Formal Methods for Probabilistic Failure Analysis of Behavioral Specifications

Debasmita Lohar

Formal Methods for Probabilistic Failure Analysis of Behavioral Specifications

Thesis submitted to the
Indian Institute of Technology Kharagpur
For award of the degree

of

Master of Science (by Research)

by

Debasmita Lohar

Under the guidance of

Dr. Soumyajit Dey

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
APRIL 2017

CERTIFICATE OF APPROVAL

Date: / / 2017

Certified that the thesis entitled **Formal Methods for Probabilistic Failure Analysis of Behavioral Specifications** submitted by **Debasmita Lohar** to Indian Institute of Technology Kharagpur, for the award of the degree Master of Science (by Research) has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

(Member of DAC)	(Member of DAC)	(Member of DAC)

(Supervisor)

(Internal Examiner)	(Chairman)

CERTIFICATE

This is to certify that the thesis entitled **Formal Methods for Probabilistic Failure Analysis of Behavioral Specifications**, submitted by **Debasmita Lohar** to Indian Institute of Technology Kharagpur, is a record of bona fide research work under my supervision and I consider it worthy of consideration for the award of the degree of Master of Science (by Research) of the Institute.

Dr. Soumyajit Dey
Assistant Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur,
Kharagpur, West Bengal,
India– 721 302.

Date :

DECLARATION

I certify that

a. The work contained in the thesis is original and has been done by myself under the general supervision of my supervisor.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in writing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

f. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Debasmita Lohar
Department of CSE,
IIT Kharagpur.
Date:

Dedicated to my parents

Acknowledgements

I would like to thank Dr. Soumyajit Dey for his guidance during my Masters' years at IIT Kharagpur. His wealth of knowledge, enthusiasm and perseverance have been encouraging. I really appreciate the effort he has provided to push me in areas that do not come to me quite easily. His attention to detail, while making paper writing a roller-coaster experience, have been very motivating and made the most of our work. I would further like to thank my Departmental Academic Committee members Dr. Dipankar Sarkar, Dr. Pallab Dasgupta, Dr. Anupam Basu, Dr. Pabitra Mitra and Dr. Soumya Kanti Ghosh for their time, many valuable suggestions which motivated me to widen my research from various perspectives.

There are a number of people without whom these years would have been significantly less enjoyable. I would like to thank all my lab mates at Media Lab and later at Formal Lab who have contributed one or the other way making my stay at IIT Kharagpur a memorable experience: Saurav, Anirban, Pallavi, Sudakshina di, Devleena di, Arindam da, Antara di, Sumana di, Sudipa, Sanga di, Sayandeep I and II, Antonio and Arnab. Saurav, Anirban and Sayandeep I deserve special thanks for helping me dealing with intricacies of software installation in general. I would also like to thank Parakrant, Subarna di, Subho da for being great discussion and tea drinking buddies. Abik, Niravra, Souravik, Ayan were always my link to, at times much needed, non-academic reality. They helped me remember that in the end it would be awesome; and it always was.

Last but foremost, I thank my parents without whom this thesis would not have been possible. They showed me that working hard pays off, supported me whenever I needed it, let me try anything and cherished with me every great moment. Thank you very much.

Abstract

Behavioral specifications are often employed for modeling complex systems at high levels of abstraction. Failure conditions of such systems can naturally be specified as assertions defined over system variables. In that way, such behavioral descriptions can be transformed into imperative programs with annotated *failure assertions*. In this thesis, we present a scalable source code based framework for computing failure probability of such programs under the fail-stop model by applying formal methods. We further discuss the design and implementation of ProPFA (Probabilistic Path-based Failure Analyzer), an automated tool developed for this purpose.

Reliability is one of the most important quality attributes of component based complex embedded systems. It is often the case that such systems are required to ensure a minimal guarantee of correct execution. The overall reliability of such systems is determined by the reliability of different possible execution paths and their probabilities. While computation of such exact reliability figures may be expensive and even infeasible in many cases, a minimal reliability guarantee may often be established by inspecting a small number of highly probable execution paths of the system. Based on the above insight, we leverage our failure estimation framework for reliability analysis of high-level behavioral descriptions. We also present an efficient algorithm that performs the reliability analysis of such component based systems with sequential execution semantics.

Our failure analysis framework can be seamlessly employed in the domain of control softwares. We have addressed two different class of problems one being the formal analysis of control theoretic properties like exponential stability criterion. Other class of problems deals with failure analysis of safety critical properties in embedded system softwares. In both cases, exact implementation of the system in C-syntax is statically analyzed leveraging the proposed failure analysis framework.

Contents

List of Figures

List of Tables

List of Abbreviations and Symbols

$\mathbb{Z}$	Set of integers
$\mathbb{R}$	Set of real numbers
σ	Sequential Program
CFG	Control Flow Graph
N	Set of program nodes
E	Set of edges
f	Failure node
π	Program execution path
$\mathcal{S}$	Set of program statements
$\mathcal{L}$	Set of loops
$\mathcal{I}$	Set of conditionals
$\mathcal{A}$	Set of assertions
V	Set of input variables
$\mathcal{C}$	Set of software components
WP	Weakest Precondition
ϕ	Loop invariant
op	Input operational profile
Vol	Volume operation

Pr	Probability
Rel	Reliability
Σ	Linear time invariant plant
T	Sampling period of the plant
$\mathcal{U}$	Environmental uncertainty specification

List of Abbreviations and Symbols

Chapter 1

Introduction

Software is the most crucial component of a wide range of systems ranging from large scale business applications to safety critical control systems for airlines, railways and telecommunications. Some software bugs may cause only trivial problems, but errors in safety critical applications like flight control softwares and medical cyber-physical systems can have unacceptable consequences as noticed in the past. The Therac-25 medical radiation therapy is one of the deadly instances of software failures in the years 1985-87. As a side effect of a buggy software powering the radiation therapy device, 100 times the intended dose of radiation were administered to patients resulting in death of at least three patients [LT93]. In the year 1991, a Patriot missile defense system operating at Dhahran, Saudi Arabia, during Operation Desert Storm failed to track an incoming Scud because of a software problem in the system's weapons control computer. This Scud subsequently hit an Army barrack, killing 28 Americans [Def92]. Another software-induced flight crash happened in Stockholm in 1993. According to the final report of the crash, the control laws were complex which refrained the full analysis of their functions. The effects of control surface movement speed limitations had not been fully investigated for the full flight envelope [Ste03]. Recently, in March 2016, Japan's Hitomi astronomical satellite was destroyed when a thruster fired in the wrong direction, causing the spacecraft to spin faster instead of trying to stabilize. Software bugs are expensive, too. A 2002 study commissioned by the National Institute of Standards and Technology found that software bugs cost the US economy $59.5 billion every year [Tas02]. We can not even imagine the global costs.

The root cause of these failures is the complex construction of software systems managing these applications. Though the overall design complexity is generally distributed among relatively simpler subsystems with pre-specified functionalities, the precise description of the integration logic and the appropriate functional behavior of the integrated system in a specific environment are hard to establish. Formal methods provide a foundation for describing complex systems and reason about behaviors of such systems from a mathematical model. This is a complementary approach to traditional testing-based software development methodologies. Software testing is expensive, and provides only partial coverage of the range of behaviors that a piece of software may exhibit based on only the state sequences actually examined. There is little reason to assume the behavior of untested sequences will be similar to tested ones, and therefore little justification for extrapolating from tested cases to untested ones. On the other hand, the advantages claimed for formal method here is that it is a method which reasons about the entire statespace[1].

Any computer software implicitly specifies the behavior that is exhibited while running on a particular computing system in the expected operating conditions. A behavioral specification typically has two models: an interface model that specifies the external behavior of the component, and a design model which describes the concrete behavior. Such 'program-defined' behavioral specifications are often employed for modeling a complex software system at a high level of abstraction. It provides formal code level abstraction that allows the programmers to express the intended behavior of the program modules in terms of *failure assertions* defined over system variables. If the assertions are not satisfied, the program execution stops which prevents the system to progress to unsafe states as defined by some requirement specification. As an example, in case of fly-by-wire control system in airplanes, the 'Mach Number' (Ratio of aircraft speed by velocity of sound) should remain in the range [0.0 - 0.84]. This necessary requirement is expressed as a failure assertion inside the behavioral specification presented in terms of an imperative program. If this assertion is not satisfied, the system enters into an unsafe state where the wings and tail of the subsonic aircraft gets permanently damaged due to generated shock waves leading to plane crash. Again, even if this

[1]With its own issues pertaining to scalability

assertion executes successfully, there might be a small probability that the wings and tail can be damaged due to some irregular environmental situation. This can be estimated while testing the aircraft. The work reported in this thesis attempts to establish a formal framework for computing the probability of system level failure by analyzing such behavioral abstraction in terms of imperative programs embedded with failure assertions under the assumption of fail-stop failure model. It also leverages the testing data available for the system for precise estimation.

We present *Probabilistic Path-based Failure Analyzer (ProPFA)*, a path based tool that leverages program analysis techniques to compute the failure probability of such imperative programs embedded with failure assertions. Given the input environment, it also takes into account the failure probabilities of subsequent components even if the assertions are executed successfully. ProPFA extracts one program execution path at a time and computes the probability with which all assertions inside the path executes perfectly. Finally all these probabilities are enumerated to provide a scalable failure probability estimate of the whole program. In presence of loops, an optimization is also proposed to handle the exponential blowup in the number of paths using *Invariant Relations*. As is evident, it may not be feasible to consider all program paths. In that case, a quantitative measure for the imprecision in the estimation process due to loss of coverage is captured as a confidence measure. Further, the implementation of ProPFA using well known, robust formal APIs makes the framework usable and extensible.

The framework can be seamlessly employed in the domain of reliability estimation and validation of complex component based software systems. Given the set of probability density functions for input variables as an operational environment, the reliability of a software system at a particular time instant (Point Reliability) can be estimated directly from the failure probability of the whole system. Risk analysis of control softwares with bounded environmental uncertainty is one of the major applications of this framework. As discussed before, the failure conditions in the safety critical control softwares are presented in the behavioral specification as failure assertions. Due to sensor or actuation errors, the system may not attain the stability. We aim to investigate how a control system will actually behave in the face of bounded environmental uncertainty.

1.1 Motivation and Workflow

Any *failure run* of software systems, even if handled, lead to degradation in Quality of Service (QoS) where the definition of such QoS measures is dependent on the area of application [Che80]. For example, in case of Cyber Physical Systems with an underlying networked control infrastructure, environmental noise during transmission or sensor reading leads to unsafe plant state information which needs to be ignored so that an uncalled for control actuation is not passed on to the plant. In that way, the safety of the system is gracefully maintained with an associated degradation in Quality of Control (QoC) [GMGB$^+$14].

Computing the probability of system level failure provides a direct handle for estimating relevant QoS metrics for the system. Analytical methods for computing such failure probabilities using high level models and parameters are widely established. These approaches focus on reliability models or are based on high level system architecture [Gok07, HH11]. However, it has the following shortcomings.

- High level models do not capture complex execution semantics of systems as can be specified in the form of a program behavior.
- In case the system under question is a software program itself, model driven analytical techniques refrain from deriving probability bounds from the source code directly. This implies an added dependence on correctness of model construction given a source implementation.

In case of software systems, existing tools either use failure data obtained during phases of software life cycles to drive one/more of the software reliability growth models [KKLM93, LNF93] or use test coverage measurements [Den97, RGT00] to estimate reliability. Established techniques like probabilistic risk assessment lack the notion of provability as given by formal techniques which provide sophisticated reasoning mechanisms for working with high level system models as well as behavioral specifications given as imperative programs. We intend to provide an end-to-end framework answering the pertinent question of failure probability estimation. The highlighted block in Figure 1.1 presents the input and output of the framework.

Reliability is one of the most important quality attributes of such software

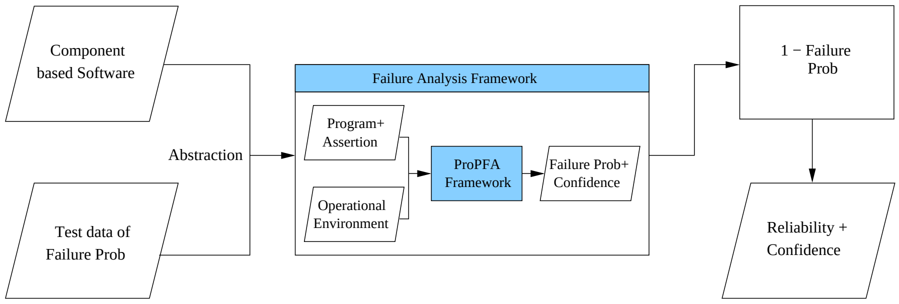

Figure 1.1: The Proposed Framework and its Application in Reliability Analysis

systems designed by integrating heterogeneous components with well defined interfaces. The overall reliability of such systems can be determined by computing the reliability of different possible program execution paths. As illustrated in Figure 1.1, the proposed framework can be seamlessly employed for estimating reliability of software systems. The glue logic of the component based software can be abstracted as an imperative system with *reliability assertions* defining the component interface specifications. These assertions, if satisfied, invoke the subsequent components. Otherwise the system stops execution under the assumption of fail-stop failure model. Component level testing data is also available which define the probability that even after satisfying the reliability assertions, the subsequent component fails resulting failure of the overall system. In practice, computation of such exact reliability figures may be expensive and even intractable. Hence, instead of full system reliability estimation, it makes sense to provide a minimal reliability guarantee by inspecting a finite set of highly probable execution paths. Based on the above insight, the proposed failure analysis framework is employed for software reliability estimation and validation of high-level behavioral descriptions of software systems involving component interface specification contracts expressed in terms of failure assertions.

This framework can also be employed for assuring strict performance guarantees like exponential stability criterion for switched control system implementations. While such guarantees are verifiable by a control engineer at the time of design, when such a system is actually deployed, it may fail to meet control performance requirements due to environmental transient noises in sensing. Erroneous sensor readings get injected to the system primarily due to the following kinds of uncertainties - a) transient faults affecting the sensor readings, b) uncertainty related with external disturbances like environmental noise fluctuations beyond a threshold up to which the control law is robust. In such scenarios, computing the control action over corrupted sensor inputs and accordingly actuating the plant may actually push the system further away from its set point. Hence, it makes sense to analyze the risk involved in control software with bounded environmental uncertainty considering the exact implementation of control software as a C-like program.

Challenges in Formal Analysis : Though formal analysis provides a foundation for describing complex systems and reason about behaviors from a mathematical model, it faces several challenges, one of them being correct abstraction of the whole program in presence of loops. A loop in a program corresponds to several program execution paths, which makes the analysis harder due to exponential increase in the number of paths to be considered. A recent approach towards formal software reliability [FPV13] built on the tool Symbolic Path Finder (SPF) has similar issue. It provides a formal technique for reliability estimation of imperative programs by generating path conditions. In presence of loops, it sets a bound for unfolding the loop. If the bound is reached and loop condition does not fail, SPF backtracks and generates path conditions for which the success status is unknown. In this thesis, we present an optimized software reliability estimation in presence of loops by utilizing the concept of invariant relations. Also, a confidence measure associated with the estimate is provided depending on path coverage. We summarize the objectives and thesis contributions in the subsequent section.

1.2 Objectives and Contributions

In this thesis we propose a theoretical framework for failure estimation of behavioral specification of a software system presented in terms of an imperative program. This scheme utilizes path based failure estimation approach in order to compute the system-level failure probability. This framework is then utilized in the domain of software reliability and risk analysis of control softwares.
To summarize, the major contributions of the thesis are as follows.

- We present a path based framework for estimating failure probability of an imperative program in C-like syntax.

- The proposed approach handles loops gracefully using *invariant relations* (wherever possible) thus resulting in improved scalability. The imprecision in the estimation process are captured as a confidence measure.
- An automatic path based tool flow ProPFA (Probabilistic Path-based Failure Analyzer) integrating state-of-the-art static analysis techniques is developed for this purpose.
- ProPFA finds application in Point reliability estimation and validation of component-based system specifications represented in terms of imperative programs. Additionally, it takes into account the component level testing statistics with the framework to provide a scalable semi-formal solution to the pertinent question of reliability. Experimental results are presented over few case studies from safety-critical avionics and automotive domain exhibiting the applicability of the approach.
- Application of the proposed framework in risk estimation of a control system implementation for an uncertain noisy environment is also presented.

1.3 Work done and Thesis Organization

The focus of this thesis is to estimate failure probability of software systems at early design stages and design an end-to-end framework for the same. We attempt to deploy the framework in diverse domains to provide scalable solutions to the problems of reliability estimation and validation of component based software systems and risk analysis of control software systems. The remaining part of this section summarizes the work done in the thesis.

Chapter 1: Introduction
This chapter provides a brief introduction followed by motivation, objective, scope and main contributions of the thesis.

Chapter 2: Failure Estimation of Behavioral Specifications

In this chapter we have provided an illustration of the proposed theoretical framework for failure probability estimation of behavioral description of a software system presented in terms of an imperative program annotated with failure assertions. The chapter begins with the comprehensive theory followed by the algorithmic details of the proposed approach. Finally, it provides an overview of ProPFA, an automatic tool developed for the purpose. Employing current state-of-the-art static analysis tools ProPFA uses path-based analysis approach to compute the system level failure probability in a push button manner.

Chapter 3: Towards Reliability Analysis of Component based Software Systems
The theoretical framework proposed in Chapter 2 can be seamlessly employed in the domain of reliability. Chapter 3 attempts to establish a provable reliability guarantee of a component based software systems by analyzing the behavioral specifications. Based on path coverage, a measure of confidence is also provided with the estimate. For reliability validation we propose a greedy strategy where execution paths are explored based on their likelihood. Thus we present a scalable approach for reliability estimation and validation by utilizing our theoretical framework.

Chapter 4: Reliability Analysis of Control Software
Chapter 4 provides a formal approach towards probabilistic formal analysis of switched case control software implementations and a detailed investigation on the behavior of a control system in the face of bounded environmental uncertainty.

Chapter 5: Conclusion and Future Work
Finally, chapter 5 provides a summary of the important aspects of the work and proposes certain future directions.

Chapter 2

Failure Estimation of Behavioral Specifications

A behavioral specification is often employed for modeling a complex system at a high level of abstraction. Well known failure conditions of such systems can be naturally specified as assertions defined over system variables. In that way, the situation transforms to an imperative program with annotated *failure assertions*. In this chapter we discuss a framework for computing failure probability of such programs under the fail-stop failure model. We also discuss the design and implementation of a tool developed for this purpose in the subsequent sections.

2.1 Overview of the Proposed Framework

In this section we consider imperative programs with C-like syntax. A program is characterized as a Control Flow Graph (CFG) which is a standard graph based representation of computation and control flow. The nodes of the CFG represent program points at which the program states are captured. After executing each program statement, the state of the program changes. The edges of the CFG are labeled with program statements. The CFG of the program has one entry and one exit node depicting the start and end points of the program respectively. For a program σ defined over a set V of variables, a program state implies a possible *value assignment* to the variables. The CFG can be formally defined as follows.

Definition 1. ***Control Flow Graph of a Program*** *(G_σ): The control flow graph $G_\sigma =< N, E >$ of a program σ has one node $n_i \in N$ for each program point at which the program state is captured. Node n_{start} and n_{end} depict the start and end node of the CFG. An edge $e_i = (n_i, n_{i+1})$ is labeled with a program statement $S_i \in \sigma$. Node n_{end} represents the end state of the program.* □

Figure 2.1 shows an example of CFG representation of a program, i.e., G_σ where node 1 and 6 represent n_{start} and n_{end} respectively.

```
void main(int x, int y)
{
    if ($x \leq 50$){
        assert($x + y \leq 100$);
        $y = y + 2$;
        assert($y \leq 50$);
        $x = x + 1$;
    else
        $x = x - 1$;
        if($x < 60$)
            $x = x + 1$;
            assert($x + y \leq 100$);
            $y = y + 2$;
        else
            $x = x - 1$;
            assert($x + y \leq 100$);
            $y = y - 2$;
}
```

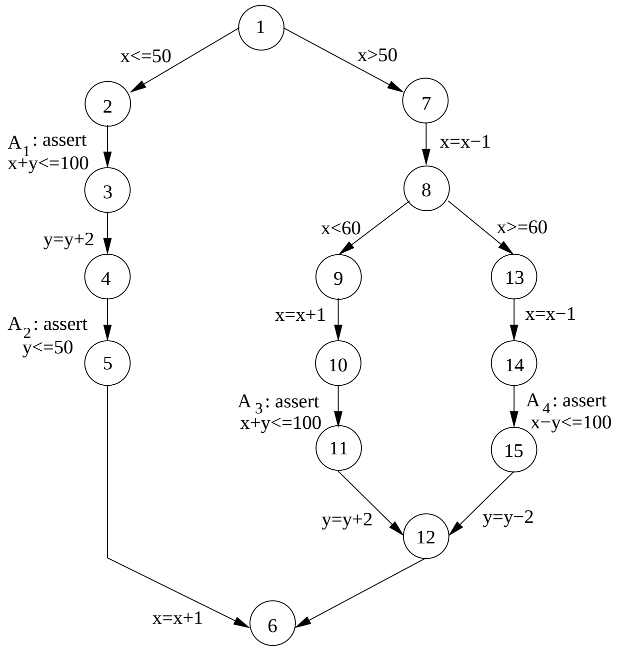

Figure 2.1: Program specifications annotated with Failure Assertions

We consider programs with *Failure Assertions.* A program is modularized into smaller segments and the interactions between them are controlled by placing failure assertions in the beginning of such program segments. These assertions define the predicates that need to be true at that program point. Failure assertions define the region of input state space in which the subsequent program segment executes successfully.

Definition 2. ***Failure Assertion****: A Failure Assertion A placed immediately before some program point 'n' is a predicate defined over a set V of program variables. In an* *execution instance of the program, if A evaluates to 'True', it implies that the program has executed successfully till n. If A evaluates to 'False', it implies that execution of subsequent program segments shall lead to undesired program behavior to be interpreted as a failure of the execution instance.* $\square$

The execution semantics of failure assertions is 'fail stop', i.e., if at least one of these assertions fail, the overall program execution fails. As an example, consider the program presented in Figure 2.1 which is annotated with failure assertions A_1, A_2, A_3 and A_4. These assertions describe the expected preconditions of the subsequent program segments.

For a program σ defined over set V of variables, let $V_{in} \subseteq V$ be the set of input variables. Variables in V_{in} assume values from *input state space* of the program. For example, consider a program P(int x, int y) whose input variables x and y are defined as integers. In this case, x and y can take values from the range of integers [-32768, 32767] if the storage size is 2 bytes. Hence, the input space of the program P is [-32768, 32767]$\times$[-32768, 32767] which defines the *operational profile* of P. In general, the input variables (x and y for program P) can be distributed following some specific distribution within the range. Dynamically, in any random run of a program σ, the set of input variables V takes values from its operational profile following independent/joint probability distributions of the input variables. Depending on these values, the failure assertions are evaluated to 'True' or 'False'. Statically speaking, the assertions may fail from a specific region of the input operational profile. We assume that an input *operational profile*, i.e., a quantitative characterization of probability mass functions and ranges of input variables are made available. In this light, our objective is to compute the failure probability of the program annotated with failure assertions under fail-stop failure model. In summary, the overall problem statement for the work is provided as follows.

Problem Statement : *We are given an imperative program σ in C-type syntax annotated with a set of failure assertions $\mathcal{A} = \{A_1, \cdots, A_n\}$. Any random execution run of σ is considered a 'failure run' iff any one of the failure assertions $A_i \in \mathcal{A}$ actually fail in that run. Given such an instance of σ along with an input level operational profile, we intend to compute the failure probability of σ in any random execution run.*

Let an imperative program σ be a 2-tuple $\langle \mathcal{S}, \mathcal{A} \rangle$ where the set of program statements is represented as ($\mathcal{S}$) and the set of failure assertions is defined as $\mathcal{A}$. The branching nodes of G_σ represent the conditionals present in σ. Let $\mathcal{I}$ be the set of conditionals. Generally, in any random run of σ, the set $V = \{v_1, v_2, \cdots v_k\}$ of variables assumes values from the domain of variables $\{D_1, D_2, \cdots, D_k\}$ following independent/joint probability distributions. In a run of σ, a path of G_σ is followed. A program path (π) starts at node n_{start}, executes a sequence of program statements ($S \in \mathcal{S}$) and failure assertions ($A \in \mathcal{A}$) following the conditionals and ends at n_{end}. Formally, π can be defined as follows.

Definition 3. ***Program Execution Path**: Given a CFG $G_\sigma = < N, E >$ of a program σ, an execution path (π) from the start node n_{start} to node n_{end} is a sequence of nodes $\pi = (n_{start}, n_1, \cdots, n_{end})$ depicting the program points and edges labeled with program statements, conditionals and assertions.* □

The likelihood of failure in an execution path depends on both unsuccessful execution of a failure assertion and the probability of inputs, generated as per the operational profile, actually driving the program down to that particular path where the failure assertion is not satisfied at a particular program point. According to our fail-stop failure model, the program fails if any failure assertion in the path taken do not execute successfully.

Let an assertion (A_i) be a label of an edge ($n_i, n_{i+1} \in E$) in G_σ. The program point n_i characterizes the program state before A_i is executed and n_{i+1} denotes the program state after executing A_i successfully. For our purpose, we apply a transformation function ρ on G_σ as defined below.

Definition 4. ***Transformation** ρ: The transformation ρ is a mapping from $G_\sigma \rightarrow G'_\sigma$ such that, the set of failure assertions $\mathcal{A}$ in σ are converted into conditionals in σ'.* □

For each assertion A_i such that A_i is the label of some edge (n_i, n_{i+1}), we introduce a new node f_i and an edge $e_i = (n_i, f_i)$ labeled with $\overline{A_i}$. This process is continued for all failure assertions $A_i \in \mathcal{A}$. The derived CFG is defined as $G'_\sigma =< N', E' >$ where $N' = N \bigcup \{f_i \mid \exists A_i \in \sigma\}$ and $E' = E \bigcup \{(n_i, f_i) \mid A_i \in \sigma$ and A_i is label of $(n_i, n_{i+1})\}$. The conditional set for G'_σ is defined as, $\mathcal{I}' = \mathcal{I} \bigcup \mathcal{A} \bigcup \overline{\mathcal{A}}$.

If any of these transformed conditionals is not satisfied, i.e., conditional in $\overline{\mathcal{A}}$ is satisfied, the program flow enters into one of the newly introduced nodes. We call them *failure nodes*. The transformed CFG G'_σ corresponding to the CFG G_σ of Figure 2.1 is presented in Figure 2.2. It may be noted that in G'_σ, the only terminating node which is not a failure node is n_{end}. We call this node as *success node*. An execution path (π) is considered as a success path if it starts at the entry node of G'_σ i.e., n_{start}, traverses π and finally reaches the success node of G'_σ i.e., n_{end}. The set of success paths in a program is defined as Π_s. All other program execution paths are considered as failure paths. In Figure 2.2, the execution paths $\{1-2-3-4-5-6\}$, $\{1-7-8-9-10-11-12-6\}$ and $\{1-7-8-13-14-15-6\}$ are included in the set Π_s of the program.

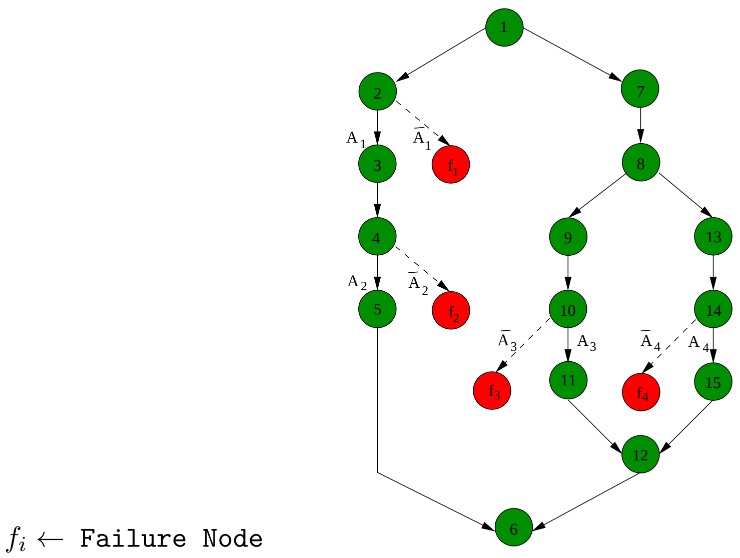

Figure 2.2: The notion of Failure and Success path

We propose a semi formal path based approach for program level failure estimation by employing static analysis techniques [NNH99] on G'_σ. Considering the input profile, there is a specific probability that a program path is taken. We estimate this specific probability associated with each success path. The success probability of the whole program is then enumerated by adding the probabilities of the success paths. Finally, the overall failure probability of the program is estimated. Our method computes the failure probability of the program in Figure 2.1 as 0.23034979 assuming that the variables x and y are uniformly distributed over the integer ranges [0, 100] and [0, 50] respectively.

2.1.1 Failure Probability Estimation of a Program Path

Let us first consider σ as a loop-free program. The original set of conditionals $\mathcal{I}$ of σ contains only the 'if-else' conditionals, while the set $\mathcal{I}'$ contains all original if-else conditionals and the conditionals derived by converting the assertions. For a given input scenario, a path π is considered as a success path i.e., $\pi \in \Pi_s$ iff the control flow reaches the success node (n_{end}) of the transformed CFG following π. Let the total number of conditionals in π be k. The probability $Pr(\pi)$ that any path $\pi \in G'_\sigma$ is taken in any random run of σ can be defined as follows.
Let $\mathcal{I}'_\pi$ be a subset of $\mathcal{I}'$ containing all conditional that occur in a path π. Note that for the path $\pi \in \Pi_s$ to execute, all conditionals $\in \mathcal{I}'_\pi$ need to be evaluated to true. Let the sequence of conditionals $\in \mathcal{I}'_\pi$ be $\{I_1^\pi, I_2^\pi, \cdots, I_k^\pi\}$.

$$\begin{aligned} Pr(\pi) &= Pr(n_{start} \xrightarrow[op]{\pi} n_{end}) \\ &= Pr(I_1^\pi) \times Pr(I_2^\pi / I_1^\pi) \times \cdots Pr(I_k^\pi / (I_1^\pi \wedge I_2^\pi \wedge \cdots \wedge I_{k-1}^\pi)) \end{aligned} \tag{2.1}$$

The term $Pr(I_1^\pi)$ denotes that I_1 is true in π. The term $Pr(I_2^\pi / I_1^\pi)$ denotes that I_2 is true given the fact that I_1 is already known to be true.
The computation of Equation 2.1 can be done using standard program analysis methods like Weakest Precondition (WP) analysis [Dij75].

Definition 5. ***Weakest Precondition Function***: *If S is a code fragment and A is an assertion about states, then the weakest precondition of S with respect to A (written as $WP(S, A)$) is an assertion that is true for precisely those initial states from which S must terminate, and executing S must produce a state satisfying A.* □

The $WP(\pi, A)$ basically provides the initial predicates that if satisfied by the input, will drive the program flow down to the path where assertion A executes successfully. We describe a program path as a sequence of basic blocks and conditionals. Let $\pi = I_1 S_1 I_2 S_2 \cdots S_{k-1} I_k$, where $\mathcal{I}'_\pi = \{I_1, I_2, \cdots, I_k\}$. [1] The S_is are basic blocks comprising sequence of assignment statements of affine expressions.

[1]ignoring superscript π in I_j^π when it is clear from the context.

In this case we need to compute the *path condition* [Kin76] of π which is essentially captured by the following expression.

$$\begin{aligned} WP(I_1S_1I_2S_2,\cdots,S_{k-1},I_k) &= WP(I_1S_1I_2S_2,\cdots,I_{k-1},WP(S_{k-1},I_k)) \\ &= WP(I_1S_1I_2S_2,\cdots,S_{k-2},I_{k-1}\wedge WP(S_{k-1},I_k)) \end{aligned} \tag{2.2}$$

The above recursive expression is used to compute the predicates characterizing the input subspace of the program which drives the execution through path π. We restrict ourselves to static affine programs so that the input subspace generated by WP computation is a convex polytope.
The operational profile '*op*' for a program σ is ideally a probability density function over the input space of the variables in $V_{in} = \{V_1, V_2, \cdots, V_k\}$ such that,

$$\int_{V_1\in D_1}\cdots\int_{V_k\in D_k} op(V_{in}) = 1$$

For the present work, we restrict our attention to '*op*' being defined independently over each variable $\in V_{in}$. For each input variable $v_i \in V_{in}$ let the probability distribution '*op*' be provided over different possible value ranges annotated with the probability masses. For example, a discretized piecewise uniform probability distribution for v_i can be expressed as $([l_i^1,u_i^1],p_i^1),([l_i^2,u_i^2],p_i^2),\cdots,([l_i^{k_i},u_i^k],p_i^{k_i})$ such that $\sum_{m=1}^{m=k_i} p_i^m = 1$. This means that the variable v_i is uniformly distributed within the ranges $[l_i^1,u_i^1],[l_i^2,u_i^2]\cdots[l_i^k,u_i^k]$ with the probability masses $p_i^1,p_i^2,\cdots p_i^k$ respectively. In other words,

$$\int_{V_1\in D_1}\cdots\int_{V_i=l_i^j}^{V_i=u_i^j}\cdots\int_{V_k\in D_k} op(V_{in}) = p_i^j$$

The specification is natural for software running on some digital system with finite precision. It may be noted that, we have only considered uniform and piecewise uniform distribution for input variables. However, complex probability distributions can be typically approximated by dividing the range of possible values or the range of cumulative probabilities into a set of collectively exhaustive and mutually exclusive intervals. We follow such an approach here.

Given this input specifications, if there exists an intersection between the convex polytope $WP(I_1 \cdot S_1 \cdot I_2 \cdot S_2 \cdots S_{k-1}, I_k)$ and the input state space, then with input variables assuming values from the intersection region, the path π actually executes, i.e., π is a *feasible* path.

Let $Pr(\pi)$ denote the probability that π executes any random run of the program. In case the input operational profile 'op' is uniform,

$$Pr(\pi) = \sum_{\mathcal{V}_i} \frac{Vol(WP(I_1 \cdot S_1 \cdot I_2 \cdot S_2 \cdots S_{k-1}, I_k) \wedge \mathcal{V}_i)}{Vol(\mathcal{V}_i)} \tag{2.3}$$

where $Vol(C)$ denotes the volume of convex polyhedron C. In case the input operational profile is discretized as piecewise uniform, a polyhedron C can be decomposed accordingly so that the weighted volume $Vol(C, op)$ may be computed.

The failure probability $Pr(\sigma)$ of the program σ can thus be estimated as follows.

$$Pr(\overline{\sigma}) = 1 - \sum_{i=1}^{|\Pi_s|} Pr(\pi_i) \tag{2.4}$$

where $\Pi_s = \{\pi_1, \pi_2, \cdots \pi_m\}$, $m = |\Pi_s|$.

2.1.2 Failure Estimation of programs with loops

Till now we have considered loop-free programs. A loop in a program σ corresponds to several program paths depending on the number of loop iteration. Loop unrolling replaces the loop by as many instances of its body as the number of iterations. Thus, for success probability estimation of σ, all generated paths leading to the success node are considered and success paths and the probabilities of them are computed as discussed in Section 2.1.1. Apart from the additional requirement of loop bound analysis, in general, this method will not scale for large programs with significantly deep loops.

To accelerate the analysis in presence of loops, a technique is proposed which sacrifices some accuracy while improving scalability. This optimization works on simple computational loops which do not contain failure assertions. L = `'while (`C`) do `S'`;` The physical significance of such a loop L_i is, the program segments inside do not fail in any event. For failure probability computation of such loops we apply the concept of *Invariant Relations.* A loop invariant is a condition that is necessarily true immediately after each iteration of a loop [MGL+11]. Typically, modern invariant synthesis tools generate invariants that satisfy a given post condition in terms of assertions. In our case, the post condition is the assertion defining successful execution region of the subsequent program module. Let $\sigma = \sigma'$; [while (C) do S;] S'; A'; σ^2; where σ', σ^2 are prefix and suffix program segments, S is the loop body, S' is the basic block immediate next to loop and A' denote the conditional immediately next to S'. We compute $WP(S', A')$ and use this as a post condition for the loop in order to synthesize a loop invariant ϕ such that $\phi \Rightarrow WP(S', A')$ if ϕ exists. It may be noted that,

$$\sigma = \sigma' \cdot \overline{C} \cdot S' \cdot A' \cdot \sigma^2 + \sigma' \cdot (C \cdot S)^+ \cdot \overline{C} \cdot S' \cdot A' \cdot \sigma^2$$

such that σ is a disjunction of two subprograms,

1. σ' executes and ensures $\overline{C}$ is true followed by execution of S' which in turn ensures that $WP(S', A')$ is true followed by execution of σ^2. Let the characterization of this path be, $\pi^{\overline{L_i}}$.

2. σ' executes and ensures C is true followed by execution of the loop for any finite no. of iterations such that on exit from loop, ϕ is true followed by execution of S' making A' true followed by execution of σ^2. Let the characterization of this path be, π^{L_i}. The computation of σ enables WP computation for loops without unrolling by individually handling both the programs. Actually for this case, we leverage the concept of path programs i.e., soundly merging information about executions along multiple paths into one single path which is π^{L_i} in this case.

The first case completely bypasses the loop. Hence, success probability estimation of this path $(Pr(\pi^{\overline{L_i}}))$ is computed in a similar manner as discussed in Section 2.1.1. The second case represents the scenario where program flow actually enters into the loop and leaves the loop after some finite number of iterations thus bringing the invariant into the picture. Current state-of-art invariant synthesis tools assume a post condition after the loop to be true. Hence, the WP of the nearest downstream assertion $WP(S', A')$ is considered as the post condition while synthesizing the loop invariant. Let us assume ϕ_i be the synthesized invariant of

the loop L_i. In general, the actual invariant ϕ can be thought of as a disjunction of invariants which may be possibly uncovered by typical invariant synthesis tools. Even state-of-the-art invariant synthesis tools cannot generate disjunctive invariants in most practical situations [GR09]. Hence, in actuality we only have a partial characterization of ϕ in terms of component loop invariants. Depending on whether we are able to synthesize some loop invariant or not, two situations arise which are discussed separately as follows.
Case 1 : In case we are able to generate invariants, i.e. we are successful in identifying program paths which execute the loop at least once followed by $WP(S', A')$ being true. In general, the synthesis tool may fail to generate the exact invariant capturing all possible paths (which satisfy $WP(S', A')$) as provided by the loop. The exact set of paths for π^{L_i} are captured by the Kleene expression $\sigma_1 \cdot [\overline{C} + (C \cdot S)^+ \cdot \phi] \cdot S' \cdot A' \cdot \sigma_2$ while the subset of paths covered by the analysis are actually $\sigma_1 \cdot [\overline{C} + (C \cdot S)^+ \cdot \phi_i] \cdot S' \cdot A' \cdot \sigma_2$. The success probability of the path containing loop L_i can be expressed as,

$$\begin{aligned} Pr(\pi^{L_i}) &= Pr(\sigma' \cdot ((C \cdot S)^+ \cdot \phi_i) \cdot \sigma^2) \\ &\leq Pr(\sigma' \cdot ((C \cdot S)^+ \cdot \phi) \cdot \sigma^2) \end{aligned} \tag{2.5}$$

Since the exact probability contribution cannot be accounted for due to the non-exact nature of the invariant, we actually underestimate the success probability of the program execution path which in turn overestimates failure probability leading to a safe approximation.
Case 2 : In case the invariant generator fails to compute any invariant which may satisfy the post condition ($WP(S', A')$), it is likely that the executing the loop at least once ensures that the post condition $WP(S', A')$ fails since no such computational path could be discovered. However due to the lack of completeness of invariant generation techniques, it may not be necessarily true that execution of loop L guarantees the postcondition to fail. Hence, for this case, we unroll the loop, generate possible execution paths and add the success contributions of these paths to the overall success probability of the system which in turn is used to compute the failure probability of the overall program.

In general, most loops found in actual scenarios require simple quantifier-free invariants that are conjunctions of variables. For these loops, it is advantageous to compute invariants [CH78]. However, some loops (i.e., multi-phase loops) in real programs fundamentally require disjunctive invariants, for which the synthesis tool may fail to generate sufficient invariants satisfying $WP(S', A')$. To minimize the overestimation, such loops are unrolled as discussed earlier.

2.1.3 Estimating Confidence Measure

Due to loop approximations as discussed above and specified time and memory bounds, all the execution paths of the system may not be explored. We propose a confidence measure to indicate a formal bound of analysis coverage. Let us consider that after approximating all *simple* loops, we were able to explore k success paths with execution probabilities $Pr(\pi_1), \cdots, Pr(\pi_k)$. Let the set of failure assertions covered by these k paths be $\mathcal{A}_k$. For any $A \in \mathcal{A}_k$, let $\overline{A}$ be the label of an edge (n, f) where f is a failure node as per the construction of G'_σ. We enumerate the set F_k of all such failure nodes and compute the total (success + failure) coverage for k paths as follows.

$$coverage = \sum_{i=1}^{i=k} Pr(\pi_i) + \sum_{f \in F_k} Pr(\pi = (n_s, f)) \tag{2.6}$$

where π is a failure path with source, target pair $= (n_s, f)$. We report this quantity as the confidence measure. For estimating confidence, we have considered two distinct cases.

1. For computing the confidence measure in case a program execution path π does not contain any simple loop, all failure assertions $A_i \in \mathcal{A}_\pi$ are set as 'TRUE'. In effect, while estimating the confidence, both the underlying success path and failure path (with assertion $A_i \in \mathcal{A}_\pi$) are covered.

2. However, this efficiency can not be achieved in case of program execution path with simple loops whose failure probabilities are estimated using invariants as described in Section 2.1.2. This is because while computing invariants we might have missed some program paths due to incompleteness of invariant synthesis. Let A_i be the failure assertion just after loop L_i in a path π. The loop invariant ϕ_i for loop L_i is utilized to compute the failure

probability of the path π. If we consider A_i as 'TRUE' here, the estimated measure of confidence will be greater than the actual value in case ϕ_i is not complete. There might exist paths with unknown status in L_i, meaning we have no information whether these paths contribute to the success or failure of the program. Hence, we can not provide any guarantee for these paths. For this reason we consider both the success and failure of A_i (i.e., synthesize invariants for both A_i and $\overline{A_i}$) and compute the probability of π being explored with success/failure of A_i, represented as $Pr(A_i)$ and $Pr(\overline{A_i})$. Similarly, for a path π with two loops L_1, L_2 having subsequent failure assertions A_1, A_2, we again consider all possible success as well as failure scenarios individually, i.e, we compute $Pr(A_1)$, $Pr(\overline{A_1})$, $Pr(A_1\overline{A_2})$, $Pr(\overline{A_2})$. Similarly we can extend the argument for any number of simple loops in a path. All other assertions apart from the assertions following the loops can be considered as 'TRUE' as described earlier.

2.1.4 Generalized Algorithm and Performance Analysis

Our generalized algorithm starts with an input program σ annotated with a set of failure assertions $\mathcal{A}$. Let $\mathcal{L}$, $\mathcal{L}_A$, $\mathcal{L}_I$ denote the sets of looping structure present in σ, loops with assertions inside the loop body and loops followed by an assertion but no assertion inside the loop body respectively. Hence, $\mathcal{L} = \mathcal{L}_A \bigcup \mathcal{L}_I$. The procedure `MAIN` (Algorithm 2) makes a call to `PROCESS_LOOP` (Algorithm 1) for synthesizing invariants of the set of loops without failure assertions ($\mathcal{L}_\mathcal{I}$). In Algorithm 1, the function `RANGE_ANALYSIS` (Line 5) creates an Abstract Syntax Tree (AST), generates data-flow equations, iteratively solves them until a fixed point is reached or till a fixed number of iterations (which can be overwritten manually) and finally generates the ranges of program variables at each loop entry point. This step actually provides initial templates (set of 'assume' clauses) in terms of linear inequalities over the program variables and annotates the loops with the generated assume clauses for invariant synthesis. The function `INV_SYNTHESIS` (Line 7) invokes a state-of-the-art invariant synthesis tool and generates invariant relations for each loop $l_i \in \mathcal{L}_\mathcal{I}$ wherever possible. Then the invariants are annotated before the corresponding *while* loops in σ (Line 7). The preprocessed program σ is considered next for failure analysis (Algorithm 2).

In Algorithm 2, after annotating invariant relations in the loops wherever possible (Line 8), the loops with assertions ($\mathcal{L}_A$) and the loops without assertions for which invariant relations could not be generated due to the limitations of invariant synthesizers ($\mathcal{L}_U$ such that $\mathcal{L}_U \subseteq \mathcal{L}_I$), are unrolled by an user provided `UNROLL_FACTOR` (Line 9-10). Then the transformation function ρ as defined in Definition 4 is applied (Line 12) on the original program σ in order to generate σ'. As discussed in Section 2.1.1, ρ converts each assertion $A_i \in \mathcal{A}$ into 'if-else' conditionals. Let Π be the set of success paths in σ'. Till the set Π is not empty, i.e., if there exists a success path in σ', function `EXTRACT_PATH` extracts an unprocessed success path $\pi_i \in \sigma'$ and function `WP_ANALYSIS` is called (Line 13) which utilizes a third party static analyzer for weakest precondition computation. If path π_i does not contain a loop l_i, it directly computes the volume by calling `VOLUME_COMPUTATION` function (Line 23). It leverages model counting tools for estimating volume of polytopes. The confidence measure is also computed (Line 2) as discussed in Section 2.1.3.

Algorithm 1: PROCESS_LOOP: Process loops without failure assertion

```
Input: Program annotated with failure assertions
Output: Program annotated with assertions and invariants
1: $\sigma \leftarrow$ Input program with failure assertions.
2: $\mathcal{L}_I \leftarrow$ Set of loops without assertion inside loop body;
3: $A_i \leftarrow$ Assertion subsequent to $l_i$, for $\forall\ l_i \in \mathcal{L}_I$
4: $\Gamma_i \leftarrow$ Set of invariant templates of loop $l_i$; $\phi_i \leftarrow$ Set of invariants of loop $l_i$
5: $\Gamma_i \leftarrow$ RANGE_ANALYSIS($l_i$); Annotate $l_i$ with $\Gamma_i$
6: for each $l_i \in \mathcal{L}_\mathfrak{I}$ do
7:     $\phi_A \leftarrow$ INV_SYNTHESIS($l_i, A_i$); Annotate $l_i$ with $\phi_A$
8: end for
```

If π_i contains a loop with invariants ($l_i \in (\mathcal{L}_I \setminus \mathcal{L}_\mathfrak{U})$), the estimation of success probability is optimized as discussed in Section 2.1.2. The loops remains intact in the path π_i while computing the overall success probability. For estimating confidence, the weakest precondition for both success and failure paths are generated (Line 15-17), the probability of these paths are computed using the function `VOLUME_COMPUTATION` and added with the overall confidence measure (Line 24). This whole process is continued till there exists an unprocessed path $\pi_i \in \Pi$. Finally the failure probability along with a confidence measure is returned (Line

27).

Algorithm 2: Failure Estimation of a program annotated with Failure Assertions

```
Input: A program annotated with failure assertions, Ranges of input variables with probability distributions.
Output: Overall Failure Probability of a program with a confidence measure.
 1: $\sigma \leftarrow$ Input program with failure assertions; $\sigma' \leftarrow$ Program after transformation.
 2: $\mathcal{L}_U \subseteq \mathcal{L}_I \leftarrow$ Set of loops without assertions for which invariants could not be generated.
 3: $\mathcal{L}_A \leftarrow$ Set of loops with failure assertion inside the loop body.
 4: $\Pi \rightarrow$ Set of program execution path in $\sigma'$; $\pi_i \rightarrow i^{th}$ program execution path in $\sigma'$
 5: $P_i \leftarrow$ Set of Predicates generated for path $\pi_i$; $\mathcal{I}_i \leftarrow$ Set of Conditionals in path $\pi_i$
 6: $\pi_i' \leftarrow$ Path $\pi_i$ s.t. $\forall A_i \in \mathcal{I}_i \rightarrow TRUE$; $\pi'_{i\_A} \leftarrow$ Path $\pi_i$ s.t. $(\forall A_i \in \mathcal{I}_i) \setminus A \rightarrow TRUE$;
 7: $\mathcal{A}_i \rightarrow$ Set of assertions transformed into Conditional in $\sigma'$
 8: $\sigma \leftarrow$ PROCESS_LOOP($\sigma$)
 9: for each $l_i \in \mathcal{L}_A \bigcup \mathcal{L}_U$ do
10:     $\sigma \leftarrow (\sigma \setminus l_i) \bigcup$ LOOP_UNROLL($l_i$, UNROLL_FACTOR )
11: end for
12: $\sigma' \leftarrow \rho$_TRANSFORM($\sigma$)
13: if $\Pi \neq \emptyset$ then
14:     $\pi_i \leftarrow$ EXTRACT_PATH($\sigma'$); $P_i \leftarrow$ WP_ANALYSIS($\pi_i, I_k$)
15:     if $l_i \in \pi_i$ then
16:         $P_{i\_A}$= WP_ANALYSIS($\pi'_{i\_A}, I_k$)
17:         $\phi_{\overline{A}} \leftarrow$ INV_SYNTHESIS($l_i, \overline{A_i}$); Annotate $(l_i \setminus \phi_A)$ with $\phi_{\overline{A}}$
18:         $P_{i\_\overline{A}}$= WP_ANALYSIS($\pi'_{i\_\overline{A}}, I_k$)
19:         $p_i \leftarrow (P_{i\_A}) \bigcup (P_{i\_\overline{A}})$
20:     else
21:         $p_i \leftarrow$ WP_ANALYSIS($\pi_i'$)
22:     end if
23:     $Pr(\sigma) \leftarrow Pr(\sigma)+$ VOLUME_COMPUTATION($P_i$)
24:     $Confidence \leftarrow Confidence+$VOLUME_COMPUTATION($p_i$)
25:     $\Pi \rightarrow \Pi \setminus \pi_i$
26: end if
27: RETURN $(1 - Pr(\sigma), Confidence)$
```

2.2 ProPFA: Probabilistic Path-based Failure Analyzer

In this section, we introduce Probabilistic Path-based Failure Analyzer (ProPFA), an automatic tool for failure estimation of imperative programs. ProPFA takes as input a program annotated with failure assertions, the discrete ranges of the input variables and the probability density functions associated with all input variables. It uses a path based failure analysis approach and finally returns overall failure estimate of the program with an associated confidence measure. A command-line version of ProPFA is available at https://github.com/dlohar/ProPFA.git. Figure 2.3 shows the overall architecture of the tool. Its input language is C.

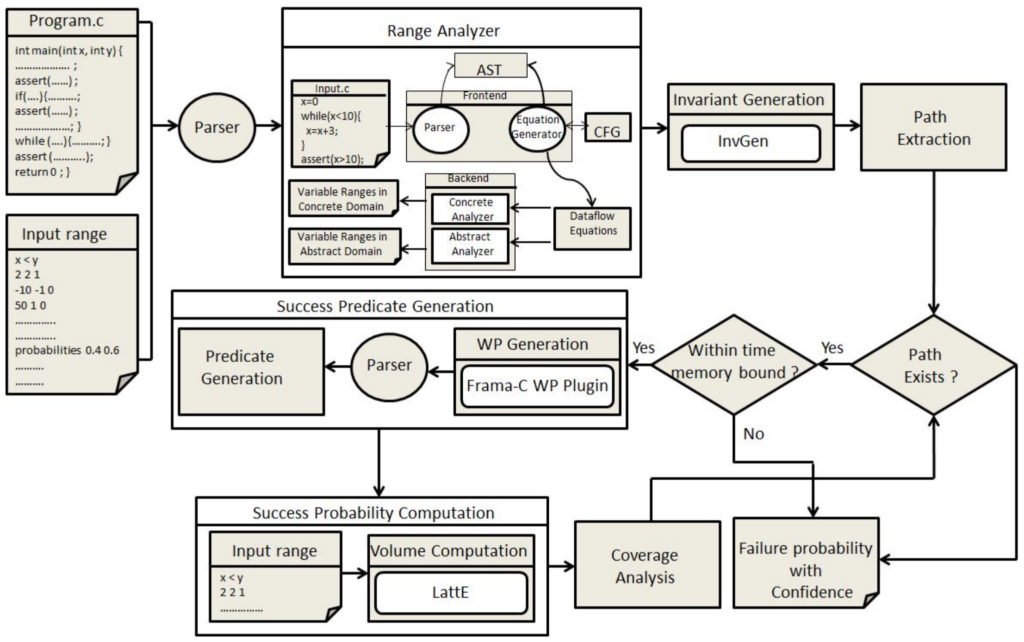

Figure 2.3: Architecture of ProPFA

2.2.1 Key Features of ProPFA

The ProPFA tool leverages state-of-the-art static analysis tools for failure estimation. It is designed as an integrated framework that takes input programs annotated with failure assertions and input distributions. Multiple execution paths are then explored depth-wise and for each path, success predicates are generated using third party static analyzer. The probabilities of these predicates under the operating region are computed using lattice point enumeration which in turn is used for program level failure probability estimation.
We mainly use Frama-C WP plug-in [CKK$^+$12], a source code analysis platform that generates weakest preconditions of industrial-size C programs, as a third party static analyzer. For loops in the source code, invariants are synthesized by deploying state-of-the-art template based tool InvGen [GR09]. Finally, failure probability computation involves volume computation of convex polytopes which is offloaded to the lattice point counting tool LattE [DLHTY04].

Next, we separately describe the components of ProPFA, following Figure 2.3.

- **Range Analyzer :** This module is a minimalistic Range Analyzer for C-type programs. It analyzes a program statically and determines an interval of values for each of its variables at each program point (the abstract analysis result in Figure 2.3). It can also compute the set of discrete values each program variable can take (the concrete analysis result in Figure 2.3). This tool is used in conjunction with the invariant generation tool called InvGen for generating linear arithmetic invariants. InvGen requires an initial invariant template at the beginning of each loop in the program. The range analyzer provides these templates to InvGen. The range analyzer module is divided into two major parts : the *frontend* and the *backend.* The frontend consists of two components, a *Parser* for the input program, which generates the *parse tree* and *abstract syntax tree*(AST). The second component of the front end is a *Data Flow Equation Generator*, which takes as input the abstract syntax tree generated by the parser module, traverses the AST and produces a *control flow graph* which is used to generate the set of data flow equations. The data flow module then generates a set of *data-flow equations* from the control flow graph it generated previously.
  The *backend analyzer* solves those data flow equations generated by the frontend. Backend is again subdivided in two parts : one for analysing in the *concrete domain* and the other for the *abstract domain.* The abstract domain analyzer also exploits the concept of *widening technique* to minimize the number of iterations while solving the equations. Both of these sub engines solve the data flow equations separately in their respective domains and output the ranges of program variables at each program point.

- **Invariant Generation :** The input program is parsed and for each `while` loop without any failure assertion inside the loop block, ProPFA uses InvGen to generate invariants. Initial templates for InvGen are provided by the range analyzer module of ProPFA. Such automatically synthesized templates are in general precise enough for invariant generation in most cases. InvGen implements constraint solving approach and finds an instantiation of the template parameters that yields a safe invariant satisfying the subsequent assertion. The invariants are then used to abstract out the loops in the input program to accelerate failure analysis. ProPFA also computes the confidence corresponding to the loop under inspection as described in Section 2.1.3.

If no invariant is generated in any of these cases, ProPFA allows loop unrolling by user provided unrolling factors. After unrolling, each loop iteration is considered as a separate program execution path and success probabilities of each path is computed which in turn is used to compute the failure estimation of the whole program.

- **Path Extraction :** After abstracting out the loops wherever possible, within a specified time and memory bound, ProPFA extracts all possible program execution paths depth-wise. Assertions in each path are considered for success analysis under the fail-stop failure model until there is no assertion left while the end node of the CFG is reached. The path probability is computed in similar fashion in order to estimate the confidence measure.

  ProPFA generate weakest preconditions for each success path by employing WP plug-in of Framework for Modular Analysis of C program (Frama-C). It implements a weakest precondition calculus for ACSL (ANSI C Specification Language) annotations through C programs. ACSL is a Behavioral Interface Specification Language (BISL) implemented in the Frama-C framework. It is interfaced with ProPFA to compute the WP for a program segment P and post-condition A. The specific goal of input predicate generation is to find the initial region for which the failure assertion A in the specific path executes successfully. This region defines the success region of A.

- **Failure Probability Computation :** In this module, we intend to compute the volume of the convex polytope that is generated by the intersection of generated weakest preconditions of success paths with the initial input regions. The discrete domain model counting tool 'LattE' is integrated with ProPFA in order to compute the volume of the success region. If the intersection includes the initial input region fully, it guarantees that the program does not fail for that particular assertion. After estimating success probabilities of all assertions, we estimate the failure probability of the program employing Eq. 2.4.

It may be noted that, the ProPFA considers uniform distribution for input variables within a specified region. It also allows discrete uniform regions of input variables associated with probability distribution functions.

2.2.2 External Tools Used

In this section we provide a brief overview of the external tools interfaced with ProPFA.

- **Frama-C WP Plug-in :** Frama-C [PL10, CKK+12] is a platform dedicated to the static analysis of industry-level C source code base. It has a collaborative and extensible approach that allows plug-ins to interact with each other. The Frama-C deductive verification plug-in WP is a tool for compositional verification of C code based on code contracts. The plug-in implements a weakest precondition calculus for ACSL annotations through C programs. It starts with C programs annotated with behavior specifications and then generates a bundle of proof obligations, i.e., mathematical first-order logic formula that must be valid in order for each program unit to meet its specification. All annotations are written as comments, using one of the notations //@ $\cdots$ or /*@ $\cdots$ @*/, for single- and multi-line annotations, respectively. We intend to compute the initial predicates in terms of linear equations. Hence, ProPFA converts the generated proof obligations in terms of first order logic formulae to linear half space equations.

- **InvGen :** InvGen [GR09] is an efficient automatic tool for synthesizing linear arithmetic invariants for imperative programs. Given a set of initial templates in terms of boolean combination of linear inequalities over program variables at a loop entry point, it generates invariants of the loop. InvGen combines static and dynamic analysis for this purpose. InvGen also deals with multiple program paths and incrementally generates safe invariants for each path.

  It may be noted that InvGen is highly sensitive to the initial template choices. If the template is too expressive, e.g., if it admits a number of conjuncts that is larger than required, then the efficiency of InvGen decreases due to the increased difficulty of constraint solving. If the template is not expressive enough, it can miss paths and few candidate invariants can be missed. Also, it is not practical to use InvGen for disjunctive invariants, i.e., it can not handle multiphase loops (loops with conditionals). If it fails to generate any invariant, ProPFA unrolls the loop and perform the failure estimation considering multiple program execution paths.

- **LattE :** LattE (Lattice point Enumeration) [DLDK+12, BBDL+14] is a software for computing integrals of polynomial functions over polytopes. It has the ability to compute the exact volume of polytopes. LattE implements two different integration algorithms for this purpose: 1) Triangulation and 2) Cone decomposition. The first algorithm triangulates the polytope into simplices and integrates over each simplex. On the other hand, the second algorithm integrates over each tangent cone of the polytope. Each tangent cone is triangulated into simple cones for this purpose. The experimental results shown in [BEF00] confirms that triangulating the polytope is better for polytopes that are 'almost simplicial'. Cone decomposition works faster for simple polytopes.

  It may be noted that volume computation of polytopes are restricted to uniform distributions of input variables and linear constraints.

Performance Analysis : The performance of the proposed failure probability estimation algorithm is linear in the number of program paths. Let the number of program input variables, all program variables (temporary + input variables), loops with invariants and success paths in σ be $\mid V_{in} \mid$, $\mid V \mid$, $\mid \mathcal{L}_I \mid$ and $\mid \Pi \mid$ respectively. Also, for each path $\pi_i \in \Pi$, let π_i in Static Single Assignment (SSA) form contain S_i statements and V_i' variables.

- Frama-C: For generating the weakest preconditions, Frama-C is utilized for each path $\pi_i \in \Pi$ with S_i number of program statements as input.

- InvGen: InvGen synthesizes invariants for $\mid \mathcal{L}_I \mid$ simple loops with assertions following the loop. The loop might contain all program variables $\mid V \mid$ in worst case. To generate the initial clauses for InvGen, we spend $\mathcal{O}(\sigma_{LOC} \times FIX\_ITR)$ time in worst case where σ_{LOC} represents Lines of Code in σ.

- LattE: For each program path $\pi_i \in \Pi$ the tool LattE is called with $| P_i | + 2 \times V_{in}$ number of half spaces as input in worst case.

2.2.3 Results

Success Path	Predicates	Success Prob.	Cumulative Fail Prob.
assert $[x \leq 50]^1$; assert $[x + y \leq 100]^2$; $[y := y + 2]^3$; assert $[y \leq 50]^4$; $[x := x + 1]^5$	$(x_0 - 50 \leq 0) \wedge (x_0 + y_0 - 100 \leq 0) \wedge (y_0 - 48 \leq 0)$	0.50000000	0.50000000
assert $[x > 50]^1$; $[x := x - 1]^2$; assert $[x < 60]^3$; $[x := x + 1]^4$; assert $[x + y \leq 100]^5$; $[y := y + 2]^6$;	$(-x_0 + 49 \leq 0) \wedge (x_0 - 60 \leq 0) \wedge (x_0 + y_0 - 100 \leq 0)$	0.09929494	0.40070506
assert $[x > 50]^1$; $[x := x - 1]^2$; assert $[x \geq 60]^3$; $[x := x - 1]^4$; assert $[x + y \leq 100]^5$; $[y := y - 2]^6$;	$(-x_0 + 49 \leq 0) \wedge (-x_0 + 61 \leq 0) \wedge (x_0 + y_0 - 102 \leq 0)$	0.17035527	0.23034979

Table 2.1: Success paths, success probabilities and cumulative failure probability of the program presented in Figure 2.1

In this section, we show in details the generated results for the program in Figure 2.1. First, the program is parsed for program execution paths. The ‘if’ conditions are transferred to asserts while generating predicates for the success paths. Then Frama-C is invoked in order to generate the success predicates. Finally the success probability of the path under consideration is computed by LattE. The path probability is also computed in order to provide the confidence measure. All success paths along with the success probabilities, success probabilities and estimated failure probabilities after considering each program execution path are shown in Table 2.1. The overall failure probability of the program presented in Figure 2.1 is computed as 0.23034979 with 0.98702970 as the confidence measure. The overall failure probability is computed by addition of success probabilities of all the paths and subtracting the result from 1. It may be noted that ProPFA covers all three program execution paths in the example test-case. Hence, theoretically the confidence measure is 1. According to our confidence estimation presented in Equation 2.6, we leverage LattE, the lattice point enumeration tool, for estimating the volume of the polyhedron. Due to imprecision associated with discrete lattice point counting by LattE, practically we estimate the confidence to be 0.98702970.

Program	*op*	*LOC*	$\mathcal{A}$	Failure Prob.
Newton-Raphson	[-2147483648, 2147483647]	∼50	$A \in S$	0.000000001
			$f' \neq 0$	
Trapezoidal	[-32768, 32767]	∼50	$A \in S$	0.000030518
			$b - a > 0$	
Regula-Falsi	[-32768, 32767]	∼50	$A_1 \in S$	0.000169400
			$f(b) - f(a) \neq 0$	
			$A_2 \in L$	
			$i \leq Max$	

Table 2.2: Experimental evaluation of the proposed framework

It may be noted that ProPFA is able to handle piecewise uniform distribution of input variables. In that case it considers all input subregions while computing success probabilities of program execution paths.

We evaluate the proposed failure probability estimation framework over a set of programs from numerical analysis domain in Table 2.2. The operational profile *op* and assertions for each program is also presented in Table 2.2. Let a, b, i, Max and e denote Lower Limit and Upper Limit for integration, iteration number, maximum number of iterations allowed and error respectively. Also, $f(a), f(b)$ denote the evaluation of the function at a and b respectively. It may be noted that the program inputs are restricted to affine input polynomials only in our case. For example, in case of Regula-Falsi method, the function f is a linear function as the non-linear case can not be handled by ProPFA.

2.2.4 Remarks on different aspects of ProPFA

We highlight several important aspects of ProPFA in details.

- **Input Format and Intricacies handled** ProPFA takes input two files. One input file contains the program in C syntax. The assertions are provided in the proper position using the C-key word 'assert'. The other file contains the discrete ranges and probability density functions of all input variables. The input format of this file is almost identical to the LattE half-space representation. The only difference is that the discrete ranges of the variables are provided along with their probability density functions using the keyword 'probabilities' after defining the regions of the particular variable in LattE format. The path extraction algorithm is implemented in C++ with sophisticated data structures and the generated proof obligations of Frama-C are converted into the LattE input format for volume computation using a separate Python script written for this purpose.

- **Invariant Generation** The ranges of the program variables used in the loop are generated by the range analyzer module and are provided as initial templates to InvGen. The range analyzer module is capable of parsing and analyzing programs with integer variables only. Complex program data structures are also omitted. It may be noted that ProPFA is restricted to the limitations of InvGen as discussed in Section 2.2.2. Hence, for these cases, a safe over approximation of failure estimation is achieved.

- **Weakest Precondition Generation** Currently, ProPFA can not handle disjunctive assertions in a generic way. Complex data-structures as well as multi-threading unlike [FPV13] are not yet handled in the present version. Arrays can not be handled as inputs to the program. Also, ProPFA does not handle nested while loops.

- **Failure Probability Computation** Due to the limitations of LattE, ProPFA is limited to linear programs only. Also, ProPFA tackles only uniform distribution for input variables for the same reason. The handling of complex distribution is done in ProPFA in case they are discretized in smaller regions with uniform probability associated with defined probability density functions.

We applied ProPFA over a collection of 20 test programs. The test programs can be found here https://github.com/dlohar/ProPFA.git.

2.3 Summary

In this chapter, we have discussed the proposed theoretical framework for failure probability estimation of imperative programs in C-like syntax. This path based approach tries to explore all possible execution paths and computes success probabilities of each path which in turn is enumerated to estimate failure of the overall program. As is evident, it may not be always feasible to consider all program paths. In that case, a quantitative measure for confidence on the effectiveness of the estimate is provided. This chapter also presents a brief overview of ProPFA, an automatic tool for the proposed approach. Subsequently we employ this failure analysis framework in various domains in the next few chapters.

Chapter 3

Towards Reliability Analysis of Component based Software Systems

Large scale software systems are increasingly invading everyday life. The complexity of such systems mandates assembling off-the-shelf component with pre-specified functionalities in order to distribute the overall design complexity among relatively simpler subsystems. For successful execution of such systems, the components need to be integrated in a reliable manner so that the appropriate functionalities of the components along with their intended interactions are properly exhibited by the integrated system. Hence, reliability analysis of such component based systems at early design stages become a relevant but complicated task. Engineers and researchers use reliability testing in a wide variety of industrial applications to determine how well critical components and materials will perform under different operating conditions.
System reliability is defined as the degree to which a system can perform its intended functionality for a specified period of time. As is evident, failure probability analysis is a major part of reliability analysis research. Decisions can be made that influence the design of products so that they meet customer requirements by estimating failures. Our proposed framework for failure estimation of behavioral specifications can be seamlessly employed for reliability analysis of such component based software systems. According to IEEE 729-1991 standard, software

reliability is defined as follows.

Definition 6. ***Software Reliability**: Software reliability is the probability of failure-free operation of a software for a specified period of time in a specified environment.* □

Our notion of system reliability is the reliability at a certain instant of time (*Point Reliability*). Point Reliability is defined as follows.

Definition 7. ***Point Reliability** (Rel): Point Reliability (Rel) is defined as the probability with which a component based software executes successfully at a particular time instance in a specified environment.* □

In this chapter, we explore a formal modeling of component interactions in terms of reliability and illustrate applicability of the proposed failure estimation framework towards provable measures of reliability. We always refer to the notion of point reliability.

3.1 Related Work in Reliability Analysis

Significant studies have been made on reliability analysis in software domain over the past few decades. There exist a considerable number of reliability models derived by architectural abstraction for reliability analysis of modular softwares. In general, such formal reliability models can be classified under two broad categories: 1)Black box and 2)White box [L^+96, GPT01, Gok07]. Black box model treats the whole software system as monolithic structure and concentrates on the input and output functionalities of the system without analyzing the internal behavior in the testing phase. This method is mostly used to obtain failure data which is analyzed to estimate the present as well as future reliability. However, the white-box model focuses on the components of the system and their interactions for reliability analysis. Three major approaches for reliability analysis of such systems in literature [GPMT01] are presented as follows.

1. **State based models:** In state based model, the software architecture is represented as a control flow graph and the interactions between components are determined by Markov Properties which need to hold true for successful interactions with components [Lit79,GPT01,GT02,Gok07]. The architecture

of software has been modeled as a discrete time Markov chain (DTMC), continuous time Markov chain (CTMC), or semi-Markov process (SMP). State based models can be further classified into composite and hierarchical models. Composite models combine the underlying architecture and the failure behavior in a single model to analyze the reliability of the application. On the other hand, Hierarchical models divide the architectural modeling and reliability analysis into two separate stages and thus is more scalable than the composite models. However, the primary assumptions in the state based model are as follows.

   (a) The reliabilities of the individual components are known or
   (b) It determines component reliabilities using the ENHPP SRGM.

2. **Path-based models:** Path-based models consider software architecture explicitly and assume that components fail independently similar to state based models. These models take into account program execution paths for reliability estimation. The reliability of each path is computed as the product of the individual reliabilities of the components constituting the path for estimating the program level reliability. The paths are found out either experimentally [Sho76, KM97] or analytically [YCA99]. Recently, Hsu and Huang [HH11] proposed an adaptive path-based reliability analysis technique for complex component based software systems. This work proposes reliability computation methods which can handle sequence, branch and loop structures. The aggregate path reliability is computed and used as an approximation of software reliability. In this model also the reliabilities of the individual components are assumed to be known.

3. **Additive models:** Each component reliability can be modeled by non homogeneous Poisson process (NHPP) in this approach. This approach does not explicitly include the architecture of the software. Then, system failure process is also NHPP with the cumulative number of failures and failure intensity function that are the sums of the corresponding functions for each component [XW95, Eve99].

Several tools have been developed for software reliability estimation. However, these existing tools either use failure data during phases of software life cycles to drive one or more of the software reliability growth models or use test coverage measurements in order to estimate reliability. We present a brief overview of the tools available for reliability estimation purpose.

- **CASRE** (Computer Aided Software Reliability Estimation) [LN92] was developed as a software reliability measurement tool that is easier for nonspecialists in software reliability engineering to use than many other currently-available tools. It allows users to determine whether a set of failure data indicates that the system's reliability is increasing during test, whether it is decreasing, or whether there is no discernible trend. CASRE incorporates the mathematical modeling capabilities of the public domain tool SMERFS (Statistical Modeling and Estimation of Reliability Functions for Software) [FS93].

- **SoRel** (Software Reliability analysis and prediction) [KKLM93] provides qualitative and quantitative elements concerning a) the evolution of the reliability in response to the debugging effort, b) the estimation of the number of failures for the following periods of time so as to plan the test effort and the numerical importance of the test and/or maintenance team and c) the prediction of reliability measures such as the mean time to failure, the failure rate or the failure intensity.

- **SHARPE** (Symbolic Hierarchical Automated Reliability and Performance Evaluator) [HSZT00] is a general hierarchical modeling tool that analyzes stochastic models of reliability, availability, performance, and performability. SHARPE models were designed to answer the question: given time-dependent functions that describe the behavior of the components of a system and a description of the structure of the system, what is the behavior of the whole system as a function of time? The functions might be cumulative distribution functions (CDFs) for component failure times, CDFs for task completion times, or the probabilities that components are available at a given time.

- **SREPT** (Software Reliability Estimation and Prediction Tool) [RGT00] implements several software reliability techniques including complexity metrics based techniques used in the pre-test phase, inter failure times-based techniques used during the testing phase, and architecture-based techniques that can be used at all stages in the software's life-cycle. SREPT also has the ability to suggest release times for software based on release criteria, and has techniques that incorporate finite repair times while evaluating software reliability.

- **ReliaSoft's BLOCKSIM** [WLV04] software uses Reliability Block Diagrams (RBD) and Fault Tree Diagrams (FTD) as underlying system models for performing reliability analysis, maintainability analysis, availability analysis, reliability optimization, throughput calculation, resource allocation, life cycle cost estimation and other system level analyses.

A recent approach [FPV13] provides a source level reliability analysis method built on Symbolic Path Finder (SPF). SPF determines failure and success path conditions by executing the source code using symbolic inputs. In [FPV13] a bound is set for unfolding the loop constructs. If the bound is reached and loop condition does not fail, SPF backtracks and generates path conditions for which the success status is unknown. The method is also applicable towards multi-threaded programs.

3.2 Motivation

Most of the existing reliability analysis mechanisms either assume predefined reliability values for individual components or implement statistical software testing methods for estimating reliability of the system. With reliability and safety standards becoming increasingly strict in diverse mission-critical domains like automotive, avionics, nuclear plant control etc., the software development standards in such domains (e.g. DO-178B/C and AUTOSAR recommendations) mandate the use of formal methods to ensure reliability and safety. Hence, in the context of system development, a formal modeling of component interactions in terms of reliability becomes a desirable design attribute. However, exact reliability estimation employing formal methods without any notion of abstraction on the source code

can be expensive and infeasible. Further, component software may be available from third party in a closed source form. In such cases only the interface specifications may be known along with the behavior without any information about the exact implementation details.
The present work provides an approach to estimate reliability of component based software systems by statically analyzing the top level description of the components and their interactions tagged with assertions. The assertions characterize operating regions for components and associated reliabilities. These assertions are termed as *Reliability Assertions*. The assertions, if not satisfied lead to the failure of the overall system under the assumption of fail-stop failure model. Such reliability estimates are particularly useful at an early stage of design when multiple implementation options have been developed for components but the integration phase is yet to begin.

Deriving exact analytical estimation of reliability for the entire component based implementations may become computationally impractical. For such systems, if the system validates a given reliability target (R_T) for a given operational profile satisfying established standards like ISO/IEC, Functional User Requirements (FUR), the design is considered acceptable. Hence, validating a reliability query rather than full system reliability estimation seems to be a more relevant question in the context of requirement driven construction of large scale systems. Before undertaking an actual system integration step, the validation of a reliability requirement significantly saves memory and resources. In case the estimated reliability does not meet some minimum target reliability for a given choice of component options, an alternate choice of component options produced using diverse development methods can be considered and the resulting system level reliability may be checked.

The present work provides a formal approach for reliability estimation by considering success probabilities of all program execution paths along with a confidence measure. As it is not practical to consider all program execution paths because of a possible exponential blowup, a reliability validation method is also proposed. This approach overcomes the intricacies of multiple execution paths by inspecting an adequate set of highly probable execution paths.

For analyzing reliability formally, the system needs to be modeled. All specification models explored so far are either too high level or source code analysis. Formulation of very high level models from source code is not verifiable, also they might not capture intricacies involved in the implementation. Again, for large scale software systems formally analyzing the source code is also not practical. We illustrate our specification model in the subsequent section.

3.3 Component Specification Model

As a part of specification model, the top-level behavioral description σ of a component based software system along with a set of components $\mathcal{C}$, their functionalities and a set of reliability assertions $\mathcal{A}$ (in C syntax) are provided. This top-level software (or its specification) activates components and controls their interaction. For each component $C_i \in \mathcal{C}$, the testing phase has revealed a set of mutually exclusive reliability assertions $A_1, \cdots, A_n \in \mathcal{A}$, characterizing input regions. Using such assertions, test suits are generated which test the component for each operating region. For example, consider an operating region characterized by a reliability assertion A such that a test suit of 1000 test vectors was created satisfying A. Out of the 1000 test runs, black-box testing of the component reports 9 independent failures. Hence, the reliability of the component under the operating region characterized by A is assumed to be 0.991.

Problem Statement : The top-level behavioral description σ along with a set of components $\mathcal{C}$, their functionalities in terms of sequential program statements and a set of reliability assertions $\mathcal{A}$ are provided. For each component $C_i \in \mathcal{C}$, the testing phase has revealed a set of mutually exclusive reliability assertions $A_1, \cdots, A_n \in \mathcal{A}$, characterizing input regions for which the component exhibits reliabilities $r_1, \cdots, r_n$ respectively. Before undertaking an actual system integration step, the objective is to analyze reliability of the overall system specification σ which shall be deploying the components in C given an input *operational profile*, i.e, the set of probabilities with which the inputs may be originating from different specified input regions. It may be noted that the exact implementation details of each component is not available. Instead the functional behavioral description of each component is considered to be provided.

The problem of reliability analysis can be seamlessly mapped to the problem of failure estimation of behavioral specifications presented in the previous chapter. The top level description σ of the component based system is considered as an imperative program represented in C-like syntax. Considering reliability assertions of σ as failure assertions, we can estimate the success probability of the program which brings the notion of reliability estimation at one time instance in this case. Again, as we estimate the success probabilities utilizing the concept of program execution paths, it is also viable to validate a target reliability R_T considering a set of more probable program execution paths.

3.4 Reliability Estimation

Our theoretical framework described in Chapter 2 for failure estimation of behavioral specifications can be seamlessly employed for source level reliability estimation of component based software systems expressed in terms of C program (σ). Let n be the total number of program execution paths in σ. The success probabilities of each path can be enumerated in order to estimate the reliability of σ. This method is useful as it provides provable guarantees for the estimated reliability of the system. Reliability in terms of failure probability ($Pr(\overline{\sigma})$) is defined as follows.

$$Rel = 1 - Pr(\overline{\sigma}) = \sum_{i=1}^{n} Pr(\pi_i) \tag{3.1}$$

Also, a confidence measure is provided with the reliability measure. Confidence is computed as the summation of execution path probabilities explored so far (as discussed in Chapter 2).

Running Example : As an example, let us consider a software system described in Figure 3.1. The system specification comprises seven components $(C_1, C_2, \cdots C_7)$. The assertions before components C_1, C_2, C_5, C_7 imply that the components are tested within this region and their success probabilities are also given as shown in Figure 3.1. For simplicity, we have assumed that components C_3, C_4, C_6 are fully reliable. If the variables x and y are uniformly distributed over the range [0,100] and [0,50] respectively, the reliability of the system is computed as 0.97861. Hence, employing ProPFA, we can compute the failure probability of the program, from which the reliability of the system is computed. This method can be employed for

```
void main(int x, int y)
{
   if (x ≤ 50){
      assert(x + y ≤ 100);
      y = y + 2;
      assert(y ≤ 50);
      x = x + 1;
   else
      x = x − 1;
      if(x < 60)
         x = x + 1;
         assert(x + y ≤ 100);
         y = y + 2;
      else
         x = x − 1;
         assert(x + y ≤ 100);
         y = y − 2;
}
```

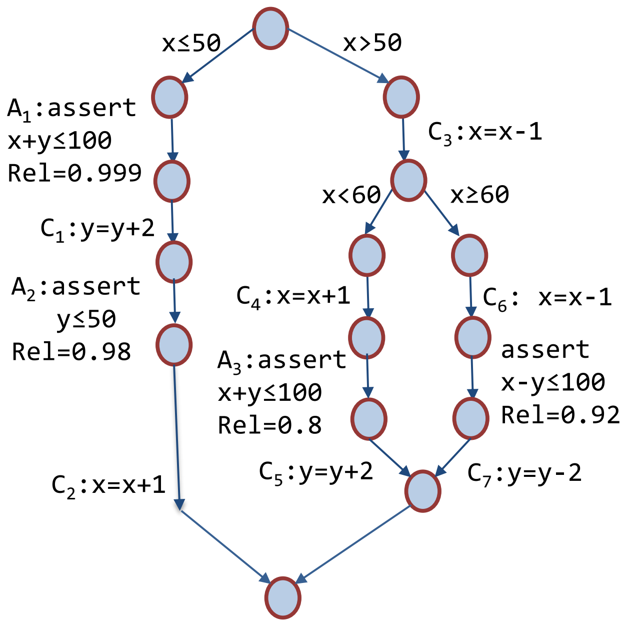

Figure 3.1: Reliability Estimation Example

variables with other distributions in case the variables are independent and can be discretized into smaller intervals. Within that interval, the distributions of the variables can be considered as uniform.

3.5 Reliability Validation using Program Path Enumeration

In Section 3.4, we have discussed a formal approach towards reliability estimation that involves enumeration of success probabilities of all possible program execution paths. Computation of such exact reliability figures may be expensive and even infeasible for complex component based systems with exponentially many possible program execution paths. As it is not possible to explore all paths within a predefined time and memory bound, an estimate is returned with a confidence measure in such cases. The drawback of this method is that it might return a very low reliability value with low confidence, which is futile. Yet a minimal reliability guarantee may often be validated in many cases, by inspecting a small number of highly probable execution paths. This problem of reliability validation is relevant while checking reliability guarantees at early design stages.

In literature, there are two possible methods for validating reliability: 1) Physical testing methods and 2) Analytical estimation methods. The former physically tests each of the components in order to verify whether the system achieves its targeted reliability. The later validates the reliability of a design analytically using detailed reliability models as references. However, for complex systems, exhaustive physical testing of the entire implementation is unrealistic. Deriving reliability bounds using exact analytic methods may also be computationally impractical.

The present work provides a formal approach for reliability validation that overcomes the intricacies of multiple execution paths by inspecting an adequate set of highly probable execution paths. If the enumerated probability of executing a set of paths is c, the contribution in failure probability from unexplored paths is at most $(1 - c)$ by conservatively considering all of them as failed paths. With probability c being sufficiently high, we can (over)-estimate the failure probability of the system which leads to deriving a lower bound on the reliability of the overall system. If the reliability bound thus computed is higher than some target reliability (R_T), the unexplored execution paths need not be considered. Otherwise, the less probable paths are searched to satisfy R_T. If there is no path available for path reliability analysis, the method infers that the system does not maintain its minimal target reliability with a certain confidence measure depending on the path coverage. [1] Our idea of employing an adequate set of paths for reliability validation is philosophically influenced by existing work on static program analysis [SCG13]. However, we provide a more methodical approach while choosing such set of paths. Figure 3.2 depicts our validation procedure. The behavioral description of each component C_i along with the integration logic and reliability assertions are specified in terms of a C program σ. We try to validate whether it meets a desired reliability target R_T. The program σ has a set of n program execution paths $\Pi = \{\pi_1, \pi_2, \cdots, \pi_n\}$. Within the time and memory bound, the method keeps on exploring paths and accumulating their success probabilities. If R_T is validated already, the process stops there, else it iterates over other pos-

[1]It may be noted that it is not practical to consider all program execution paths because of a possible exponential blowup.

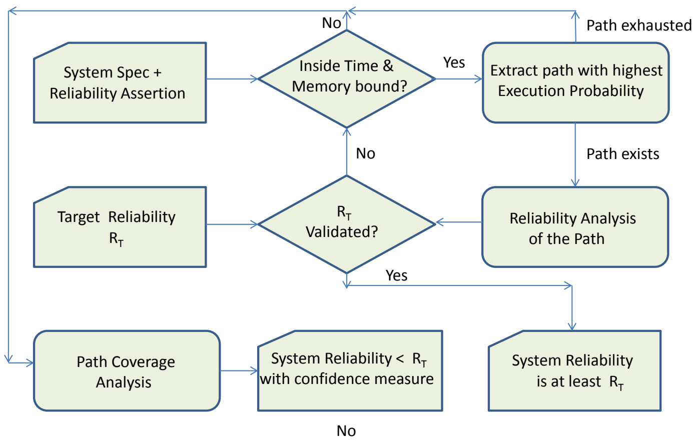
System Spec + Reliability Assertion
Inside Time & Memory bound?
Yes
No
Extract path with highest Execution Probability
Path exhausted
Path exists
No
Target Reliability R_T
R_T Validated?
Reliability Analysis of the Path
Yes
Path Coverage Analysis
System Reliability < R_T with confidence measure
System Reliability is at least R_T
No

Figure 3.2: Framework for Reliability Validation

sible execution paths. Due to predefined bounds it might happen that R_T can be validated with a few execution paths, but these paths are not explored during the process. In order to handle this problem, we propose a heuristic approach to explore an execution path which is more probable at that point of time (locally optimal) and compute the probability of that path. In that way, chances are higher that we could end up actually validating the reliability inside the time bound specified as the more probable paths affect the overall reliability by a higher degree. This greedy heuristic is explained with an example system shown in Figure 3.3. The system needs to ensure a reliability of at least 0.80. The system comprises six components C_1, C_2, C_3, C_4, C_5 and C_6 shown in red in the CFG. Components C_1, C_2, C_5 and C_6 are tagged with reliability assertions while C_3 and C_4 do not have any reliability assertion indicating that these two components execute successfully for every possible inputs. Each reliability assertion is labeled with reliability values determined using test data. The operational profile (op) indicates that the inputs x and y are uniformly distributed in the range $[0, 100]$ and $[-10, 50]$ respectively.

We propose a greedy path selection strategy in order to choose locally optimal program execution paths. The CFG of a given program σ is traversed depthwise and execution paths are uncovered. While traversing the CFG, at every decision node (either an `if-else` or a loop), a local choice of most probable path (computing probability of path condition) is made by the greedy path selection strategy. The strategy extracts the conditional and computes its probability w.r.t. the current path. Based on the probabilistic likelihood, a path (if or else) is chosen

```
function func(int $x$,int $y$)
    if ($x \leq 50$)
        assert($x + y \leq 100$);
        y=y+2;
        assert($y \leq 50$);
        x=x+1;
    else
        if($x < 60$)
            $x = 0$
            while($x < y$)
                $x = x + 1$;
            if($y > 0$)
                assert($x == y$);
                $x = x + 1$
        else
            while($x \leq 65$)
                assert($x + y \leq 150$);
                $x = x - 5$;
    return 0;
```

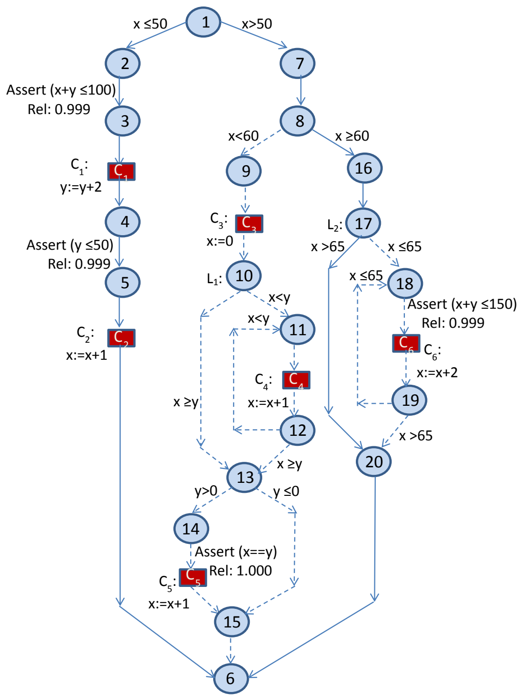

Figure 3.3: Reliability Validation Example

or a loop is unrolled by one level in each pass. The alternative path choices which are not taken at this point of time are enqueued in a priority queue for next-level consideration if required. After extracting the highest probable path chosen locally, failure probability of that path is computed as discussed earlier. If there exists a loop for which the loop invariant is generated, the reliability evaluation is accelerated using the synthesized invariant. In this case, all paths generated by the loop are considered for reliability validation. At node 1 in Figure 3.3, both the 'if' and 'else' branches are equiprobable (probability=0.5). Hence the 'if' branch is chosen and the execution path is explored which is considered for reliability computation next. The reliability of path $\langle 1-2-3-4-5-6 \rangle$ (colored in solid blue) is computed as 0.48236715. It does not validate the target reliability of 0.8. The 'else' branch is then considered for reliability analysis next. At node 8, a locally optimal decision is taken and 'else' branch is considered as the probability of 'else' branch (probability = 0.4) is higher than the 'if' branch (probability = 0.1). After exploring path $\langle 1-7-8-16-17-20-6 \rangle$ (colored in solid blue) the enumerated reliability is estimated as 0.8470045. As this set of paths already guarantees the targeted reliability, the less probable dotted paths are safely left out from the analysis.

It may be noted that computing the most probable global path choices will not scale and goes against the philosophy of the validation method. Hence, the path selection strategy employs local path selection based on probabilistic likelihood. Inside a preset time and memory bound, if the accumulated reliability meets the target, the validation becomes successful. Otherwise it returns the estimated reliability with a confidence measure. The reliability validation algorithm is described below.

Algorithm 3: RELIABILITY_VALIDATION

```
Input: CFG of a program $CFG_\sigma$ annotated with reliability assertions, ranges of input variables with
    probability distributions and a Target Reliability $R_T$.
Output: Returns 'TRUE' if $R_T$ is validated, otherwise returns a reliability estimate ($Rel$) with an associated
    confidence measure($Confidence$).
 1: $\Pi \leftarrow$ Set of program execution paths; $n_{start} \leftarrow$ Start node
 2: $\mathcal{I} \leftarrow$ Set of conditionals; $I_i \in \mathcal{I} \leftarrow i^{th}$ conditional
 3: $Rel = 0$
 4: $S \leftarrow$ INITIALIZE_MAX_PRIORITY_QUEUE()
 5: VISITED$[I_1, \cdots, I_n] \leftarrow$ FALSE
 6: while $Rel \neq R_T$ && $\Pi \neq \Phi$ do
 7:     $\pi_i \leftarrow$ EXTRACT_PATH ($CFG_\sigma, n_{start}$)
 8:     $Rel = Rel+$ COMPUTE_SUCCESS_PROBABILITY ($\pi_i$)
 9:     $Confidence \leftarrow$ EVALUATE_CONFIDENCE($\pi_i$)
10:     $\Pi \leftarrow \Pi \setminus \pi_i$
11:     if $R_T \geq Rel$ then
12:         CONTINUE
13:     end if
14: end while
15: if $Rel \geq R_T$ then
16:     RETURN TRUE
17: else
18:     RETURN ($Rel, Confidence$)
19: end if
```

Algorithm 3 describes the reliability validation procedure. A max priority queue (S) is initialized first (Line 4). A list containing the global markers for all conditionals in the program σ is declared as 'False' (Line 5) initially. These two data structures are utilized while extracting locally optimal paths. Then the path extraction function EXTRACT_PATH is called (Line 7) that takes as input the CFG of σ denoted as CFG_σ and the start node n_{start}. EXTRACT_PATH greedily explores a local optimal path (π_i) which is considered for reliability estimation (Line 8,9). The functions COMPUTE_SUCCESS_PROBABILITY(π_i)

and EVALUATE_CONFIDENCE(π_i) basically utilizes the algorithms described in Chapter 2 in order to estimate the success probability of π_i and confidence measure after considering π_i. This process is continued until R_T is validated or all paths are explored. If R_T is validated in the process, it returns 'TRUE', otherwise it returns the estimated reliability associated with a confidence measure (Line 15-19). Algorithm 4 illustrates the path extraction algorithm in details.

Algorithm 4: EXTRACT_PATH (CFG_σ, s)

```
Input: CFG of a program $CFG_\sigma$ annotated with reliability assertions, start node $s$ and Ranges of input
    variables with probability distributions.
Output: Returns a locally optimal execution path $\pi$.
1: $\sigma_i \leftarrow$ Program segment till conditional $I_i$; $\overline{I_i} \leftarrow$ Else branch of the conditional; $\pi$ initialized to $\Phi$
2: if $s \neq I_n$ then
3:   $\pi \leftarrow \pi \bigcup s$;
4:   $I_i \leftarrow$ TRAVERSE ($CFG_\sigma, s$)
5:   if VISITED[$I_i$] == TRUE then
6:     $I_j \leftarrow$ REMOVE_MAX_ELEMENT()
7:     $\pi \leftarrow \pi \bigcup I_j$
8:     EXTRACT_PATH ($CFG_\sigma, I_j$)
9:   else
10:    VISITED[$I_i$] == TRUE
11:    $P_i \leftarrow$ WP_ANALYSIS($\sigma_i, I_i$)
12:    $Pr(I_i) \leftarrow$ VOLUME_COMPUTATION ($P_i$)
13:    if $Pr(I_i) \geq Pr(\overline{I_i})$ then
14:      INSERT_ITEM($Pr(\overline{I_i}), \overline{I_i}$)
15:      $\pi \leftarrow \pi \bigcup I_i$
16:      EXTRACT_PATH ($CFG_\sigma, I_i$)
17:    else
18:      INSERT_ITEM($Pr(I_i), I_i$)
19:      $p_i \leftarrow \pi \bigcup \overline{I_j}$
20:      EXTRACT_PATH ($CFG_\sigma, \overline{I_i}$)
21:    end if
22:  end if
23: else
24:   RETURN $\pi$
25: end if
```

Algorithm 4 tries to greedily explore an execution path with highest execution probability recursively. The algorithm begins with a start node, traverses till the next branching node I_i (Line 4) in case the start node is not the last conditional (I_n) present in σ. The truth value of VISITED[I_i] indicates the node is already visited. In that case, the priority queue S is dequeued in order to extract the branch with highest probability among all branches explored so far (Line 6). The function REMOVE_MAX_ELEMENT is the standard dequeue function defined for S. Otherwise, the VISITED list entry for this branching node becomes 'TRUE' (Line

10). The probability of the branching node I_i is computed next (Line 11-12) and the path with higher probability is explored (Line 13). The other path condition is stored in the priority queue (Line 14, 18). The function INSERT_ITEM is the standard enqueue operation with the conditional probabilities as comparators. Then it chooses the next start node accordingly and calls itself recursively until the path with highest probability is extracted (Line 24).

Observe that it is not necessary for the most likely path to also be the most reliable. Since the reliability contribution of any path is dependent on both the likelihood of execution as well as the failure probability, a path with higher probability may not always provide higher reliability contribution. In that way, our choice of paths are not necessarily optimal w.r.t. reliability contribution.

Running Example : The behavioral specification model in Figure 3.3 demonstrates the proposed reliability validation algorithm of a component-based system. If R_T is increased to 0.9999, the dotted paths need to be considered for validating R_T. Given the current path $\{1-7-8\}$, the extended path $\{1-7-8-16\}$ is more probable than the extended path $\{1-7-8-9\}$. However, extending the path $\{1-7-8-16\}$ further, the probability of the loop L_2 being executed is less than the execution probability of the path $\{1-7-8-9\}$. Hence $\{1-7-8-9\}$ is considered first according to the greedy path selection strategy.

Subsequently, an invariant is synthesized for loop L_1 and the path $\{1-7-8-\cdots-14-15-6\}$ is considered with the loop invariant. With this set of paths, the current reliability estimate is 0.9370045 which does not validate $R_T = 0.9999$. Therefore loop L_2 is fully unrolled generating all the possible paths within the predefined time and memory bound. The reliability is estimated as 0.9869545 with a confidence measure of 0.99625. This implies that, 99.625% of all the paths are considered in the analysis. Still the reliability target of 0.9999 is not satisfiable with the existing component choices.

3.6 Results

In this section, we show in details the generated results for the program in Figure 3.3. We have assumed two Target Reliability values for this cases and examine whether the target reliabilities are validated. If validated, the program returns with

'TRUE' value along with confidence 1. Else, an estimate measure of reliability is returned with a confidence measure. Execution paths considered following the greedy path selection strategy, Target Reliability, Cumulative Reliability values after considering one execution path, return value and the confidence measure are shown in Table 3.1.

Target Rel	Path	Cumulative Rel	ReturnVal	Confidence
0.80	<1-2-3-4-5-6>	0.48236715	TRUE	1.0000
	<1-7-8-16-17-20-6>	0.8470045		
0.99	<1-2-3-4-5-6>	0.48236715	0.9869545	0.99625
	<1-7-8-16-17-20-6>	0.8470045		
	<1-7-8-· · · -14-15-6>	0.9370045		
	<1-7-8-· · · -19-20-6>	0.9869545		

Table 3.1: Experimental evaluation of Reliability Validation framework

For the next set of experiments, we have considered a Binary Tree implementation as discussed in [FPV13] with only two actions: addition (a) and deletion (d). It is known that this implementation has a bug which is triggered only when the deletion operation is performed in case of non-leaf nodes. The assumption on the action profile states that the addition operation adds elements in the range [0, 10] and delete operation deletes elements in the same range. Figure 3.4 explains the scenario.

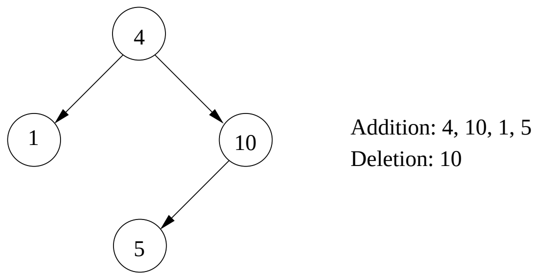

Figure 3.4: Binary Tree Example

Let us assume that 4 addition and 1 deletion operations are performed sequentially. According to the action profile, addition operation adds nodes to the tree by taking elements uniformly from the range [0, 10]. Deletion operation selects elements uniformly from the range [0, 10] and deletes nodes from the tree. In Figure 3.4, the elements 4, 10, 1 and 5 are added according to the binary tree structure. The deletion operation tries to delete element 10. Hence, the action sequence is as follows: <(a, 4), (a, 10), (a, 1), (a, 5), (d, 10)>. Due to the bug in the implementation, this causes failure of the program.

We aim to estimate the reliability of the code for 5 possible sequences of addition and deletion operations. These different sequences are used to form single monolithic programs. It may be noted that no loop is present in the analysis. The failure assertion states that the deleted node does not have any child. If the assertion is violated, the program fails due to the *known* bug in the implementation. Considering the action profiles, Table 3.2 presents the reliability results along with a confidence measure obtained for this binary tree implementation. It also mentions the number of delete and addition operations performed the sequence of the operations. All experiments are carried out in x86_64 architecture based Intel(R) Xeon(R) CPU E5-2667 system. It may be noted that ProPFA does not handle complex data structures. Hence, all complex data structures were reconstructed using simpler data structures like arrays presenting 5 sequences of operations in C.

Action	No.	Sequence	Action Profile	Reliability	Confidence
Delete	3	<d, a, a, d, d>	[0, 10]	0.98889	0.99831
Add	2		[0, 10]		
Delete	2	<a, d, a, a, d>	[0, 10]	0.95174	0.99831
Add	3		[0, 10]		
Delete	0	<a, a, a, a, a>	[0, 10]	0.99997	0.99831
Add	5		[0, 10]		

Table 3.2: Experimental evaluation of Reliability Analysis framework

3.7 Summary

In this chapter, we have explored an application of the theoretical framework proposed in Chapter 2 in the domain of reliability. As is evident, the framework can easily be employed in source level reliability estimation of component based software systems. Based on path coverage, a measure of confidence is also provided with the estimate. For reliability validation we propose a greedy strategy where execution paths are explored based on their likelihood. If the validation becomes successful within a predefined time and memory bound, the algorithm returns 'TRUE'. Else, it estimates the reliability with a certain confidence measure.

Chapter 4

Formal Analysis of Control Software

In this chapter we have addressed two different classes of problems in the domain of control systems, one being the formal analysis of control theoretic properties and the other being safety analysis of embedded control software. Embedded control software implementations are generally required to meet strict control theoretic performance guarantees. While these properties are mathematically verifiable for the underlying control law at the time of design, there does not exist any standard methodology which may formally verify the same over the actual program implementation of the software. We attempt to formally analyze the failure of these control theoretic properties given the exact implementation of the system as C-programs. Other class of problem deals with safety analysis of embedded software. We present a source code based safety analysis of embedded control software systems given the exact behavioral logic specification annotated with the safety requirements in terms of safety assertions.

4.1 Formal Analysis of Control Theoretic Properties

In traditional *sampled data systems* for controlling continuous plants, embedded control software implementations are generally required to meet strict timing and performance guarantees. These control theoretic performance guarantees are mathematically verifiable for the underlying control law at the time of design in an ideal scenario. Unfortunately, the implementation details are not captured here. Hence, it lacks formal assessment of the applied control strategies over the actual program implementation of the software because of software bugs, sensor errors and imprecisions in finite bit-width representation. In this work, we present an approach to perform automated static analysis of controller code for guaranteeing such properties of the controlled physical system.

One of the most important control theoretic properties is stability, the requirement that the physical plant converges to the desired reference behavior under the control actions of the controller in a desired operating region. The theory of control provides a mathematical foundation for stability analysis of dynamical systems. A model of the dynamical system to be controlled is constructed as a set of equations and a feedback controller is designed for providing the controller inputs to the plant based on the plant state observations. The plant + controller model is then analyzed to ensure stability of the system. If the system is stable, the complete implementation in hardware and software is done. However, the mathematical analysis does not guarantee that after implementation, effective control performance offered by the system continues to hold in actual hardware and software with discrete sensing and actuation of physical signals, limited precision arithmetic, potentially faulty sensors and actuators and uncertainty related with external disturbances like environmental noise fluctuations beyond a threshold up to which the control law is robust. In such scenarios, wrong measurements to the controller are provided. Computing control actions over such wrong sensor measurements and actuating the plant can lead to violation of stability. Hence, it makes sense to analyze the risk involved in control software with bounded environmental uncertainty considering the exact implementation of control software in terms of imperative programs. Existing works rely on extensive but non-exhaustive simulation of the code to ensure that the implementation satisfies the stability properties proved for the mathematical model. For transient faults, the control law is adjusted in regard with the fault [GMGB^{+}14]. Also, methods for synthesizing finite precision control laws with provable stability regions have been discussed in [TP06]. However, there does not exist any standard methodology which may formally verify stability of the controlled system over the actual program implementation of the software.

The present work provides a methodology for automatic formal analysis of control system implementations for stability property. In presence of initial disturbance, we perform static program analysis on the software code implementing the plant + controller to compute failure probability of stability criterion in a specific environment given a system level uncertainty profile emanating from sensor noise. Our failure analysis framework can be seamlessly employed for estimating the failure probability of the stability guarantee. If this probability is high, the control designer may then be asked for remedial measures like synthesizing a different control law or restricting the application of the existing control law on a different possible state space region. It may be noted that this approach is not focused on fixed-point control implementations as done in [AMST10] where the authors study the implication of such implementations on the control guarantees.

4.1.1 Theoretical Background

Typically the specification of the dynamics of a discrete time, linear time invariant plant $\Sigma = (A_p, B_p)$ derived from a continuous time plant with sampling period T can be captured by the following equation.

$$x(t+1) = Ax(t) + Bu(t) \tag{4.1}$$

where, the vector $x(t)$ gives the plant state vector at some time instant $t \in \mathcal{N}$ and $u(t)$ defines the control input at the t-th time instance. Thus, for $x(t) \in \mathbb{R}^n$, $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$, where m is the dimension of the control input, we assume a state feedback control action of the following form.

$$u(t) = -Kx(t) \tag{4.2}$$

In this work, we analyze *Exponential Stability Criterion* as one of the control theoretic properties and attempt to estimate the failure probability of it by statically analyzing the plant + controller implementation in C-syntax (σ) leveraging our proposed framework. Following [WA07], the notion of exponential stability is defined subsequently.

Definition 8. ***Exponential Stability***: *A linear time invariant plant* $\Sigma = (A_p, B_p)$ *is* (L, ϵ)*-exponentially stable, with* $L \in \mathbb{N}$ *and* $\epsilon \in (0, 1]$*, if*

$$\frac{||x(t + L)||}{||x(t)||} < \epsilon \tag{4.3}$$

for every $t \in \mathbb{N}$ *and* $x(t) \in \mathbb{R}^n$*, where* $||.||$ *represents the euclidean norm (2-norm).* □

For a control system operating in ideal conditions with no sensor error, it is expected that the plant is exponentially stable. In presence of sensor errors, the plant might not satisfy the stability criterion. It may be noted that, we only investigate the behavior of the control system in the presence of transient faults affecting the sensor readings.

4.1.2 Reliability Estimation of Control Software

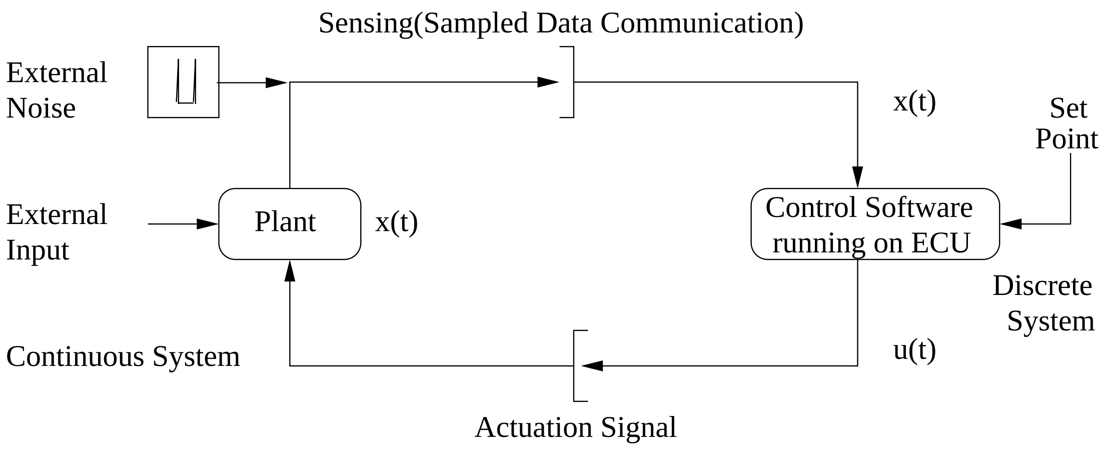

Figure 4.1: Control System Model

We consider an autonomous system Σ as shown in Figure 4.1. Starting from a fixed reference point, the exponential stability requirement states that Σ converges to the desired reference behavior under the control actions of the controller in presence of an initial disturbance d_{init} which is assumed to be uniformly distributed in the range $[0, d_{init}]$. At any control loop iteration, the plant variable x can be disturbed by a high frequency noise spike (E) with an amplitude equi-probable in the range $[-d, d]$. We have considered a specific upper bound (n_d) on the number of transient noise spikes that can be introduced in the system. This defines our notion of environmental uncertainty specification $\mathcal{U}$. The control software implementation of Σ is also provided in terms of a C-program σ. Our goal is to estimate the reliability of the control software implementation (σ) given the environmental uncertainty ($\mathcal{U}$). We have defined reliability of control software as follows.

Definition 9. ***Reliability of Control Software Implementation****: The reliability of a control software implementation $Rel(\sigma)$ is considered as the probability with which the system, in presence of an initial disturbance, satisfies a given control requirement for a specific uncertainty specification $\mathcal{U}$ starting from a fixed reference point.* □

The actual implementation of plant and controller is presented as a C program σ embedded with the required stability criterion as an assertion. Formally, the problem can be stated as follows.

Problem Statement : *We are given an actual C-implementation of plant + controller system (σ) annotated with the stability criterion (L, ϵ) as an assertion. Considering an environmental uncertainty profile $\mathcal{U}$, we intend to compute the reliability of σ after L control iterations in presence of an initial disturbance d_{init}.*

We summarize the steps of our analysis for computing the reliability of a control software implementation as follows.

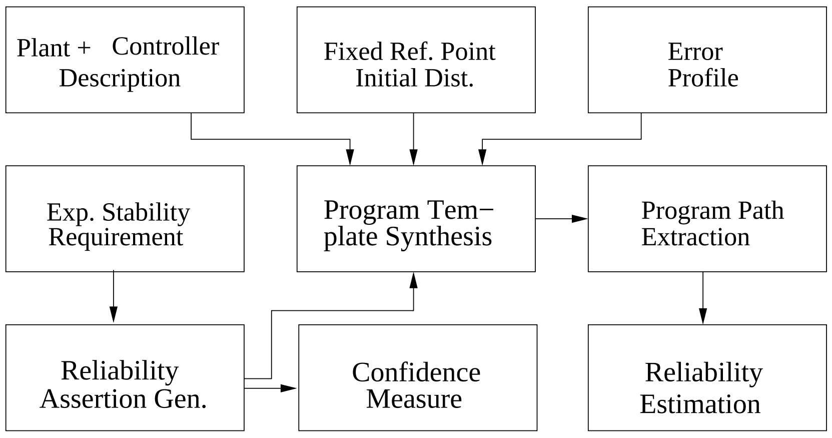
Plant + Controller Description
Fixed Ref. Point Initial Dist.
Error Profile
Exp. Stability Requirement
Program Template Synthesis
Program Path Extraction
Reliability Assertion Gen.
Confidence Measure
Reliability Estimation

Figure 4.2: Reliability Estimation of plant control system

1. Given the instances of σ, $\mathcal{U}$ and d_{init} along with an (L, ϵ) exponential stability criteria, we create n template programs σ_i such that $\forall i, 1 \leq i \leq n$. The

template program σ_i starts from an initial reference point with an initial disturbance d_{init} and captures L consecutive rounds of closed loop control execution in presence of sensing errors.

2. The (L, ϵ) exponential stability is embedded within σ_i as an assertion whose success probability we want to estimate. We derive suitable linearized conditions that safely approximate the quadratic condition.

3. We employ our failure analysis framework ProPFA which extracts program paths from σ_i and computes their weakest pre-conditions (WPs) subject to the conditions derived in the earlier step.

4. ProPFA computes the reliability for σ_i by employing LattE. Eventually the reliability of σ is estimated by accumulating the reliability values of σ_i.

Running Example: We illustrate the proposed approach over a linear double integrator circuit presented in Figure 4.3, which is one of the most fundamental systems in control applications. The output voltages o_1 and o_2 of the two Op-amps are the state variables of the plant and u is the input voltage. The aim of the controller is to control the final output voltage o_2. We intend to estimate the reliability of this system.

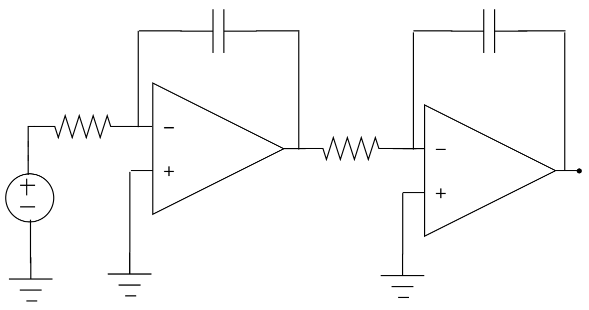

Figure 4.3: Double Integrator circuit

Using a sampling rate of 0.01s the dynamics of the discrete time LTI system is as follows.

$$x_p(t+1) = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} x_p(t) + \begin{bmatrix} -1 \\ 0.5 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 0 & 1 \end{bmatrix} x_p(t) \tag{4.4}$$

A discrete-time Linear Quadratic (LQ) state feedback controller corresponding to this plant model is designed in MATLAB, which regulates the output voltage x_2 to $0V$, whose dynamics is provided here.

$$u(t) = \begin{bmatrix} 0.312 & 0.049 \end{bmatrix} x_c(t)$$

Starting from a fixed reference at $(x_1, x_2) = (0, 3)$ our goal is to estimate failure probability of the stability criterion (L, ϵ) where L is 20 and the value of ϵ is 0.1. The environmental uncertainty specification $\mathcal{U}$ states the following.

- An initial disturbance (d_{init}) uniformly distributed in the range of [0, 10] is added with the plant variable x.
- In L consecutive control loop iterations, the maximum number of sensor errors (n_d) due to transient faults present in the system is 2.
- Sensor error profile (E) is expressed as an uniform distribution function in the interval [-5, 5].

We provide detailed description of the reliability estimation methodology on the double integrator circuit and elaborate on different phases of the approach.

- **Program Template Synthesis:** As stated earlier, the semantics of a loop free template program σ_i captures L consecutive rounds of closed loop control execution. It starts from a fixed reference value of the plant state vector x(0) (at time instant 0). Next, in addition to computing the control law and simulating the plant for L iterations, the template program σ introduces disturbance in the plant variables as sensed by the controller in some choices of control loop iterations according to the uncertainty profile $\mathcal{U}$. The template program σ also contains the assertion validating the (L, ϵ) stability requirement. The other phases of the framework works on the synthesized template program σ and reliability of σ is estimated. This process continues until a maximum of n_d number of disturbances are introduced in all possible combinations of L consecutive control loop iterations. For the double integrator circuit, introducing the initial disturbance and sensor errors at iteration 1 and 19, the template program resembles as follows.

```
int main(float err1, float err2, float dInit){
   float x[20][2];
   float k1=0.312,k2=0.049,u=0; //State feedback control action
   float A1=1,A2=0,A3=-1,A4=1,B1=-1,B2=0.5; //Dynamics of plant
   float x[0][0]=0,x[0][1]=3; //Initial fixed reference
   x[0][1]=x[0][1]+ dInit; //Adding Intial disturbance
   u=-(k1*x[0][0]+k2*x[0][1]); //Controller code
   x[1][0]=A1*x[0][0]+A2*x[0][1]+u*B1; //Plant state at iteration 1
   x[1][1]=A3*x[0][0]+A4*x[0][1]+u*B2;
   x[1][1]=x[1][1]+err1; //Sensor error is introduced
   ..................................
   u=-(k1*x[18][0]+k2*x[18][1]);
   x[19][0]=A1*x[18][0]+A2*x[18][1]+u*B1;
   x[19][1]=A3*x[18][0]+A4*x[18][1]+u*B2;
   x[19][1]=x[19][1]+err2; //Sensor error is introduced
   ..................................
   assert(sqrt((x[20][0]^2 + x[20][1]^2))/sqrt((x[0][0]^2 +
      x[0][1]^2))< 0.1); // 2-norm Exponential stability
}
```

- **Reliability Assertion Generation** The 2-norm exponential stability requirement is added in the template program σ_i. Considering the 2^{nd} norm, the stability criterion presented in Equation 4.3 can be rewritten as follows.

$$x_{t+l}{}^2 \leq \epsilon^2 \times x_t{}^2 \tag{4.5}$$

This non-linear exponential stability equation in terms of an assertion A is embedded at the end of the code as presented in the template program of the double integrator circuit. It may be noted that the static failure analysis framework ProPFA is strictly limited to linear programs. Hence the non linear equation is required to be approximated by a set of linear equations in order to employ ProPFA. As we always start our analysis from a fix reference, the R.H.S. of Equation 4.5 becomes constant. For the double integrator circuit shown in Figure 4.3 the stability equation reduces to the following considering a fixed reference point (0, 3).

$$x[20][0]^2 + x[20][1]^2 < 0.1^2 \times (0^2 + 3^2) \tag{4.6}$$

It may be noted that Equation 4.6 reduces to an equation of a circle for 2-dimensional state vector. This non linear equation is then approximated by a set of triangles as shown in Figure 4.4. Our implementation automatically

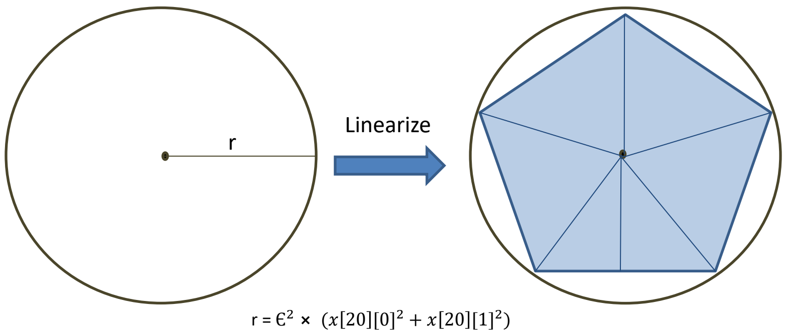

Figure 4.4: Linearization of Stability Criterion

generates the linear constraints for the triangles. The linear equations for each smaller triangular region are then inserted into the template program as assertions and reliability is estimated. Application of the linearization in a recursive manner for handling state spaces of higher dimension is currently under investigation.

- **Confidence Measure:** Our notion of confidence in case of reliability analysis for control software is different from earlier notion as the question of path coverage does not arise here. For control software implementation exponential number of program execution paths are not generated as we do not consider loops. Nonetheless the linearization of the assertion leads to coverage loss for the original failure assertion. Accounting for this approximation, we estimate the confidence considering percentage of error introduced in linearization where the permissible percentage of error is user provided.

- **Program Path Extraction:** This phase is required only if the control system is a switched system. The reliability estimation procedure is applicable for both the sequential and state dependent switched control laws. For switched control system the switching conditions among different controllers are expressed using `if-then-else` constructs thus constituting multiple number of program execution path. Our failure estimation framework extracts each program execution path at a time and computes success probabilities which in turn are accumulated to estimate the reliability of the switched control system. For sequential control systems, there is only one program execution path as in the case of the double integrator circuit shown in Figure 4.3.

- **Reliability Estimation:** Our failure estimation framework ProPFA is employed for estimating success probabilities of the generated template programs in presence of environmental uncertainty $\mathcal{U}$. We generate success predicates for each linearized region and estimate the reliability of that particular region using lattice point counting as discussed in Chapter 2. Then the reliabilities of all regions are accumulated for estimating the overall reliability of the control system in presence of errors.

4.1.3 Experimental Results

Detailed experimental results on the double integrator circuit in Figure 4.3 is presented in this section. We have considered two different error profiles while estimating the reliability. The semantics of the sensor error profile is presented as discrete regions with associated probability values. For example, the error profile ⟨[-5, -2], 0.3; [-2, 2], 0.4; [2, 5], 0.3⟩ suggests that the error can originate uniformly from the range [-5, -2] with probability 0.3, [-2, 2] with probability 0.4 and [2, 5] with probability 0.3 respectively. Given the maximum number of errors allowed as 2 and for two different error profiles the failure probabilities are estimated considering all possible iterations where the errors can be introduced. The linearization errors present in the experimental results shown in Table 4.1 and Table 4.2 are considered to be 5% and 2% respectively. itr1 and itr2 in Table 4.1 and Table 4.2 indicate that error 1 and 2 are introduced at 'itr1' and 'itr2' respectively.

4.1.3. Experimental Results

Error Profile: [-5, 5], 1   Confidence: 0.95

Itr1	1	1	1	1	1	1	1	1	1
Itr2	2	3	5	7	10	13	15	17	20
Rel	0.972683	0.95864	0.935859	0.978393	0.778407	0.957141	0.904232	0.704861	0.594173
Itr1	2	2	2	2	2	2	2	2	3
Itr2	3	5	7	10	13	15	17	20	5
Rel	0.948088	0.92471	0.966894	0.789643	0.957085	0.903799	0.703814	0.594172	0.916547
Itr1	3	3	3	3	3	3	5	5	5
Itr2	7	10	13	15	17	20	7	10	13
Rel	0.957978	0.797684	0.955678	0.902382	0.702066	0.594173	0.959877	0.793811	0.94948
Itr1	5	5	5	**7**	7	7	7	7	10
Itr2	15	17	20	**10**	13	15	17	20	13
Rel	0.896994	0.697053	0.590732	**0.754238**	0.942938	0.890099	0.690519	0.582589	0.938061
Itr1	10	10	10	13	13	13	15	15	17
Itr2	15	17	20	15	17	20	17	20	20
Rel	0.855414	0.680234	0.564649	0.773702	0.651773	0.544503	0.612531	0.522203	0.49288

Error Profile: [-5, -2], 0.4; [-2, 2], 0.2; [2, 5], 0.4   Confidence: 0.95

Itr1	1	1	1	1	1	1	1	1	1
Itr2	2	3	5	7	10	13	15	17	20
Rel	0.922623	0.91862	0.915859	0.948497	0.798417	0.937231	0.883130	0.698486	0.575941
Itr1	2	2	2	2	2	2	2	2	3
Itr2	3	5	7	10	13	15	17	20	5
Rel	0.929480	0.91224	0.941668	0.751896	0.939570	0.891037	0.687038	0.582941	0.901657
Itr1	3	**3**	3	3	3	3	5	5	5
Itr2	7	**10**	13	15	17	20	7	10	13
Rel	0.935798	**0.787684**	0.935678	0.880232	0.680206	0.65173	0.935987	0.789311	0.91448
Itr1	5	5	5	7	7	7	7	7	10
Itr2	15	17	20	10	13	15	17	20	13
Rel	0.886964	0.679053	0.579072	0.735423	0.924238	0.878099	0.680519	0.552589	0.923861
Itr1	10	10	10	13	13	13	15	15	17
Itr2	15	17	20	15	17	20	17	20	20
Rel	0.835414	0.660234	0.574649	0.753702	0.625173	0.514503	0.601231	0.512203	0.49288

Table 4.1: Experimental results for Double Integrator with 5% Linearization Error

Error Profile: [-5, 5], 1 Confidence: 0.98									
Itr1	1	1	1	1	1	1	1	1	1
Itr2	2	3	5	7	10	13	15	17	20
Rel	0.953351	0.931525	0.890268	0.961587	0.758244	0.977501	0.884441	0.610007	0.462605
Itr1	2	2	2	2	2	2	2	2	3
Itr2	3	5	7	10	13	15	17	20	5
Rel	0.917366	0.874887	0.941194	0.774055	0.976258	0.884186	0.609309	0.462605	0.864732
Itr1	3	3	3	3	3	3	5	5	5
Itr2	7	10	13	15	17	20	7	10	13
Rel	0.926699	0.781894	0.974286	0.883506	0.608159	0.462605	0.929651	0.763874	0.97038
Itr1	5	5	5	7	7	7	7	7	10
Itr2	15	17	20	10	13	15	17	20	13
Rel	0.880126	0.605044	0.459419	0.69943	0.969102	0.875716	0.601077	0.452351	0.952183
Itr1	10	10	10	13	13	13	15	15	17
Itr2	15	17	20	15	17	20	17	20	20
Rel	0.823984	0.594877	0.438547	0.678276	0.565081	0.453649	0.527496	0.450372	0.430046
Error Profile: [-5, -2], 0.4; [-2, 2], 0.2; [2, 5] , 0.4 Confidence: 0.98									
Itr1	1	1	1	1	1	1	1	1	1
Itr2	2	3	5	7	10	13	15	17	20
Rel	0.922623	0.91862	0.915859	0.948497	0.798417	0.937231	0.883130	0.698486	0.575941
Itr1	2	2	2	2	2	2	2	2	3
Itr2	3	5	7	10	13	15	17	20	5
Rel	0.929480	0.91224	0.941668	0.751896	0.939570	0.891037	0.687038	0.582941	0.901657
Itr1	3	3	3	3	3	3	5	5	5
Itr2	7	10	13	15	17	20	7	10	13
Rel	0.935798	0.787684	0.935678	0.880232	0.680206	0.65173	0.935987	0.789311	0.91448
Itr1	5	5	5	7	7	7	7	7	10
Itr2	15	17	20	10	13	15	17	20	13
Rel	0.886964	0.679053	0.579072	0.735423	0.924238	0.878099	0.680519	0.552589	0.923861
Itr1	10	10	10	13	13	13	15	15	17
Itr2	15	17	20	15	17	20	17	20	20
Rel	0.835414	0.660234	0.574649	0.753702	0.625173	0.514503	0.601231	0.512203	0.49288

Table 4.2: Experimental results for Double Integrator with 2% Linearization Error

The tabulated results can be interpreted as follows. Let us consider that the highlighted text in Table 4.1. It indicates that with errors injected in two specific iterations (7^{th} and 10^{th}), the overall reliability of the control loop for error profile [-5, 5], 1 is 0.754238. Again, for error profile [-5, -2], 0.4; [-2, 2], 0.2; [2, 5], 0.4 the reliability is estimated as 0.787684 in case two sensor errors are injected at iteration 3 and 10 (2nd highlighted text in Table 4.1). The permissible error introduced during linearization is 5% in both cases. Hence, the confidence measure is 0.95. Table 4.2 provides reliability results for the same system and same set of error profiles with confidence 0.98.

4.2 Safety Analysis of Embedded Control Software Systems

In this section, we have analyzed safety requirements of embedded control software. Safety is a critical factor in the control system construction. As it is literally impossible to be confident about software correctness, increasing use of software controlling the interactions among components in control system raises issues in safety aspect. Also, design of these systems introduces new types of human errors while interacting with or within highly automated systems. Hence, safety and risk analysis become necessary and essential part of such systems' construction. Two distinct research attempts have been made so far for fast and accurate analysis of safety critical systems [LMP06], one of them is failure logic modeling. Fault Propagation and Transformation Notation (FPTN) [FMNP94] is one of the first developments in failure logic modeling for generation and propagation of component failures in the system. However error and system model were separate in this case. To overcome this issue, Fault Propagation and Transformation Calculus (FPTC) [Wal05] was proposed which linked between the error model and system architectural model while maintaining all dependencies. Failure expressions were directly embedded within the components and propagation of failure was described. Probabilistic analysis can also be performed in an extension [GPM09] where probability values can be provided along with the expressions. Component Fault Trees (CFTs) is a graphical representation of the failure logic of components building on the specification used by FPTN. Automated fault tree analysis is provided in [WS78]; however it still required manual intervention in fault tree construction. A few more comprehensive approaches have also been made which automate the synthesis of the analysis model such as fault trees and then analysis of these models. Another strand of research relies on Failure Injection for automatically generating a list of failure configurations that are significant from the safety perspective. This method takes as input a formal design model and extends the intended behavior captured in the model with possible deviations monitored by analysts under conditions of failure i.e. failure modes [JH05]. This approach exploits exhaustive verification capabilities of model checking and provides implicit guarantee of correctness of the results with respect to the design model. However,the shortcoming of the method is the reliance on existence of formal, executable design models.

Nonetheless, we attempt to provide a source code based safety analysis of embedded control software systems. In order to ensure safety, the requirements can be expressed in terms of safety assertions embedded into the execution logic specification presented in terms of an imperative program in C-like syntax. Failure of these assertions leads to failure of the whole system that might endanger human life or cause extensive environmental damage. We aim to compute the failure probability of such behavioral logic to determine the risk associated with the system by employing our failure estimation framework. Such notion of risk is captured in the definition of safety factor is stated as follows.

Definition 10. ***System Safety Factor***: *Safety Factor is defined as the probability with which the system under consideration satisfy all the safety requirements irrespective of whether or not the system conforms to its specification.* □

It may be noted that we are only considering safety analysis at a particular time instance thus ignoring the feedback connection. Our failure analysis framework ProPFA can be seamlessly employed for estimating the System Safety Factor. The glue logic of the system under consideration is presented in terms of a C program (σ) annotated with safety assertions. After extracting one program execution path (π_i) from σ, we compute success probability of that particular path as reported in Chapter 2. Eventually, we extract all program execution paths and enumerate the success probabilities in order to estimate the System Safety Factor. In case all program execution paths are not explored, the imprecision associated with the Safety factor is reported as a confidence measure.

4.2.1 Case Studies

In this section we demonstrate promising results obtained for benchmarks from embedded control software domain, namely non-redundant version of Fly-by-wire (FBW) [Sut68] from avionics domain and Automatic Cruise Control System (ACC) [VE03] from automotive domain.

- **Fly-by-Wire (FBW):** Fly-by-wire (FBW) [Sut68] replaces the conventional manual flight controls of an aircraft with electronic interfaces thus reducing weight and fuel consumption. It also allows to exploit multiple aircraft configurations which increases aerodynamic efficiency and provides better overall performance. The flight control commands (for Roll, Pitch and Yaw) are converted to electronic signals transmitted by wires (hence the fly-by-wire term), and flight control computers determine the exact control commands for the actuators at each control surface of the aircraft. However, this may result in a reduced natural stability, with the aircraft becoming unstable over part of the range of speed and altitude conditions (the flight envelope). FBW systems overcome this by providing high-integrity automatic stabilization of the aircraft to compensate for the loss of natural stability. All these factors provide the pilot with good control and handling characteristics over the whole flight envelope and under all loading conditions.

  The embedded control software (∼1KLoC) for non redundant FBW interacts with several physical components. It periodically samples data from a set of sensors and periodically activates a set of actuators (stabilizers, rudder, elevators etc) inside a control loop. We have considered the ranges of input variables according to the standard specified by Aviation Authorities for the Airbus A320 long haul aircraft. Since we do not have the actual A320 FBW software controller implementation, we use the input specifications for A320 FBW for a generic FBW software specification. These air planes spend their maximum cruising time within the altitude range $[30000 - 40000]$ft. At cruise altitude, the speed of the airplane varies between $[520 - 550]$mph, the air density varies between $[0.1841 - 0.03996]$kg/m^3 and sound velocity is considered as 659.8mph. Airbus A320 surface area is 248 m^2. The input range for sensors which sense the different orientations of the flight can be in the $\pm 90^\circ$ range. The actual pitch angle is only allowed to vary within the range $[0^\circ - 15^\circ]$. We have embedded safety assertions which are required as per recommended standard to model the environmental scenario in which different components execute successfully. Few safety assertions are presented subsequently.

1. Mach number (Ratio of aircraft speed by velocity of sound) should remain in the range $[0.0 - 0.84]$. After 0.84, subsonic aircrafts generate shock waves permanently damaging wings and tail.
2. Angle of Attack (AoA) is limited by 20°. Beyond 20°, the downstream software component fails to activate the lower level controller that performs the final AoA actuation.
3. Load factor safe range is provided to be $[0.9 - 2.5]$. Excessive load factor is avoided because of the possibility to exceed the structural strength of the aircraft and possibly leading to engine failure under stress. Hence this assertion limits the activation of several components.
4. The pitch angle is only allowed to vary in the range of $[0^\circ - 15^\circ]$ to avoid aircraft spin.

We have considered a behavioral logic of the system which includes these constraints in the form of safety assertions. Our baseline system acts within the perfect operational envelope as described previously. For this system we have considered that the pilot provides pitch angle commands within $[0^\circ - 15^\circ]$ with probability 1. If this ideal environment scenario changes, by our lightweight analysis technique, we can compute the resulting change in safety factor as some of the components may not work successfully due to violation of assertions. We have considered the following modifications in standard operational profile. This modified profiles are presented in Table 4.3.

1. Let the pilot provide the pitch angle within the specified practical range with probability 0.999. Hence, with probability 0.001, the pilot can accidentally feed any pitch angle within the range $[(-90)^\circ - (-1)^\circ]$ and $[16^\circ - 90^\circ]$. Using our framework, such human error factors can be naturally captured as probability mass functions distributed uniformly in two partitions of the entire operating region.
2. For the second scenario, let the maximum velocity of the aircraft be 555mph. It will violate the assertion that Mach number should not exceed 0.84 for few cases.

- **Adaptive Cruise Control (ACC):** Adaptive Cruise Control (ACC) [SYM97], a driver support system, controls both speed and headway of the vehicle, slowing the vehicle down when presented with an obstacle and restoring target speed when the obstacle is removed. A typical Automatic Cruise Control (ACC) System consists of a radar, a throttle and a braking unit as its primary components. If a slower moving vehicle is detected by the radar, it slows the vehicle down and controls the time gap between the vehicle and the forward lead vehicle. If the system does not detect any forward vehicle it accelerates the vehicle to its set cruise control speed. This operation allows the vehicle to autonomously slow down and speed up with traffic without intervention from the driver. We have considered certain failure scenarios of this system that constitute the safety assertions in the behavioral description.

  1. ACC can only operate if the vehicle speed is above 12 meters per second (m/s) and current distance between cars is greater than 100 meters (m).
  2. ACC should only be activated when safe distance is greater than 10m.
  3. Though the throttle unit accelerates the speed of the car, the expected speed is not permitted to exceed safe speed.

The results obtained for these cases are presented in Table 4.3. We have considered failure assertions as the safety assertions of component invocation, i.e. every component has a single assertion specifying the fully reliable operational region. From safety perspective, the complement region is considered fully unreliable. In all the above examples, complete path coverage could be achieved.

Program	Operational Profiles (*op*)	*LOC*	#Assertion	Safety Factor
FBW	Recommended *op* of Airbus A320 [Sut68] in cruising altitude	~ 1000	21	1.0000000
FBW	Recommended *op* [Sut68], but velocity range $[520, 555]$mph	~ 1000	21	0.9780000
FBW	Recommended *op* [Sut68], but pitch angle between $[-90°, -1°]$ with Prob. 0.001, between $[0°, 15°]$ with Prob. 0.999	~ 1000	21	0.9981000
ACC	Recommended *op*	~ 200	7	0.9723000

Table 4.3: Evaluation of ProPFA on control softwares

4.3 Summary

This chapter provides a formal approach towards probabilistic formal analysis of control software implementations and a detailed investigation on the behavior of a control system in the face of bounded environmental uncertainty. It also addresses the problem of safety analysis for embedded control softwares and presents promising results obtained for non-redundant version of Fly-by-wire (FBW) from avionics domain and Automatic Cruise Control (ACC) system from automotive domain.

Chapter 5

Conclusion and Future Scope

With increasingly complex and costly implementation methods for software systems, the current software engineering practice of ‘build then test’ is becoming unaffordable. Establishing a minimum reliability guarantee at early design stages thus becomes relevant. In this context, estimation of failure probability assumes major significance. In this thesis we propose a framework for failure estimation when provided with the behavioral description of software systems in terms of imperative programs. We do not aim at estimating exact reliability figures, rather we provide an overestimation of failures so that a provable lower bound on the probability with which the system can execute successfully can be established. A summary of the main contributions of thesis followed by scope of future research are discussed in the subsequent sections.

5.1 Contributions

The major contributions of the thesis are summarized below.

- We present a formal approach towards failure probability estimation of software systems presented in terms of an imperative program in C-like syntax in a specific environment. Unlike other established reliability analysis methods present in literature, the proposed approach utilizes *invariant relations* to reduce program execution paths in presence of simple loops with no failure assertions thus resulting in improved scalability. The imprecision in the estimation process due to the incompleteness of the synthesized invariant as well as time and memory bound is captured as a confidence measure. An automatic path based tool flow ProPFA (Probabilistic Path-based Failure Analyzer) is also designed and experiments are carried out to estimate failure probabilities of numerical programs with well-known failure conditions converted into failure assertions.

- We have demonstrated a provable Point reliability estimation and validation approach for component based software systems utilizing the proposed framework. Additionally, it takes into account the component level testing statistics to provide a scalable semi-formal estimate of reliability. For reliability validation, a greedy strategy is developed which tries to validate a minimum reliability guarantee within the specified time and memory bound. Experimental results are presented over few examples from software systems.

- A formal approach towards the risk estimation of a control system implementation in C-syntax for an uncertain noisy environment is also presented. By utilizing our proposed framework, we have demonstrated that the failure probability of the stability criterion can be estimated given the implementation of the controller plant. This method is also applicable in the domain of control systems.

The work addressed in this thesis proposes a new framework for estimating provable failure probability leveraging program analysis techniques. Further extensions that are possible are listed in the next section.

5.2 Future Scope

The research presented in this thesis can be extended in various directions. The key areas of extension are listed as follows.

- In our failure estimation framework, following are the constraints on environmental variables - 1) the variables are independent and 2) the variables are uniformly distributed in particular ranges. Other complex distributions are discretized into smaller intervals where the distributions can be considered as uniform. Precise failure estimation is possible only in case the discrete intervals are narrow enough that avoid huge under-approximation. Again executing the framework on small discrete intervals increases analysis time. In this research work, the discretization is solely provided by the user. Automated generation of these intervals with optimized granularity of discretization is an immediate extension. Also, the distribution of environmental variables are always not known. Future scope entails handling the imprecise probabilistic inputs for independent as well as dependent environmental variables.

- Our estimation framework deals with programs with simple data structures. Extending these ideas over a complete full-fledged tool-flow for failure estimation of programs with complex data structures is future work.
- We only reduce the number of program execution paths for simple loops without failure assertions. Else, we unroll the loop according to the user provided unrolling factor which again generates huge number of program execution paths for the analysis procedure. In future, more sophisticated failure estimation approach can be established for handling assertions inside loops using techniques like probabilistic abstract interpretation.

There are still more than a few stones left unturned!

Disseminations of the Research

- Debasmita Lohar and Soumyajit Dey, “Integrating formal methods with testing for reliability estimation of component based systems”, in *International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, pp. 33-36, Gaithersburg, MD, USA, 2015.

- Debasmita Lohar, Anudeep Dunaboyina, Dibyendu Das, and Soumyajit Dey, “Failure Estimation of Behavioral Specifications”, in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, Springer International Publishing, pp. 315-322, Beijing, China, 2016.

References

[AMST10] Adolfo Anta, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic verification of control system implementations. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 9–18. ACM, 2010.

[BBDL+14] Velleda Baldoni, Nicole Berline, J De Loera, Brandon Dutra, Matthias Köppe, Stanislav Moreinis, Gregory Pinto, Michele Vergne, and Jianqiu Wu. A userâĂŹs guide for latte integrale v1. 7.2. *Free Software available from http://www. math. ucdavis. edu/˜ latte*, 2014.

[BEF00] Benno Büeler, Andreas Enge, and Komei Fukuda. Exact volume computation for polytopes: a practical study. In *PolytopesâĂŤcombinatorics and computation*, pages 131–154. Springer, 2000.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[Che80] Roger C. Cheung. A user-oriented software reliability model. *IEEE transactions on Software Engineering*, (2), 1980.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[Def92] Patriot Missile Defense. Software problem led to system failure at dhahran, saudi arabia. *US GAO Reports, report no. GAO/IMTEC-92-26*, 1992.

[Den97] JA Denton. Robust, an integrated software reliability tool, 1997.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. 18(8):453–457, 1975.

[DLDK$^+$12] JA De Loera, Brandon Dutra, Matthias Koeppe, Stanislav Moreinis, Gregory Pinto, and Jianqiu Wu. Software for exact integration of polynomials over polyhedra. *ACM Communications in Computer Algebra*, 45(3/4):169–172, 2012.

[DLHTY04] Jesús A De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of symbolic computation*, 38(4):1273–1302, 2004.

[Eve99] William W Everett. Software component reliability analysis. In *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET'99. Proceedings. 1999 IEEE Symposium on*, pages 204–211. IEEE, 1999.

[FMNP94] Peter Fenelon, John A McDermid, Mark Nicolson, and David J Pumfrey. Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review*, 2(1):21–32, 1994.

[FPV13] Antonio Filieri, Corina S Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 622–631. IEEE Press, 2013.

[FS93] William H Farr and Oliver D Smith. Statistical modeling and estimation of reliability functions for software (smerfs) user's guide. revision 3. Technical report, DTIC Document, 1993.

[GMGB$^+$14] Dip Goswami, Daniel Müller-Gritschneder, Twan Basten, Ulf Schlichtmann, and Samarjit Chakraborty. Fault-tolerant embedded control systems for unreliable hardware. In *International Symposium on Integrated Circuits (ISIC)*. IEEE, 2014.

[Gok07] Swapna S Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on dependable and secure computing*, 4(1):32, 2007.

[GPM09] Xiaocheng Ge, Richard F Paige, and John A McDermid. Probabilistic failure propagation and transformation analysis. In *International Conference on Computer Safety, Reliability, and Security*, pages 215–228. Springer, 2009.

[GPMT01] Katerina Goševa-Popstojanova, Aditya P Mathur, and Kishor S Trivedi. Comparison of architecture-based software reliability models. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 22–31. IEEE, 2001.

[GPT01] Katerina Goševa-Popstojanova and Kishor S Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2):179–204, 2001.

[GR09] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *Computer Aided Verification*, pages 634–640. Springer, 2009.

[GT02] Swapna S Gokhale and Kishor S Trivedi. Reliability prediction and sensitivity analysis based on software architecture. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 64–75. IEEE, 2002.

[HH11] Chao-Jung Hsu and Chin-Yu Huang. An adaptive reliability analysis using path testing for complex component-based software systems. *Reliability, IEEE Transactions on*, 60(1):158–170, 2011.

[HSZT00] Christophe Hirel, R Sahner, Xinyu Zang, and K Trivedi. Reliability and performability modeling using sharpe 2000. In *Computer Performance Evaluation. Modelling Techniques and Tools*, pages 345–349. Springer, 2000.

[JH05] Anjali Joshi and Mats PE Heimdahl. Model-based safety analysis of simulink models using scade design verifier. In *International Conference on Computer Safety, Reliability, and Security*, pages 122–135. Springer, 2005.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KKLM93] Karama Kanoun, Mohamed Kaaniche, Jean-Claude Laprie, and Sylvain Metge. Sorel: a tool for reliability growth analysis and prediction from statistical failure data. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 654–659. IEEE, 1993.

[KM97] Saileshwar Krishnamurthy and Aditya P Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 146–155. IEEE, 1997.

[L$^+$96] Michael R Lyu et al. *Handbook of software reliability engineering*, volume 222. IEEE computer society press CA, 1996.

[Lit79] Bev Littlewood. Software reliability model for modular program structure. *Reliability, IEEE Transactions on*, 28(3):241–246, 1979.

[LMP06] O Lisagor, JA McDermid, and DJ Pumfrey. Towards a practicable process for automated safety analysis. In *24th International system safety conference*, volume 596, page 607. Citeseer, 2006.

[LN92] Michael R Lyu and Allen Nikora. Casre: a computer-aided software reliability estimation tool. In *Computer-Aided Software Engineering, 1992. Proceedings., Fifth International Workshop on*, pages 264–275. IEEE, 1992.

[LNF93] Michael R Lyu, Allen P Nikora, and William H Farr. A systematic and comprehensive tool for software reliability modeling and measurement. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 648–653. IEEE, 1993.

[LT93] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[MGL$^+$11] Olfa Mraihi, Wided Ghardallou, Asma Louhichi, Lamia Labed Jilani, Khaled Bsaies, and Ali Mili. Computing preconditions and postconditions of while loops. In *Intl. Conf. on Theoretical Aspects of Comp.*, pages 173–193. 2011.

[NNH99] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 1999.

[PL10] Dillon Pariente and Emmanuel Ledinot. Formal verification of industrial c code using frama-c: a case study. *Formal Verification of Object-Oriented Software*, page 205, 2010.

[RGT00] Srinivasan Ramani, Swapna S Gokhale, and Kishor S Trivedi. Srept: software reliability estimation and prediction tool. *Performance evaluation*, 39(1):37–60, 2000.

[SCG13] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. *ACM SIGPLAN Notices*, 48(6):447–458, 2013.

[Sho76] Martin L Shooman. Structural models for software reliability prediction. In *Proceedings of the 2nd international conference on Software engineering*, pages 268–280. IEEE Computer Society Press, 1976.

[Ste03] Gunter Stein. The practical, physical (and sometimes dangerous) consequences of control must be respected, and the underlying principles must be clearly and well taught. *IEEE Control Systems Magazine*, 272(1708/03), 2003.

[Sut68] JP Sutherland. Fly-by-wire flight control systems. Technical report, DTIC Document, 1968.

[SYM97] Neville A Stanton, M Young, and B McCaulder. Drive-by-wire: the case of driver workload and reclaiming control with adaptive cruise control. *Safety science*, 27(2):149–159, 1997.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[TP06] Weehong Tan and Andrew Packard. Stability region analysis using sum of squares programming. In *2006 American Control Conference*, pages 6–pp. IEEE, 2006.

[VE03] Ardalan Vahidi and Azim Eskandarian. Research advances in intelligent collision avoidance and adaptive cruise control. *IEEE transactions on intelligent transportation systems*, 4(3):143–153, 2003.

[WA07] Gera Weiss and Rajeev Alur. Automata based interfaces for control and scheduling. In *International Workshop on Hybrid Systems: Computation and Control*, pages 601–613. Springer, 2007.

[Wal05] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, 2005.

[WLV04] Wendai Wang, James Loman, and Pantelis Vassiliou. Reliability importance of components in a complex system. In *Reliability and Maintainability, 2004 Annual Symposium-RAMS*, pages 6–11. IEEE, 2004.

[WS78] Richard B Worrell and Desmond W Stack. *A SETS user's manual for the fault tree analyst.* The Commission, 1978.

[XW95] Min Xie and Claes Wohlin. An additive reliability model for the analysis of modular software failure data. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 188–194. IEEE, 1995.

[YCA99] Sherif M Yacoub, Bojan Cukic, and Hany H Ammar. Scenario-based reliability analysis of component-based software. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 22–31. IEEE, 1999.