DeploymentProcessor(x) and ProcessorArray(x)
Name:	Public Sub Create_EquationFile()
Description:	Writes Matlab file called equation.m that contains the equation needed to run Matlab simulation. Uses global arrays SequenceDiagram(x), UseCaseUseCase(x), and ProcessorArray(x)

B.2.3.4 WriteFile

Methods:

Name:	Public Function WriteFile(Name As String) As Boolean
Description:	Creates XML file Name, calls the procedures below, and returns true if no errors occurred while writing the file. Uses global arrays UseCaseActor(x), UseCaseUseCase(x), UseCaseConnection(x), DeploymentProcessor(x), ProcessorConArray(x), and SequenceDiagram(x)
Name:	Private Sub WriteActor(newFile As Object, temp As Actor)
Description:	Writes a single actor to the XML file.
Name:	Private Sub WriteUseCase(newFile As Object, temp As UseCase)
Description:	Writes a single use case bubble to the XML file.
Name:	Private Sub WriteUCConnection(newFile As Object, temp As UCConnection)
Description:	Writes a single use case connection to the XML file.
Name:	Private Sub WriteProcessor(newFile As Object, Position As Integer, temp As Processor)
Description:	Writes a single deployment diagram processor along with its processes to the XML file.
Name:	Private Sub WriteProcessorCon(newFile As Object, Position As Integer, temp As PCArray)
Description:	Writes a single deployment diagram connection to the XML file.
Name:	Private Sub WriteSequence(newFile As Object, temp As Sequence)
Description:	Writes information for a single sequence diagram to the XML file, calls WriteSDProcess, and WriteSDConnections.
Name:	Private Sub WriteSDProcess(newFile As Object, temp As Process)
Description:	Writes a single sequence diagram process to the XML file.
Name:	Private Sub WriteSDConnections(newFile As Object, Position as Integer, temp As Integer)
Description:	Writes a single sequence diagram connection to the XML file. Uses global arrays DeploymentProcessor(x) and ProcessorConArray(x)

B.2.4 Matlab files

The ECRA project uses seven Matlab files. These files are either created statically or dynamically. The static files are:

- Alphabeta.m
  - This file calculates a and b using (beta incomplete with upper limit CI2) (beta incomplete with upper limit CI1)
  - Requires: CI1, CI2, and diff where diff = (1-fp)/fp
- Setup.m
  - Starts simulation by calculating the a and b values for each component and connection.
  - Requires: variables.m and alphabeta.m
- Runset.m
  - Runs a single simulation set and saves data to results(x,sample)
  - Requires: sample, samplelength, results, and equation.m
- Closeure.m
  - Combines results into a single array called finalResults and writes to file.
  - Requires: filename
- PlotHist.m
  - Reads file into single array called finalResults. Plots histogram of results and calculates failure probability, and 95% confidence interval.
  - Requires: filename

The dynamic files are:

- Variables.m
  - Contains filename, number of connector, components, samples, samplelength, and connector and component data arrays.
- Equation.m
  - Function that returns a single value using the passed parameters comO and conO, and simulation equation.

Curriculum Vitae

William B. Smith V

Skills

- Professional experience in Ada 85, ASP, C, C++, DOM, Java, JavaScript, HTML, PL/SQL, SQL, UML, Visual Basic, VBScript, XML, and XSL
- Experience with Microsoft SQL Server 2000 and Oracle 8i.
- Professional experience with both Microsoft Visual Studio and Borland programming environments
- Administrative knowledge of all Microsoft Operating Systems including XP
- Experience with Unix
- Proficient in all Microsoft Office products

Education

2001 - 2002 West Virginia University — Morgantown, WV

- Pursing Master of Science in Electrical Engineering with an emphasis on Software Engineering
- GRE: Q: 750 (86%) A: 750 (94%) GPA: 4.0
- Expected Graduation Date: December 2002

1998 - 2001 West Virginia University — Morgantown, WV

- Bachelors of Science in Computer Engineering
- Bachelors of Science in Electrical Engineering
- Graduated Magna Cum Laude, with a final GPA of 3.72

Experience

2002 National White Collar Crime Center — Westover, WV

Software Consultant

- Developed online course management system to serve over 800 agencies across the nation using Microsoft SQL Server and ASP

2001 – 2002 West Virginia University — Morgantown, WV

Research Assistant

- Currently developing a XML-based tool to extend UML using Rational Rose.
- Provided perfective and adaptive maintenance on a software training tool consisting of 1.2 million LOC (C++)
- Released 1 major and 7 minor versions

Summer 2001 NASA — Fairmont, WV

Research Consultant

- Researched ways to develop software visually.
- Studied software life cycle models.
- Wrote paper on the current methods of visual software development.

Experience

2000 - 2001 WVU History Department Morgantown, WV

Web Developer

- Developed on-line course on the history of American religion using WebCT, Real Player, and FLASH.

Summer 2000 Westvaco, Corp. Luke, MD

Summer Intern

- Developed and maintained a computer and IP tracking database.
- Configured and installed over 40 computers.
- Provided network support for NT and Novell based network.
- Provided internet support and technology recommendations.
- Taught two week course on Microsoft Visual Basic 6.0

1999 - 2000 WVU UACDD Morgantown, WV

Network Administrator

- Obtained $70,000 in government contracts
- Designed and maintained Internet based projects.
- Provided network and computer support.
- Maintained an NT server running IIS.
- Designed a statewide database for hospitals in the state of WV.
- Gave multiple presentations on statewide database and ADA standards.

Honors and Awards

- Honorable Mention as National Outstanding Junior Electrical Engineer
- President of Eta Kappa Nu (Electrical Engineering Honor Society)
- Member of Tau Beta Phi (Engineering Honor Society)

Lectures Given

- “Advantage of On-line Databases for Information Capture”, TBI-SCI Governors Committee Meeting: 12 Aug 1999
- “Developing the Internet for the Disabled”, TBI-SCI Conference on Disabilities: 23 – 25 June 1999

Classes Taught

- “Introduction to Microsoft Visual Basics 6.0”, Westvaco, Corp., 20 students: 7 – 18 Aug 2000

References

- Gary Custer – IS Coordinator, Westvaco (301) 359-3311 x 3436
- Dr. Roy Nutter – EE Professor, WVU (304) 293-6371 x 2510

Last Updated: March 17, 2002