CIMAT

Centro de Investigación en Matemáticas A.C.

Un Marco de Trabajo basado en Perfiles de Prueba para Evaluar la Confiabilidad de Ensambles de Componentes de Software

TESIS

que para obtener el grado de

Doctor en Ciencias

con orientación en

Ciencias de la Computación

PRESENTA:

Gerardo Padilla Zárate

DIRECTOR DE TESIS

Dr. Carlos Montes de Oca Vázquez

Abril 20 del año 2007 Guanajuato, Gto.

Un Marco de Trabajo basado en Perfiles de Prueba para Evaluar la Confiabilidad de Ensambles de Componentes de Software

A Test Profile Analysis Framework for Assessing the Reliability of Software Component Assemblies

by

GERARDO PADILLA ZÁRATE

DISSERTATION
Presented to the Faculty of
Center for Mathematical Research
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN
Computer Science

Center for Mathematical Research
Guanajuato, Mexico

April, 2007

I would like to dedicate this dissertation to my dear wife, Martha, my mother, Maria Luisa, and my father, Enrique, without whom this work would not have been possible.

ACKNOWLEDGEMENTS

First, thanks to God, for letting me reach this moment. Second, I would like to express my deepest gratitude and appreciation to my research advisor, Dr. Carlos Montes de Oca, for the opportunity he gave me to conduct research under his supervision, for helping me define my research goals, for his support and review, and for valuable guidance during this research not only in academic matters but also in personal and professional matters. I also would like to express my gratitude to Dr. Cuauhtémoc Lemus and Dr. Miguel Serrano for their help, advice, motivation and support during these extraordinary years.

I also want to express my gratitude to all faculty members at CIMAT with whom I had the honor to collaborate or to receive assistance and guidance. Particularly, I would like to thank Dr. Johan Horebeek, Dr. Enrique Villa, and Dr. Luis Escobar.

Part of my doctoral studies was developed at the University of Texas at Dallas. I would like to express my deepest gratitude to Dr. Bastani for being an excellent source of knowledge and inspiration in different areas of life; I also thank to Dr. I-Ling Yen for believing in my work and always sharing a smile. Her optimistic attitude was an inspiration. I also would like to thank Dr. Rodolfo Hernandez for his endless support and guidance during my stay there.

I am also grateful to all my colleagues and friends at CIMAT (Joaquin, Alonso, Justino, Karen, Angel, Giovanni, Arturo … and several more excellent colleagues and friends). Thank you for your help and for creating a positive atmosphere that constantly motivated me to expand my limits. I would like to express my gratitude to the CIMAT, and all the people there that create such kind of research environment.

I also would like to thank to teacher Stephanie Dunbar. Her help, support, patience and motivation were an important element during my studies.

Last but not the least, I would like also to thank my wife Martha, my parents, my brothers, my parents in law and all members of my family for the endless help and support they gave me.

I gratefully acknowledge the financial support for my research provided by CONACYT and CIMAT during my doctoral studies.

“First say to yourself what you would be; and then do what you have to do.”
Epictetus

A Test Profile Analysis Framework for Assessing the Reliability of Software Component Assemblies

Gerardo Padilla Zárate

Center for Mathematical Research CIMAT, 2007

Supervising Professor: Carlos Montes de Oca Vázquez.

ABSTRACT

Software is becoming a critical part of many computer-based systems. The range of systems that require software is growing every day given that software provides a flexible and powerful way to codify complex behaviors. Newer software systems have stricter dependability requirements such as safety, availability, reliability, and integrity.

The early assessment of system attributes, such as performance or reliability, provides information to improve the system design and its development process. Early assessment results in schedule and costs savings, and in system quality and performance improvements. The term "early" means that the assessment is performed before the actual construction of the system.

The objective of this research is to produce an early reliability assessment of a sequential assembly of software components using limited component execution-related information and considering the expected assembly use. Accomplishing this objective provides quantitative means to support design decisions and to improve the component selection process. The execution-related information, called execution traces, is gathered during the component testing process.

This research is motivated by the need of methodological frameworks addressing the early reliability assessment problem in practical settings. Practical settings have characteristics such as the limited information of the testing process, an increasing number of mandatory software dependability requirements, and the popular use of software development methods such as the Component-Based Software Development and Software Architecture-Based Development.

This research has produced a novel methodological framework, named the Test Profile Analysis Framework (TPAF), to produce early reliability assessments of software assemblies. TPAF has four phases: component testing phase, assembly operational specification phase, execution composition phase, and reliability assessment phase. Each phase has artifacts and models which provide practical, semantic, and analytic foundations for the reliability assessment.

During the component testing phase, the Component Test Profiles (CTPs) are built. A CTP includes test cases, test results, a component specification, and a set of execution traces of one component. An execution trace is a sequence of visited abstract nodes in each test case. An abstract node represents blocks of source code statements derived from a control-flow static analysis. Execution traces are dynamically collected with a run-time monitoring infrastructure together with an instrumented executable version of the component.

During the assembly operational specification phase, the Assembly Operational Profile (AOP) is built. AOP is a quantitative assembly use characterization. This artifact is used to weight the reliability assessment. AOP is specified using a UML profile that was created for such purpose and it is referred to as the UML Operational Characterization profile.

During the execution composition phase, the Assembly Test Profile (ATP) is built. ATP is a composite CTP derived from a sequential assembly of CTPs. The composition uses an *ad-hoc* formal composition model that matches test cases from different components to produce the composite CTP. Since CTPs have a limited number of test cases, it is possible to have test cases without matching during the composition process. To address this issue, the formal composition model includes a strategy to select the most appropriate execution prediction technique to predict the missing matches between components. The prediction techniques considered in the framework are artificial neural networks, classification trees, and the best binary matching technique.

During the reliability assessment phase, an *ad-hoc* reliability model combines the information from the ATP with the AOP to assess the reliability of a software assembly. This model is built on the top of the composition execution model and produces an early reliability estimate and the corresponding confidence interval.

The tool TP-Toolset is a partially automated analysis environment to perform the assembly reliability assessment. TP-Toolset has been used in two case studies to test and evaluate the framework. The results show that the framework produces a meaningful early assembly reliability assessment that is consistent with the expected usage and the execution-related information. This framework is unique since it focuses on early reliability assessment of a software assembly using a novel approach to compose limited component testing information, component execution information, and the expected assembly usage.

In addition, TPAF provides the foundation to define methodologies for assessing other sorts of assemblies using other architectural approaches and assessing other attributes that can be analyzed using information from the component test profiles.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

The goal of this chapter is to provide a summary of the entire dissertation. Additionally, this chapter includes the motivation behind this research effort, a description of the problem addressed, and an overview of the proposed solution, results, and future research works.

1.1 Motivation

The software is becoming a critical element in modern systems. The range of applications that require software is growing every day because software provides a flexible and powerful way to codify complex behaviors. It is common to hear that appliances or complex devices contain a large portion of software. We are becoming more dependent on software (Littlewood and Strigini 2000) (Humphrey 2002).

The size and complexity of applications demand more powerful software solutions. These solutions include the fulfillment of requirements about safety, reliability, security, quality, cost, and project related attributes. For example, the automobile software, that controls the automobile engine, may emphasize concerns about safety and reliability attributes rather than security attributes.

Software reliability is an important attribute that quantitatively characterizes the quality of a system (Musa 2004b). Software reliability is the probability of performing a specified function without failure under given conditions for a specified period of time (IEEE 1998). In addition, reliability is a powerful indicator of the quality of the system since it can group several characteristics into a single measurement. In other words, system non-conformities are modeled as failures thereby allowing comprehensive reliability analysis.

Reliability is an attribute that has been studied for more than 30 years. There are several works and techniques to predict or to estimate the reliability of non-software applications (Komi-Sirvio et al., 2003) (Lyu 1996). However, the use of such techniques for software requires considering the nature of software. Software reliability has special characteristics that differentiate it from hardware reliability, such as the origin of failure and the assumptions about the environmental conditions.

First, the origin of failure in non-software products is usually associated with physical degradation (i.e., the randomness of the failure is associated with physical degradation). In contrast, the randomness of the failure in software-based products is associated with the use of the system. For example, the failure in a mechanical device is originated by the physical degradation associated with the mechanical parts. In a software-based product, the failure does not depend on the number of times that a specific software function is activated; it is possible to have an endless operation without a failure if only some input values are used for the function. The failure is originated when other values are used and such values generate a failure.

Second, the assumptions about the environmental conditions of the product, that are provided by the product maker, are similar or covered by the environmental conditions required by the customer or product user. In contrast, for software-based products, the size and complexity of systems is high and it is very difficult to state the same covering assumption. For example, tire manufacturers set their environmental conditions to exceed the conditions that any user may require. However, the case for software is more complex, since the failure may be located not only in boundary values but also in other allowed values. Then, it is not possible to provide the same covering guarantee without having a comprehensive testing process.

Additionally, the assumptions that software reliability models make do not address the complexities of most software, resulting in far less adoption of theory into practice than is possible. Even though reliability models are quantitative, the industry only uses these results qualitatively. An example of this use is testing. When the failure rate falls below X according to a particular reliability model, testing stops, but developers and users cannot assume that the software will always behave with such failure rate less than X in the field (Whittaker and Voas 2000). The reasons behind this situation include the limited capability of testing to find failures, the test data generation complexity, and the high heterogeneity of operational environments.

One approach for estimating the reliability of complex non-software systems is by using the information from the components of the system (i.e., using a divide-and-conquer approach). This approach allows handling complex systems by using high-level models based on components. Then, the reliability analysis is simplified by using information form the components and combining such information into a system model (Shelemyahu and Kenett 1998) (Hoyland and Rausand 2003).

The divide-and-conquer approach has been used widely in software development with success in some aspects including the reuse of software assets and the complexity management reduction. Software reuse has focused mostly on reusing functionalities or behaviors. Relevant non functional attributes, such as performance and reliability, have been recently addressed (Wallnau 2003) (Crnkovic et al., 2002) (Atkinson et al., 2002) (Larsson 2004). The most common terms for such reusable assets are: functions, procedures, libraries, objects, sub-systems, or components. For simplicity, we use the term component to denote all kind of reusable software parts.

Similar to other non-software domains, the early assessment of attributes of the system, such as performance or latency, has been recognized as a powerful strategy to improve the software development process (Cortellessa et al., 2002) (Clements et al., 2002). The term "early" means that the assessment is performed before the actual construction of the system and it is based on evidences of the components that will be used in the system. Such evidences may come from testing results or formal analyses. This kind of assessment allows the evaluation of design alternatives before the actual construction of the system, thereby reducing schedule and project costs. This approach has been promoted by several software-oriented techniques, such as software architecture evaluation methods (Clements et al., 2002) (Svahnberg and Wohlin 2005) (Babar and Gorton 2004), predictable assembly technologies (Stafford et al., 2001) (Hissam and Ivers 2002), and several works on project estimation and attribute prediction (Stutzke 2005) (Fenton and Pfleeger 1998).

The effectiveness of any early attribute assessment method generally depends on factors such as the specific nature of the attribute and the data required for such analysis. For example, the worst- case execution time assessment uses a timing model representing the target processor (Thesing 2006). The inputs for this model include parameters obtained from source code static analysis. Then, the worst-case execution time model combines such parameters for the assessment (Sandberg et al., 2006).

The early reliability assessment of software-based systems has been addressed by different approaches, such as black-box approaches (Goseva-Popstojanova et al., 2005) (Hamlet et al., 2001) (i.e., the internal execution of components is not considered), and white-box approaches (Dolbec and Shepard 1995) (i.e., the use of internal information of every component is considered). Both approaches have their corresponding advantages and disadvantages. For example, white-box approaches generate better assessment of the reliability than black-box approaches. However, white-box approaches are difficult and expensive to implement in medium and large systems given the amount of information required while black-box approaches are simpler and cheaper.

To cope with these challenges, research on techniques, models, and tools to enable early assessment of the reliability of software-based systems are important issues to be explored. Early assessments are powerful and critical parameters reducing the uncertainty in any project. Such importance is recognized in quality process frameworks such as the Capability and Maturity Model Integration (CMMi) which defines a process area focused on Measurement and Analysis (Team 2006). Additionally, early assessments can be used to calibrate more sophisticated estimation models. Early assessment may increase the sources of information complementing people experience assessments and historical records.

Furthermore, using gray-box techniques to characterize the complexity of the system may help to generate more useful reliability estimations, as it was observed by (Whittaker and

Voas 2000). The need for better approaches and techniques to generate meaningful early reliability assessments is the main driver of this work that it is worthy to investigate.

1.2 Problem Description and Research Questions

This section starts defining the basic concepts used in all the chapters. The research context, the problem description, and the research significance of this work are discussed.

1.2.1 Basic Definitions

A *domain* is a set of all possible values that a set may have. When this concept is related to a software function, the input domain is the set of all possible values that the function may use as input. The output domain is the set of all possible output values of the function.

A *component* is a reusable piece of functionality characterized by its input domain and its output domain. In this context, a function or procedure is a component.

An *assembly* is an arrangement of two or more components. The assembly functionality is produced by the composition of all component functionalities.

A *test point* or *test case* is a set of inputs, execution conditions, and expected results (i.e., set of test outputs) developed for a particular objective, such as exercising a particular program path or verifying compliance with a specific requirement.

An *operational profile* is a quantitative characterization of the expected, recorded, or predicted use of a particular component, assembly, or system.

Reliability is defined as the ability of a device or system to perform a required function under stated conditions for a specified period of time; usually the ability is defined as a probability. *Software reliability* is the case when the device or system is a software-based system.

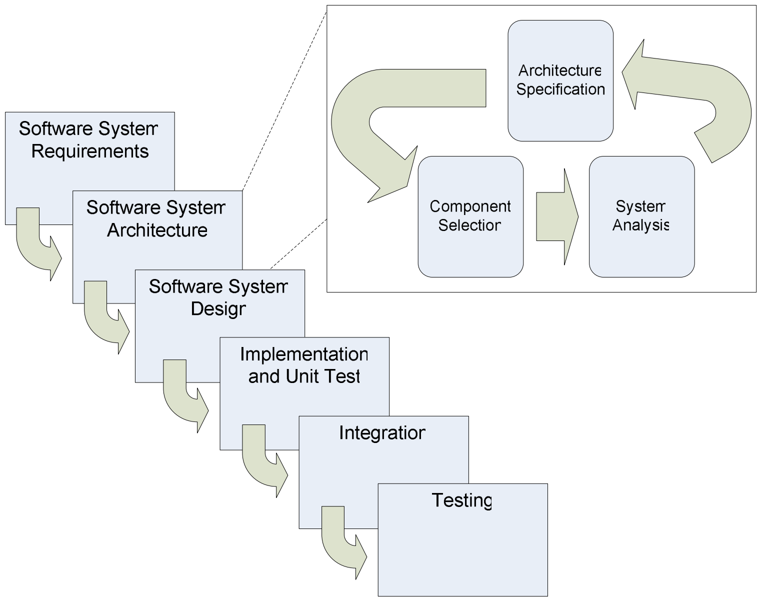
Reliability assessment is a quantitative evaluation of the reliability of a product, system, or portion thereof. Such assessments usually employ mathematical modeling, directly applicable results of tests on the product, failure data, and non-statistical engineering estimates.

An *artifact* is one of many kinds of tangible byproducts produced during the development of software. Examples of artifacts include class diagrams, architecture descriptions, test plans, and software requirement specifications.

A *model* is a theoretical construct or conceptual construct that represents something, with a set of variables and a set of logical and quantitative relationships between them.

1.2.2 Context

This work is targeted toward the early stages of the software development process when early analyses based on models of the system are performed. Examples of such analyses include architectural evaluations, model verifications, and model simulations.



Software System Requirements
Software System Architecture
Software System Design
Implementation and Unit Test
Integration
Testing
Architecture Specification
Component Selection
System Analysis

Figure 1. A Modified Waterfall Software Development Model.

Figure 1 shows a modified version of the waterfall model that includes the Software System Architecture phase. This model is a sequential software development process in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirement analysis, system architecture, design, implementation, integration, and testing (i.e., validation) (Pressman 2005). Other approaches, such as Component-Based Development process, share similarities particularly with the Software System Architecture Phase (Wallnau et al., 2002a).

The early assessment of attributes takes place in the software system architecture phase. The figure expands this phase to show some of the most important activities performed during this phase. First, it is required to have a specification that describes what components are required and which functionality they will provide. Next, a process to find, evaluate, and select components is performed. When there are no candidate components to evaluate, the only alternative is the development of such components. The last phase is the analysis of the resulting model, which is called the system architecture. The analysis is performed with information from component vendors or with the results obtained during the component evaluation.

The research presented in this focuses on the software system architecture phase. The emphasis is on the system analysis activities with special attention to the reliability assessment and the artifacts required for such activity.

1.2.3 Problem Statement and Research Questions

The problem addressed in this work is the reliability assessment of software component assemblies, before integration, using limited testing information from every software component and information about the expected assembly use.

The main research question derived from the previous problem is stated as follows:
It is feasible to produce a meaningful reliability assessment of a software component assembly, before their actual integration, based on (i) component testing information collected in (ii) a limited number of previously executed component test cases and (iii) information about the expected assembly usage?

The answer to this question aims to find a solution that could be integrated within a development process having a software system architecture phase. It is assumed to have reasonable overhead. Item (i) refers to the use of information that may be available from the component providers; item (ii) constrains the scenario with a limited number of executed test cases that represents a practical scenario; and item (iii) refers to additional information that gives a point of reference for any reliability analysis.

1.3 The Test Profile Analysis Framework

This section describes an overview of the solution proposed to answer the research question mentioned in Section 1.2.3. First, the assumptions about the context where the framework can be used are described. This information is complemented with a description about the scope of this work and the rationale behind the framework. Next, an overview of the framework is described. Finally, it is shown how the framework can be used to assess the reliability of a software assembly.

1.3.1 General Assumptions and Scope

The assumptions and constraints defined for this work are classified into three groups: information about the components, information about operational information, and information about the assembly specification.

The assumptions on the component information are:

- The component source code is provided.
- The component testing information may be provided by the component vendor or it is available by other means. This information includes a set of test cases.
- There are no assumptions regarding randomness in the testing data. Thus, it is possible to have bias in the testing information.
- The results of all component test cases are successful. It means that there are no reported failures.
- The behavior of the component is suitable for being modeled as a set of inputs and outputs.

The assumption about the operational information is:

- It is assumed that operational information about the assembly is available.

The assumptions about the assembly specification information are:

- The architecture is specified by denoting the dependencies between component inputs and component outputs.
- The assemblies covered in this thesis are those assemblies that can be modeled as sequential assemblies (i.e., following a data-flow architectural style).
- The assembly requirements are known.

The scope of this research focuses on providing a framework to assess the reliability of software assemblies. The following are out of the scope of this work:

- Generation of random test data.
- Strategies for obtaining operational information. This research focuses on modeling such information rather than means of obtaining the information.

Additional assumptions about technical issues are defined in every chapter. This section only describes the general assumptions defined for the framework.

1.3.2 Rationale

The scenario for the early assessment of a software assembly requires information about the components within the assembly. Using values representing component reliabilities do not provide enough information as it was exposed in Section 1.1. Thus, more information is required to overcome the limited meaning of component reliability values. Such values are typically estimated from evidences collected during the development process. Specifically, test cases and test results are evidences guarantying the conformity of such assembly.

Test cases and test results are not enough information to generate more meaningful reliability estimations. Similar to any benchmark, reliability estimation needs a reference point which is used to compare and measure the test coverage factor. This reference is usually defined as the component specification. The component specification usually includes the description of the functionalities, domains, constrains, protocols, and invariants about the component operation. This information represents a black-box specification since it does not show internal information regarding the implementation.

This research focuses on using a gray-box approach to characterize the component behavior. The reason for this is based on the fact that black-box approaches might not produce accurate estimates and white-box approaches might be infeasible given the size and complexity of the analysis required. This work proposes to use instead of detailed execution information (i.e., white-box approach) an approach based on abstracting the execution (i.e., abstract execution). This abstract execution is modeled as sequences of blocks of instructions rather than sequences of instructions. The criterion for defining such blocks is by grouping instructions defined by distinct execution paths, similarly to the control flow analysis of source code (Allen 1970) (Fenton and Pfleeger 1998). The blocks of instructions are named as *execution nodes* and the sequence of execution nodes is named as *execution trace*.

The component test cases, the component specification, and execution traces are organized and grouped into a new artifact called the *Component Test Profile*. This artifact is an important part of the framework since it provides the required information for performing the reliability assessment. Chapter 4 provides a more detailed description about this artifact and how to build it.

One important constraint in the entire approach is the use of operational information to provide meaningful reliability assessments for a particular user. This constraint is addressed by defining a modeling notation that allows specifying operational information about the assembly. This notation provides a set of modeling constructs to specify operational information using distinct approaches, such as operation-oriented and data-oriented approaches. Chapter 5 provides a more detailed description about the notation and strategy to specify operational information.

Given a set of component test profiles, the next issue is the composition of components that are required for an assembly. This composition uses the information provided for every component (i.e., component test profiles) and focuses only on the required reliability-related information. The composition problem has two issues to consider: first, what kind of information is combined and how this combination is performed; second, how to address the constraint of having only limited testing information.

The first issue is addressed by using the inputs to the reliability model. The key inputs are the execution information recorded during the component testing process (i.e., execution traces). Intuitively, the composition of components could be done by concatenating the execution traces of two components given that there is a match between the output of the first component and an input of the second component. The input-output information is found in the corresponding test cases of every component. Then, the composition mechanism is the concatenation of execution traces constrained by the existence of matching outputs-inputs between the components connections.

The second issue, regarding limited test information, impacts the composition mechanism defined previously. It is not feasible to have a large amount of test cases to find an exact matching between inputs and outputs. The approach to address this issue is to create a prediction model. The nature of such prediction model depends on the nature of the data contained within the component test profiles. Depending on the data, some statistical models, such as artificial neural networks or classification trees (Duda et al., 2000) are capable of functioning as prediction models. Then, the composition mechanism is performed by using predicted execution traces. Chapter 6 provides a more detailed description about the composition mechanism and execution trace prediction models.

The reliability assessment is based on using the outcome of the composition process and the assembly operational information. This assessment uses the execution traces as a source to estimate the probability of failure of every execution node. Then, this information is combined to generate more general estimates than are weighted with the operational information. This estimation generates point and interval estimates of the probability of failure of the assembly.

The reliability model defined in the framework can be also used to assess the reliability of single components under distinct operational profiles. This estimation is possible since the entire model abstracts the assembly as single component.

The ideas described in this section represent a novel strategy to address the problem described in Section 1.2.3. The use of execution traces improves the information collected during the testing process since it provides a simple behavioral representation of the software execution. Moreover, the composition process follows an innovative approach which builds a composite model on top of pieces of information represented by the test cases and execution traces. This model addresses the uncertainty associated with the information by using prediction models. Finally, the approach described in this section is instantiated into a methodological framework suitable for being extended and refined, as described in the next section.

1.3.3 Test Profile Analysis Framework Overview

The previous ideas are organized as a framework based on phases, tasks, artifacts, and models. A phase is a group of tasks that produces a specific set of artifacts (i.e., deliverables). A phase is composed of tasks and artifacts. A task is a well-defined unit of work. An artifact is one of many kinds of tangible products produced during the development of software. A model is a set of conceptual constructs that uses Mathematics or other formal language to describe the behavior, structure, or properties of specific entities or processes.

The relationship between these elements is as follows: a phase is composed of one or more tasks. Every task has a set of one or more required artifacts (i.e, inputs) and a set of produced artifacts (i.e., outputs). A model provides the formal foundations within a specific task and it may impact the semantics of some artifacts. The graphical symbol (i.e., UML stereotype) for every mentioned element is shown in Table 1 (OMG 2007).

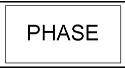
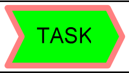
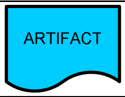
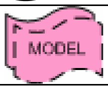
Symbol – Stereotype





Table 1. Framework Element Graphical Representation.

Figure 2 shows a diagram representing the flow of tasks, the required and provided artifacts for every task, and the corresponding foundation models. Dotted arrows denote required artifacts and solid arrows denote provided or produced artifacts. The dotted square-ended line represents the use of a model.

The first phase, the component testing phase, generates the required basic information for the reliability assessment. The task within this phase requires three input artifacts: the source code of the components, the corresponding test cases, and the component specification. The creation of component test profiles requires source code instrumentation (in order to generate the binary version of the instrumented component source code), the execution of the corresponding test cases, the information collected during the test case execution, and the organization of all the resulting information into an artifact called the component test profile.

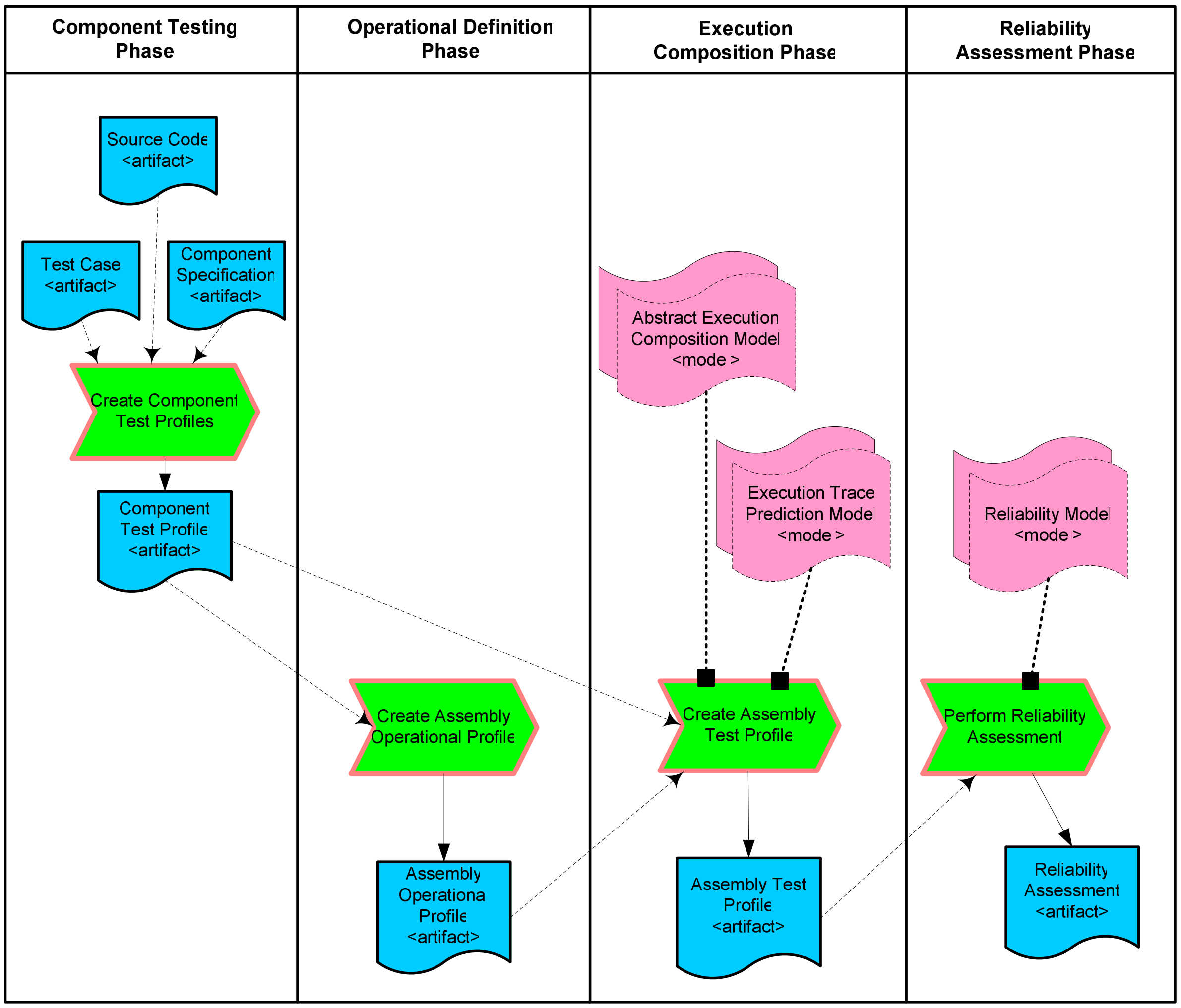

Component Testing Phase
Operational Definition Phase
Execution Composition Phase
Reliability Assessment Phase
Source Code <artifact>
Test Case <artifact>
Component Specification <artifact>
Create Component Test Profiles
Component Test Profile <artifact>
Abstract Execution Composition Model <mode >
Execution Trace Prediction Model <mode >
Reliability Model <mode >
Create Assembly Operational Profile
Create Assembly Test Profile
Perform Reliability Assessment
Assembly Operationa Profile <artifact>
Assembly Test Profile <artifact>
Reliability Assessment <artifact>

Figure 2. Test Profile Analysis Framework.

The second phase, the operational definition phase, models the operational information of the assembly using a standard notation that facilitates reliability assessment. It is not required to have a formal input artifact to the base task but it is assumed that some amount of operational data is provided. For example, the task focuses on modeling such information by generating a consistent model that can be used in subsequent phases. The artifact produced during this phase is called the assembly operational profile. This artifact contains a quantitative model about the expected use of the assembly as well as the minimum information required to specify the assembly structure.

The third phase, the execution composition phase, composes the information provided by the artifacts produced during previous phases. The task within this phase requires as input artifacts all the component test profiles as well as the assembly operational profile. During this task, the information from the component test profiles is combined to generate a single artifact called the assembly test profile. This artifact represents the outcome of combining the information by composing execution information and predicting execution. The models required in this phase are the composition formalism and the execution trace prediction strategy. The composition model is described by a formal model and the prediction model is defined by the best prediction model based on distinct prediction approaches (i.e., regression, artificial neural networks, classification trees, or the best binary matching technique).

The fourth phase, the reliability assessment phase, assesses the reliability by using the information derived from the composition process and the operational information derived from the operational modeling. The input artifacts required for the task are the assembly test profile and the assembly operational profile. The reliability model defined for this task estimates the probability of failure based on occurrence frequencies. These probabilities are combined to estimate the probability of failure of every test case. Finally, the reliability assessment is performed by weighing the operational information with the corresponding probability of failure estimates.

The previous description provides an overview of the most important elements of the framework. Chapter 3 provides a more detailed description and justification of the framework.

1.3.4 Reliability Prediction Using the Test Profile Analysis Framework

The framework has been used to assess the reliability of assemblies. This section describes a basic example to illustrate how the ideas are instantiated and the how reliability assessment is performed. A more a detailed case study is presented in Chapter 8.

The assembly defined for this example is composed of two components. The goal of this assembly is the computation of a mathematical function defined as the logarithm of the square root of a given integer input *X*. The assembly required to achieve this goal is composed of two components, one to compute the square root of an integer value and the second to compute the logarithm of a real value.

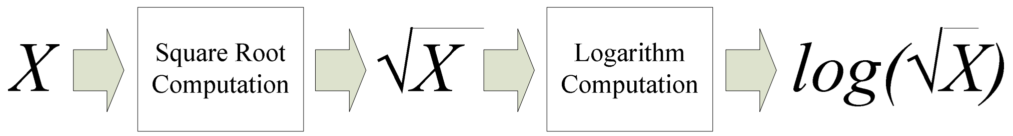


Figure 3. Basic Assembly for Mathematical Computation.

The reliability assessment of this assembly is achieved by performing the corresponding task defined in the framework. It is assumed that a set of component test cases is given.

First, during the component testing phase the corresponding component test profiles are produced. The number of test points for every component is 5000. The instrumentation for these components is simple since the size of such components is small. A fragment of both component test profiles is shown in Table 2.

Square Root Component				
Test Data				
Test Case Number	**Input Value**	**Expected Output Value**	**Execution Trace**	**Test Result**
1	1	1	ABCD	Correct
2	43	6.557438	ABBBBBBBCD	Correct
3	49	7	ABBBBBBBCD	Correct
4	55	7.416198	ABBBBBBBCD	Correct
5	56	7.483315	ABBBBBBBCD	Correct
6	58	7.615773	ABBBBBBBCD	Correct
7	84	9.165152	ABBBBBBBBCD	Correct
8	95	9.746795	ABBBBBBBBCD	Correct
9	106	10.29563	ABBBBBBBBCD	Correct
10	117	10.81665	ABBBBBBBBCD	Correct
11	123	11.09054	ABBBBBBBBCD	Correct
12	130	11.40175	ABBBBBBBBCD	Correct
13	134	11.57584	ABBBBBBBBCD	Correct
14	150	12.24745	ABBBBBBBBCD	Correct
15	154	12.40967	ABBBBBBBBCD	Correct
16	173	13.15295	ABBBBBBBBCD	Correct
17	197	14.03567	ABBBBBBBBCD	Correct
18	220	14.8324	ABBBBBBBBCD	Correct
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

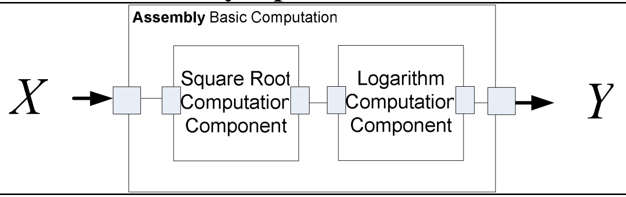
Logarithm Component				
Test Data				
Test Case Number	**Input Value**	**Expected Output Value**	**Execution Trace**	**Test Result**
1	1.90	0.278754	HJJJJJJJK	Correct
2	2.45	0.389166	HJJJJJJJK	Correct
3	4.27	0.630428	HJJJJJJJK	Correct
4	5.00	0.69897	HJJJJJJJK	Correct
5	5.74	0.758912	HJJJJJJJK	Correct
6	6.94	0.841359	HJJJJJJJK	Correct
7	8.06	0.906335	HJJJJJJJK	Correct
8	9.26	0.966611	HJJJJJJJK	Correct
9	10.09	1.003891	HJJJJJJJK	Correct
10	11.05	1.043362	HJJJJJJJK	Correct
11	11.24	1.050766	HJJJJJJJK	Correct
12	11.93	1.07664	HJJJJJJJK	Correct
13	11.99	1.078819	HJJJJJJJK	Correct
14	12.55	1.1	HJJJJJJJK	Correct
15	13.19	1.120245	HJJJJJJJK	Correct
16	13.73	1.137671	HJJJJJJJK	Correct
17	13.91	1.143327	HJJJJJJJK	Correct
18	14.35	1.156852	HJJJJJJJK	Correct
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Table 2. Partial Information from the Component Test Profiles.

Table 2 shows information about the test cases executed in every component. Additionally, there is information about the execution that is stored in the column entitled as "Execution Trace". There are four execution nodes defined in the first component (denoted by the capital letters A, B, C, and D); and four execution nodes defined for the second component (denoted by the capital letters H, I, J, and K)

Second, during the operational definition phase, the operational profile for the system is defined. The operational profile for these components follows a data-oriented definition since the assembly has a simple functionality. Table 3 shows the operational information defined for this assembly. This assembly is usually defined by the system architect during the Software System Architecture Phase using the development process described in Section 1.2.2. The assembly operational profile also includes the specification about the component dependencies and connections.

Assembly Operational Profile


Assembly Basic Computation
X
Square Root Computation Component
Logarithm Computation Component
Y

Sub domain *X*	Lower Value	Upper Value	Probability
1	0	1000	0.2
2	1001	15000	0.4
3	15001	20000	0.3
4	20001	65535	0.1

Table 3. Assembly Operational Profile Defined for the Two-Component Assembly.

Third, during the execution Composition phase, information from the test profiles is used to generate the assembly test profile. For every component, a prediction model is defined by using the test points contained in every component test profile. Next, the composite test points are produced using as reference the test points from the first component in the assembly. The resulting assembly test profile is shown in Table 4. The average prediction error is derived from each prediction model in each component test profile.

Assembly Test Profile			
Test Case Number	**Input Value**	**Execution Trace**	**Average Prediction Error %**
1	1	ABCDHJJJJJJK	20
2	43	ABBBBBBBCDHJJJJJJK	20
3	49	ABBBBBBBCDHJJJJJJK	20
4	55	ABBBBBBBCDHJJJJJJK	20
5	56	ABBBBBBBCDHJJJJJJK	20
.	.	.	.
.	.	.	.
.	.	.	.

Table 4. Derived Assembly Test Profile after Composition.

Finally, during the last phase, the reliability assessment is performed. This assessment is based on the information defined in the assembly test profile as well as the operational information provided in the assembly operational profile. The reliability model computes the probability of failure of every composite test point defined in the assembly test profile as shown in Table 4. These probabilities are combined to estimate the probability of failure of every element defined in the operational profile (in this case the operational profile is defined as a partition over the input domain of the first component). Table 5 shows the probability of failure point estimation for every subdomain. The overall probability of failure is estimated by weighting the probabilities associated with every subdomain. The probability of failure is 1.23×10^{-2}. This point estimation is complemented with an 90% confidence interval estimation (1.13×10^{-2}, 1.35×10^{-2}) using a bootstrap technique (Shelemyahu and Kenett 1998).

Subdomain	Number of Test Points	Subdomain Probability of Failure
1	78	0.007694523
2	1068	0.001155012
3	382	0.002402615
4	3472	8.86668E-05

Table 5. Subdomain Probabilities of Failure.

1.4 Discussion and Conclusions

The previous section describes how the test profile analysis framework addresses the research questions stated in Section 1.2.3. A more detailed experimentation is described in Chapter 8.

For the previous example, the reliability estimates for some subdomain are high because there are a significant number of test points in two subdomains. For those subdomains that have a small number of test points, the reliability estimation is lower.

Compared with other reliability models, our model provides a pessimistic approach. This is due to the consideration of the probability of failure of each component execution node as an internal “abstract component” where the occurrences within each execution trace are considered as one test over such execution node. This situation occurs because any execution node with a low frequency has a low reliability estimation which impacts the overall reliability estimation. Chapter 7 and Chapter 8 provide a detailed discussion about this subject.

Two case studies have been assessed using this framework. All tasks within each phase were performed and the corresponding artifacts were produced. One case is based on an assembly representing a security monitoring system and the second case is based on a small embedded system. The outcomes of these experiments are presented in Chapter 8.

The main contributions derived from answering the research questions are:

- A novel approach for the reliability assessment of component-based software assemblies based on partial information about the component testing process before the actual integration of the system.
- The conception and definition of a new artifact, named as Test Profile. This artifact organizes and consolidates the information obtained from the software testing process. This artifact extends the "traditional" testing information by considering more execution-oriented evidences.
- A reliability prediction model for software assemblies is proposed. This model is based on test profile information and it is weighted with operation information about the assembly. This model provides point and interval estimates.
- A UML profile (OMG 2007) is proposed to model and specify operational information.
- A formal model of execution composition is proposed. This model is formally defined and it is used to compose execution-related information such as execution traces.
- An execution trace prediction methodology is proposed. This methodology is based on selecting the best prediction model for every component based on the information collected in its test profile.
- A prediction technique developed for domains mostly characterized by categorical attributes. This technique is based on finding the Best Binary Matching element for the prediction.
- A Test Profile composition model is proposed. This model is suitable for composing information from component test profiles and generating assembly test profiles.
- An analysis environment, named as TP-Toolset, which performs semi-automatically the assembly reliability assessment.

Future research will address three areas: framework applicability, framework accuracy, and framework extensions. The framework applicability area includes research on topics such as tools leading to the automatic assessment with emphasis on scalability. The framework accuracy area includes topics about the accuracy of prediction and reliability models. In particular, the models will be extended to consider the execution node ordering, execution node failure dependency, and failing test cases. The framework extension area includes the exploration of other architectural styles and the assessment of other attributes, such as performance, based on the same approach.

Our results show that the framework produces a meaningful early assembly reliability assessment. This quantitative information could be used to support assembly design decisions, component selection, or cost/benefit analysis of project-related issues. Major advantages of this framework include the use of the expected usages of the assembly in the reliability assessment, the use of internal component execution information to improve the component characterization for the reliability assessment, the practical assumption about the availability of limited information, the estimations having a confidence interval, and the possibility to automate most of the framework tasks.

This framework is unique in a sense that it addresses the problem of early reliability estimation by combining testing information, execution traces, and operational information.

The framework provides the foundation to define methodologies for assessing other sorts of assemblies using other architectural approaches and assessing other attributes that can be analyzed using information from the component test profiles.

1.5 Dissertation structure

The dissertation is organized in eight chapters. Chapter 2 presents an overview of the state of the art work related with the problem addressed in this dissertation. Chapter 3 provides a general overview of the framework focusing on describing the rationale and a detailed description of the framework and its core elements. Chapters 4 to 7 present the details for every phase defined in the framework. Finally, Chapter 8 validates the entire framework by describing the experiments performed and discussing the results of such experiments.

Figure 4 shows the dissertation structure using as reference the framework diagram. Over the top of the framework, the chapters about the state of the art and the test profile overview are denoted. The remaining chapters focus on issues defined for every phase in the framework and they are denoted at the bottom of every phase.

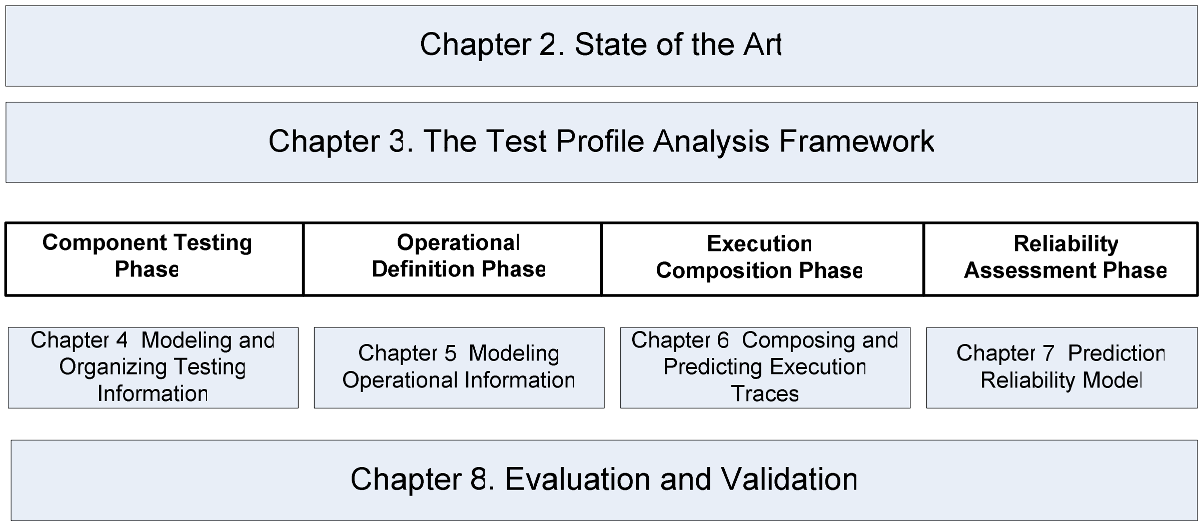

Chapter 2. State of the Art
Chapter 3. The Test Profile Analysis Framework
Component Testing Phase
Operational Definition Phase
Execution Composition Phase
Reliability Assessment Phase
Chapter 4 Modeling and Organizing Testing Information
Chapter 5 Modeling Operational Information
Chapter 6 Composing and Predicting Execution Traces
Chapter 7 Prediction Reliability Model
Chapter 8. Evaluation and Validation

Figure 4. Dissertation Structure.

Chapter 2

State of the Art

This chapter describes the state of the art works related to the problem addressed in this dissertation. The organization of this chapter is as follows: First, works about system assessment (i.e., evaluation, prediction, or estimation) are described. Next, works related to reliability prediction are presented. Then works about testing with emphasis on reliability are described. Finally, works about operational information modeling are explained.

2.1 Introduction

This research is built on past works in several areas. The main ideas come from works about model evaluation, prediction techniques, software testing, software modeling, software architectures, and software reliability estimation.

In particular, estimation is a major area where several research works have been proposed following distinct approaches. The diversity of such approaches include estimation of software size, project time estimation, project risk estimation, cost estimation, product performance estimation, and product quality estimation. An comprehensive overview of such estimation techniques and models is provided by (Stutzke 2005) and (Hughes 2002).

The focus of this research is on software reliability assessment based on components and abstract execution models of the system. Related works along this line include not only works about reliability models but also other similar works developed for other kind of attributes as described in the next section.

2.2 Architectural Component-Based Attribute Assessment

The goal of assessing or predicting properties of software-based systems is an ongoing challenge that has been addressed from different perspectives such as process-oriented or product-oriented approaches. In non-software domains, the use of models to predict, evaluate, or simulate behaviors and attributes is a well-known practice. For example, the prediction of the resulting force for a particular structure is a mature practice in civil engineering (Hibbeler 2001); another example is the prediction of energy consumption for a specific arrangement of components in electrical engineering (Nilsson and Riedel 2004). Similar techniques have been developed for some aspects in software-based systems.

The spectrum of attributes or properties that are amenable to prediction is considerable and it depends on different concerns. A significant effort along this direction has been done by the Software Engineering Institute at Carnegie Mellon University (SEI). SEI has important initiatives addressing the assessment of system attributes[1]. This set of initiatives include the Predictable Assembly from Certifiable Components initiative (PACC) (SEI 2007b), the Software Architecture initiative (SAT) (SEI 2007c), and the Performance-Critical Systems Initiative (PCS) (SEI 2007a). PACC explores how component technology can be extended to predict properties from assemblies. SAT provides effective architecture-centric practices to predict the impact of software architectural decisions on quality attributes such as survivability, security, performance, dependability, and maintainability. PCS focuses on practices and techniques that can be used to predict, evaluate, and manage critical performance attributes of software-intensive systems.

The research developed by the PACC initiative includes the definition of a general purpose reasoning framework called the Prediction Enabled Component Technology (PECT) (Hissam and Ivers 2002). This work proposes a technological framework to predict properties of component-based systems. Figure 5 shows the most important concepts defined for PECT. PECT is composed of two frameworks. The first framework describes the structure and behavior of a component-based system (i.e., Construction Framework); the second defines distinct reasoning schemes (i.e., Reasoning Framework) about distinct properties. This approach includes the use of models to abstract the complexity of the system. Examples of modeling languages to describe assemblies under this approach are the composition languages CL (Hissam et al., 2002b) and CCL (Wallnau and Ivers 2003). These languages have been designed to describe component composition and they have a semantic formal model.

The key parts of PECT are the reasoning frameworks and the abstract component technology. Some important reasoning frameworks have been developed for attribute reasoning such as latency (Stafford et al., 2001) (Hissam et al., 2002b) and performance (Hissam et al., 2005) (Hissam et al., 2004) (Hissam et al., 2002a). Other reasoning frameworks have been developed for the verification of models using techniques such as model checking (Chaki and Nishant 2006) (Chaki 2006) (Ivers and Sharygina 2004). These reasoning engines prove the effectiveness of the PECT approach. However, there are no evidences of works about reliability prediction using PECT.

Other important works have been developed following the spirit of the component-based approach. There are several works addressing the performance assessment such as (Bondarev et al., 2006) (Grassi et al., 2006) (Liu,Yan and Gorton 2005); (Grunske et al., 2005) focuses on reliability analysis of component-based systems. However, it describes reliability assessment using the system structure and assuming the existence of the reliability estimates

[1] This description uses these initiatives as reference since they provide a good structure to describe similar related works.

for the components. (McGregor et al., 2003) proposes an approach to determine system-level properties from components including an initial approach for estimating the reliability of systems. Particularly, the thoughts about achieving trustworthiness prediction provide a good reference of all efforts related with the use of components as technology to build systems (Voas 2003).

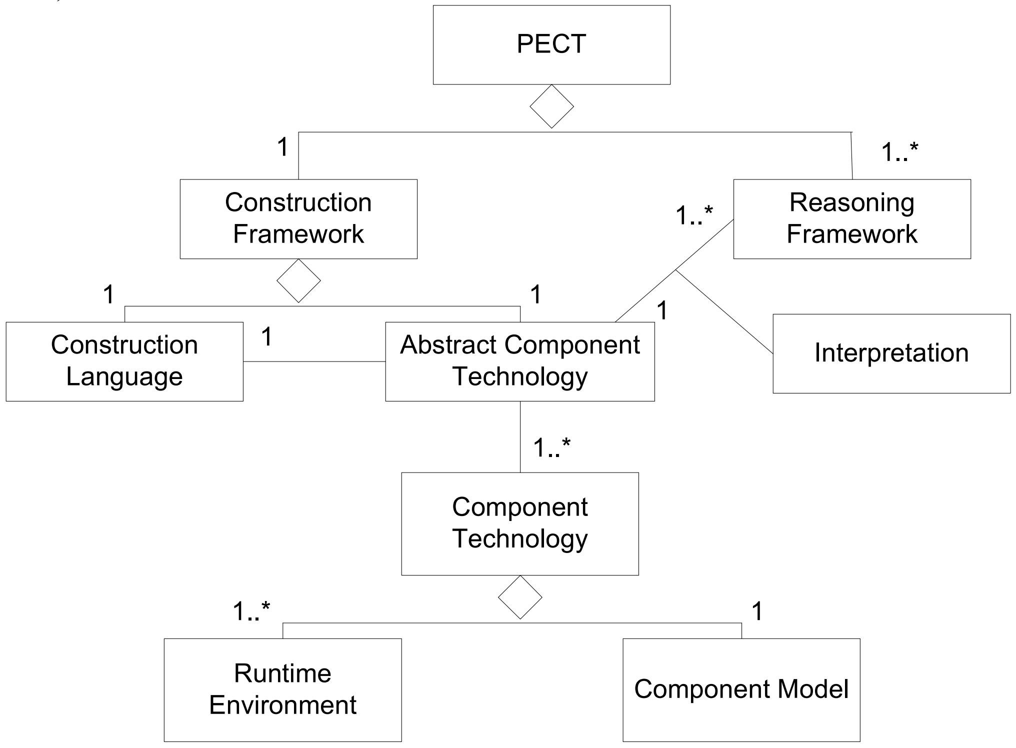


Figure 5. PECT Concepts (Hissam and Ivers 2002).

The Software Architecture initiative (SAT) shares common goals with PACC. Specifically, those related with the evaluation of architectures. This area has an extensive amount of work about defining and using architectural evaluation methods (Clements et al., 2002). The most representative methods developed under this initiative are the Scenario-Based Architecture Analysis Method (SAAM) (Kazman et al., 1996) (Kazman et al., 1994) and the Architecture Tradeoff Analysis Method (ATAM) (Kazman et al., 2004) (Nord et al., 2003). SAAM's goal is to verify basic architectural assumptions and principles against the documents describing the desired properties of an application by using scenarios as a basis for the discussion. ATAM focuses on evaluating the tradeoff between competing attributes. ATAM also considers qualitative analysis heuristics that are derived from an attribute-based architecture style (ABAS) (Klein and Kazman 1999) and are meant to be coarse-grain versions of the kind of analysis that is performed when a precise analytic model of a quality attribute is built. An example of the application of such methodology is found in (Froberg et al., 2006) where the reliability attribute is analyzed and design decisions are selected based on tradeoff analysis.

Some works on architectural evaluation methodologies address distinct attributes. For example, the modifiability attribute is analyzed by ALMA (i.e., Architecture-Level

Modifiability Analysis) (Lassing et al., 2001), performance is addressed by the method PASA (Performance Assessment of Software Architectures) (Williams and Smith 2002), usability analysis is done by (Folmer et al., 2003), potential requirements fulfillment is performed using SBAR (i.e., Scenario-Based Architecture Reengineering) (Bengtsson and Bosch 1998), and maintainability is addressed by ALPSM (i.e., Architecture Level Prediction of Software Maintenance) (Bengtsson and Bosch 1999).

Other works have used tools such as Quality Function Deployment (QFD) to assess quality attributes from any architecture (Bond et al., 2004). A survey of the most representative methods developed until 2002 is described by (Dobrica and Niemela 2002).

The Performance-Critical Systems Initiative (PCS) includes research works on domains such as automotive, aerospace and medical devices. The works around this initiative focuses on dependability issues (Avizienis et al., 2001). The prediction works have focused on attributes of critical systems around dependability by using a modeling notation referred to as Architecture Analysis and Design Language (AADL) (Feiler et al., 2006). AADL is a modeling language that supports early and repeated analyses of a system's architecture with respect to performance-critical properties through an extendable notation, a tool framework, and precisely defined semantics. Examples of property analysis include scalability analysis (Weinstock et al., 2004) (Weinstock and Goodenough 2006), timing, scheduling, and fault tolerance of critical systems (Feiler et al., 2003).

The assessment of component-based systems includes comprehensive frameworks such as PECT or evaluation methodologies such as the architecture evaluation methods. Both approaches state that it is feasible to perform reliability assessment of the system. However, there is a reduced number of works that address this issue based on execution trace information. Additionally, no works address the early reliability assessment using limited execution trace information and composing such information. The research described in this dissertation focuses on the early reliability assessment using execution trace information in a component-based system.

The following section presents an overview of the most significant works on software reliability assessment and the works based on architectures assessing the reliability of a software-based system.

2.3 Software Reliability Assessment

The reliability assessment of software is an activity that started 30 years ago. A general overview of works in this field is described extensively in (Lyu 1996) where prominent researchers describe the advances in the software reliability engineering field. Additionally, (Farr 1983) provides a survey of the models developed at that time. (Singpurwalla and Wilson 1999) provide a good analysis of the most important reliability growth models and a generic Bayesian approach to integrate such models.

The initial works on software reliability were based on the experiences and knowledge gained from the use of techniques for hardware reliability. However, models have evolved and changed over time based on experiences and computer technological advances. The most important works about software reliability were developed during the 70's and 80's when a considerable number of works addressing the reliability estimation of software based systems were published. A basic classification of such works is provided by (Komi-Sirvio et al., 2003). In this work all software reliability models are classified into four major groups including reliability growth models, input domain models (or data domain models), early prediction models, and architectural-structural models.

2.3.1 Software Reliability Growth Models

These models derive reliability predictions from failure data and failure trends. They capture failure behavior during testing and extrapolate it to behavior during operation. The most basic classification of such models divides the models in two groups: models that use the time between successive failures (i.e., Type I models) and models that use the number of failures up to a given time (i.e., Type II models). The most representative works for Type I models include the Jelinski-Moranda Model (Jelinski and Moranda 1972).

The Jelinski-Moranda model is the earliest reliability model. The elapsed time is taken to follow an exponential distribution with a parameter that is proportional to the number of remaining faults[2] in the software. This model assumes a perfect fault correction process (i.e., a correction does not introduce new faults). The failure distribution function defined for this model is based on the exponential distribution

$$F_i(t_i) = 1 - e^{-\lambda_i t_i}$$

with

$$\lambda_i = (N - i + 1)\phi$$

Here N is the initial number of faults, and ϕ is the contribution of each fault to the overall failure rate. This model assumes that fault detection and correction begin when a system contains N faults, and all fixes do not introduce new faults. Additionally, the model assumes that all faults have the same rate.

Subsequent models, such as the work proposed by (Littlewood and Verral 1973), attempt to present a more realistic finite fault model by treating the hazard rates as independent random variables. These rates are assumed to have a gamma distribution with parameters (α, β). This model represents the reliability impact improvement produced by additional testing. Similarly, (Schick and Wolverton 1978) extended the basic Jelinski-Moranda model by considering the contribution of every fault to the failure rate. This model assumes that the conditional rate is proportional to both the number of remaining faults and the elapsed time since the last failure. From another perspective, (Schneidewind 1975) proposes a model that is tailored for scenarios depending on the importance of the available information. (Langberg

[2] A fault is a defect in the code that can be the cause of one or more failures.

and Singpurwalla 1985) proposes a model which provides an alternative non-bug counting perspective of the software failure process that is represented by considering not only the faulty inputs but also the correct inputs during the testing process.

The Musa model (Musa 1971) has had the widest distribution among the software reliability models. One particular characteristic is the use of execution time rather than calendar time. The cumulative number of failures follows a Poisson process. (Everett 1992) proposes an extended execution time model that considers non-uniform execution of instructions.

The Goel-Okumoto model (Goel and Okumoto 1979) takes the number of faults per unit of time as an independent Poisson random variable. Since each fault has no memory, this model is modeled by what is called a non-homogeneous Poisson process (NHPP). The Goel-Okumoto is the NHPP variant of the Jelinski-Moranda.

The Ohba model (or hyperexponential model) (Ohba 1984) is an extension of the classical exponential models by considering that different sections of the software experience an exponential failure rate. The rates vary over these sections according to their different natures.

Other works have used distinct distributions addressing the failure modeling problem. Examples of such works include the models based on Weibull and Gamma Distribution, Logarithmic Poisson, among others.

The Bayesian approach has been proposed in models such as the Littlewood-Verral model (Littlewood and Verral 1973). This model considers two important issues: the fault correction and the fault generation for the reliability estimation. In this model it is possible to have a less reliable system after some problem correction. The prior distribution of this model is assumed to be a gamma distribution.

The models discussed in this section correspond to the classical reliability models that are usually discussed. However, there is a considerable number of works extending, improving, applying, and criticizing the classical reliability models. Our interest focuses on reliability models that can be defined using evidences about the absence of failures rather than the failure times or other similar measurement units.

2.3.2 Input Domain Models

Input Domain Models estimate the system reliability based strictly on the number of successful runs observed and compared to the total number of runs made. This category also includes those models that consider test inputs chosen according to probability distributions of anticipated operational usage (i.e., by using operational profiles).

The basic model of such works is based on considering a binomial experiment, where the estimator for the reliability of the system is defined as (Farr 1983):

$$\hat{R} = \frac{S}{N}$$

Here *S* represents the number of successful test cases in *N* executed test cases.

(Corcoran et al., 1964) proposes a model that that is a generalization of the previous binomial model. This model assumes that there is a known classification of errors and their corresponding probability of being observed. The model is defined as the basic binomial model with an additional term that weights the probability of an error type with the number of errors of such type.

(Hecht 1977) proposes an extension to this model including the average number of machine instructions found in every test case. This model addressed in some extent the issue of the system source code size.

(Brown and Lipow 1975) proposes a model that modifies the basic estimator by including the notion of homogeneous faulty regions. These regions represent a set of inputs that generate the same faults. Then, the estimation is weighted by using operational information.

Other models combine not only the results of a given set of test cases, but they also consider the input space, such as the model proposed by (Nelson 1978). This model considers the sampling probabilities associated with every point (i.e., operational information). (Thayer et al., 1976) extends the Nelson model by partitioning the input space into disjoint regions and associating a probability with every region. The difference between the Nelson and the Brown regions relies on the criterion used for defining such regions. Nelson partitioning is based on building logic paths and Brown partitioning is based on defining regions that are homogeneous with respect to error generation.

(Sugiura et al., 1974) proposes a theoretical model to determine the remaining faults in the system based on using the binomial model. (Elliot et al., 1978) proposes a testing procedure to determine whether a system has reached a given reliability based on the basic model described at the beginning of this section (i.e., binomial model).

(LaPadula 1975) (Sukert 1976) derived a regression model based on the success/failure counts observed at various stages during software testing.

In a relatively recent work, (Tian 1995) proposes a model that combines the approaches from input domain and reliability growth models. This model uses a technique called Tree-Based Modeling (Clark and Pregibon 1991). This technique attempts to establish predictive relations through recursive partitioning. This model uses different stages to estimate the reliability by using an input domain approach.

2.3.3 Early Prediction Models

The models discussed above focus on information that is provided in later phases of the development process. This section describes models that use the characteristics of the software as well as the characteristics of the software development process. The development process characteristics are used to extrapolate the software operational behavior.

Examples of models in this class include the model proposed by (Gaffney and Pietrolewicz 1990) (Gaffney and Davis 1988). This model uses information collected during the reviews performed in the requirements, design, and implementation development phases. This information includes the fault statistics that are used to predict the reliability of the system.

(Smidts et al., 1996) proposes a reliability model based on a systematic identification of software process failure modes and their likelihoods. This model uses a Bayesian approach where the prior knowledge is provided by failure process identification.

Other works are those developed by the Rome Laboratory (Rome-Laboratory 1992). These works include models to predict fault density that are transformed into reliability measures such as failure rates. Most recent works use the measurement framework proposed by the Personal Software Process (Humphrey 1994) as source of information for the reliability assessment of software systems (Davila-Nicanor and Mejia-Alvarez 2004).

2.3.4 Architectural-Structural Models

Architectural-structural models derive reliability estimates by combining estimates from different modules (components) of the software. Their emphasis is on the architecture/structure of the software. A classification of such models is based on the work of (Goseva-Popstojanova and Trivedi 2003) that classifies the models into three groups: state-based models, path-based models, and additive models.

State-based Models

State-based models use graphs to represent the structure of the system (i.e., the architecture of the system). Graphs represent the structure, decision points, and branches in a program source code. These models usually use different models of Markov Chains, such as Discrete Time Markov Chain, Continuous Tie Markov Chain, or Semi Markov Processes. Based on such models, the reliability assessment is performed by following a composite or hierarchical approach (Gokhale and Trivedi 1997). The composite approach combines the structural description (i.e., architecture) with the reliability-related information into a composite model which is then analyzed to predict the reliability of the system. The hierarchical approach first analyzes the composite behavior of the architecture and then superimposes the reliability-related information to perform the reliability assessment. One important assumption in the Markov approach is the independence of failure between modules or components.

The Littlewood model is the earliest model following the architectural approach (Littlewood 1975). This model assumes that software can be characterized by an irreducible Continuous

Time Markov Chain. This model considers also interface failures (i.e., intra-component communication). Failures occur according to a Poisson process in each component. Laprie model (Laprie and Kanoun 1996) is the particular case when only component failures are considered. A generalization of the Littlewood model is proposed in (Littlewood 1985). This generalization assumes that the architecture of the system can be described by an irreducible Semi-Markov Process. Then, the composite model focuses on the total number of failures in a given time interval.

Other works use the Markov Chain to model the architecture of the system (Hui-Qun et al., 2001) for software applications. (Zhu and Gao 2003) proposes a work that uses the input domain partition of a system to derive the usage transitions required for the Markov Chain. (Rodrigues et al., 2005) used a formal model based on a probabilistic transition system that was derived for specifications of concurrent systems.

The Cheung Model (Cheung 1980) is one of the earliest models following a composite approach. This model addressed the prediction of a system by defining an absorbing Markov Chain representing the control transfer among modules[3]. The reliability is estimated by computing the probability of reaching an absorbing state representing the correct system termination. Some projects have used this model to assess the reliability of heterogeneous architectures (i.e., distinct architectural styles) such as the work proposed by (Wang et al., 1999). This work reduces the combination of architectures to a Markov chain model. In the same direction, (Reussner et al., 2003) provides an extension of the model proposed by Cheung by using an architectural description language named as RADL. This work combines the use of an extended automaton to describe interface protocols that are combined to derive the composite model.

The model proposed by Kubat (Kubat 1989) uses information about the execution time of each module in the system following the hierarchical approach. This model incorporates the notion of task which is associated with every module. Each task may require several modules and the same module can be used for a different task. Using this information, the reliability of a module is estimated as the probability that no failure occurs during the execution of a particular task while in a particular module. Finally, the reliability of the system is computed by estimating the reliability of all modules (in a series configuration) using the expected number of times that a particular component is executed.

The model proposed by (Gokhale et al., 1998) follows the same hierarchical approach described by Kubat. This model computes the reliability of modules by considering the time-dependent failure rates and the utilization of the modules through the cumulative expected time spent in the module execution. A detailed sensibility analysis of the reliability prediction of this model is presented in (Gokhale and Trivedi 2002).

[3] The term “module” is used interchangeably with the term “component”.

(Roshandel and Medvidovic 2004) proposes an approach for estimating the reliability of components by using Hidden Markov Models to deal with the unavailability of information during early stages of the development process.

Some works about various related issues, such as the uncertainty in the reliability estimates in software architecture have been proposed. (Goseva-Popstojanova and Kamavaram 2003) described a study about the sensitivity of some parameters in architectural models by using Monte Carlo simulation and the method of moments. Additionally, Goseva-Popstojanova has performed reliability assessment on large case studies using Open Source Software (Goseva-Popstojanova et al., 2005).

Path-based Models

Similar to state-based models, path-based models use the software architecture explicitly and assume that the modules (or components) fail independently. This kind of model considers a sequence of components executed along each path and computes the path reliability by multiplying their reliabilities.

The Shooman model (Shooman 1976) is one of the earliest models defining the path-based approach. This model assumes that software execution can take different paths. This model assumes that the frequency of paths and the failure probability of every path on each run are known. Then, the model estimates the number of failures expected in N test runs and the probability of failure of the system is estimated as the weighted sum of all the path frequencies and the failure probabilities.

The model proposed by (Krishnamurthy and Mathur 1997) follows an experimental approach. The information includes the monitoring of component traces (i.e., sequences of components) observed during testing or simulation.

(Yacoub et al., 2004) proposes an algorithmic approach to estimate path reliabilities. This model is based on using execution scenarios represented as UML sequence diagrams (Alhir 1998). This kind of diagrams denotes the partial ordering of messages exchanged between entities (i.e., components). Based on this set of diagrams, the model builds a probabilistic model called the component dependency graph. This graph is analyzed using an algorithm that traverses such graphs representing execution paths. This algorithm estimates the reliability of the application as a function of the reliabilities of its component and interfaces.

Similar to (Yacoub et al., 2004), (Singh et al., 2001) and (Cortellessa et al., 2002) propose a model that uses information from UML diagrams. The reliability approach proposed in this model follows a Bayesian strategy. The prior distribution associated with the component probability of failure is a Beta distribution. The overall system estimation is performed using numerical methods.

(Dolbec and Shepard 1995) proposes an extension for the Shooman model by including the component usage ratio. Using this approach the notion of component is introduced into the Shooman model.

(Xiaoguang and Yongjin 2003) proposes a methodological approach for estimating the reliability following a path-based approach where a component transition diagram is derived and random traces are generated. Using this information, the reliability estimation is performed.

(Mason 2002) (Mason 2002) (Hamlet et al., 2001) (Hamlet et al., 2004) propose a component reliability model based on considering the input and output domains in every component. The overall system reliability is performed by profile mappings between components.

Additive Models

This class of models does not explicitly consider the architecture of the software. These models focus on estimating the overall system reliability using the component's failure data.

The Xie and Wohlin model (Xie and Wohlin 1995) assume that component reliabilities can be modeled by a non-homogeneous Poisson process (NHPP). Then, the system reliability is also a NHPP. The system failure intensity is the sum of all component failure intensities.

Everett uses the extended execution time reliability growth model at the software component level (Everett 1992) (Everett 1999). The overall system reliability estimation is performed by superimposing the component reliabilities.

Other Related Works

This classification includes other works that do not fit into the previous classifications but are related to the reliability assessment of software components. (Popic et al., 2005) analyses the error propagation in the Bayesian reliability model proposed by (Singh et al., 2001) with preliminary experimental results.

(Smidts and Sova 1999) and (Smidts et al., 1997) propose a model that uses the evaluation of failure mode probabilities combined with a Bayesian quantification framework. Failure mode probabilities of functions and attributes are propagated to the system level using fault trees. This model accounts explicitly for the type of software development life cycle.

(Shukla et al., 2004b) proposes a conceptual framework to assess the reliability of software components. This framework is based on combining ideas from statistical testing and test oracles.

Several software reliability models have been proposed during the last decades. The assumptions and rationale of each model are mostly based on the information required for reliability assessment. Combinations of reliability model approaches are not frequently found in the literature. The most popular case is the use of architectural-structural reliability models with component reliability estimations derived from software reliability growth models. The research described in this dissertation focuses on combining the architectural-structural approach with the input domain approach to assess the reliability of software assemblies.

2.4 Software Testing Related Aspects

A common source of information for reliability assessment is the testing process. May et al (May 2002) provide a good argumentation supporting the component statistical-based testing of critical systems. Specifically, benefits of using formal methods, statistical models, and the component-approach are discussed.

Statistical testing has been developed around the practical limitation of testing. This area has become popular because of the advances in model-based development. Early works on statistical testing focused on the problem of selecting finite test sets and automating this selection (Thevenod-Fosse and Waeselynck 1991). However, more recent works have proposed the use of specifications to guide the selection of best finite test sets based on coverage criteria (Denise et al., 2004) (Dulz and Zhen 2004) (Le Guen et al., 2004). Other works propose the use of specific statistical models, such as Markov chains, to guide the test generation process (Whittaker and Thomason 1994).

Other issues addressed in statistical testing are the stopping criteria for statistical testing. The decision to stop testing can be based on different criteria, such as the confidence in a reliability estimate, the degree to which testing experience has converged to the expected use of the software, and model coverage criteria based on a degree of state, arc, or path coverage during crafted and random testing (Sayre and Poore 2000).

One testing issue related with reliability models that has been discussed is the sampling approach for testing. Two approaches have been proposed: random testing and partition testing. Advocates of partition testing emphasizes its benefits based on using probabilistic metrics (Chen and Yu 2000) (Chen et al., 2006) (Chen and Yu 1996) (Cai et al., 2005) (Cai et al., 2004). Advocates on random testing emphasize its benefits by using simulation (Sayre and Poore 2000) (Hamlet 1994) (Hamlet and Taylor 1990). A good comparison between the two approaches is provided by (Duran and Ntafos 1984) (Ntafos 2001).

The research described in this dissertation uses information from the testing process. However, the issues of sampling approaches and the stopping criteria are out of the scope of this work.

2.5 Operational Information Modeling

Operational information plays an important role in Software Reliability Engineering (Musa 2004a) and in some reliability models. Reliability assessments are weighted using such information. The use of this kind of information has been recognized as an important asset in some software engineering practices (Musa 1993). However, the complexity in gathering such kind of information is recognized as an important issue by (Voas 2000).

An operational profile is related with the actual operation of the system. This relation is usually associated with other software development artifacts such as the software requirements specification. There are some works connecting the system specification and the operational information. (Runeson and Regnell 1998) proposes an integrated model representing use cases and operational information.

Other works focus on proposing a structure for operational information. (Gittens et al., 2004) proposes an extended operational profile addressing not only system operations but also process, structure, and data. Process corresponds to the usual operations defined in the basic operational profile; structure corresponds to the set of configurations (i.e., environment, hardware, software) defined for the operational profile; and data corresponds to actual values used in the operations defined in the operational profile. (Woit 1993) proposes an extended operational profile that introduced the notion of event. This kind of extension is suitable for systems that can be modeled using statecharts or similar models.

Experiences on implementing and manipulating operational information are described by (Bousquet et al., 1998). In this work, the authors described how to incorporate operational information into a formal testing tool. (Bishop 2002) described a study about rescaling reliability bounds by studying the change in operational information and discussing the results from a sensibility analysis.

(Shukla et al., 2004a) proposes a method for addressing the issue about how to derive operational profiles for software components. This method uses both actual usage data and intended usage assumptions to derive a structure (named as usage structure), usage distribution, and other parameters. Using this structure the operational distribution is produced.

In spite of the operational information importance, standard notations to describe such information are not proposed. The research described in this dissertation uses operational information to weight the reliability estimation and produce more meaningful estimations. Additionally, this research focuses on lack of a standard notation to specify operational information.

2.6 Summary

This chapter presented a summary of the state of the art in areas related to the research question addressed in this dissertation. This chapter emphasized related work in these areas:

- Works on attribute assessment based on software architectures or component-based approaches.
- Works on software reliability assessment. This review included also those works using the structure of the system.
- Works on testing issues that are relevant for reliability assessment. These works included issues in random/partition testing and statistical software testing.
- Works on operational information modeling.

The next chapter presents a description about the reliability assessment framework proposed in this dissertation.

Chapter 3

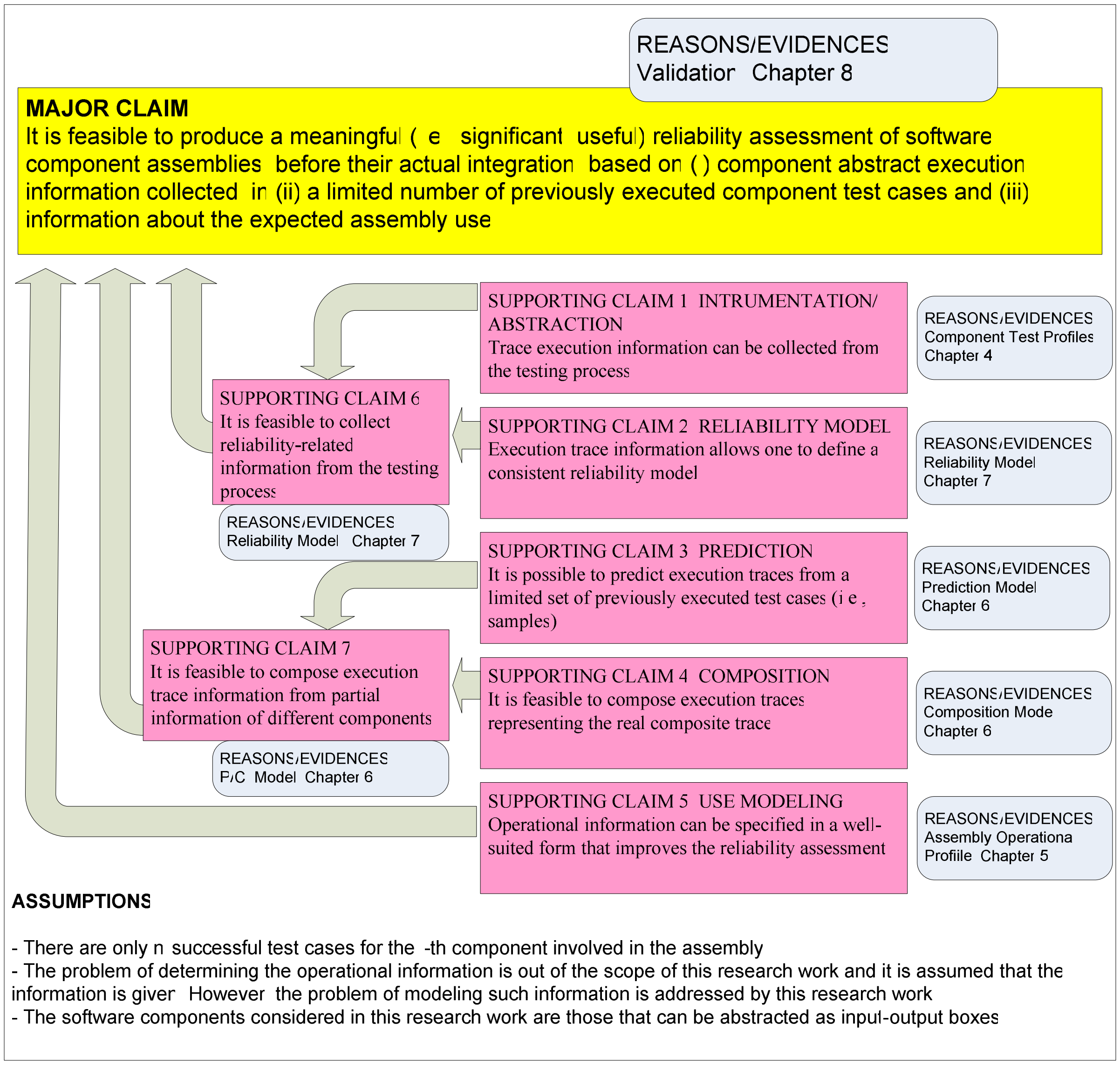
The Test Profile Analysis Framework

This chapter describes the framework proposed to assess the reliability of software assemblies. The items described in this chapter include the rationale supporting the framework; the framework architecture, including phases, tasks, and artifacts; the models required as foundations for artifacts and tasks; and a basic illustrative example used in the subsequent chapters.

3.1 Framework Rationale

This section describes the rationale supporting the framework proposed in this dissertation. The rationale is organized according the strategy suggested by (Hairston et al., 1999). First, a major claim is presented. Then, the major claim is decomposed into five basic claims. Finally, each basic claim is sustained with evidences that are described in the following chapters.
Figure 6 shows a diagram that depicts the organization of claims that support the rationale behind the proposed framework where the box at the top denotes the thesis statement (i.e., major claim). The major assumptions defined for this work are listed at the bottom of the figure. The major claim is supported using evidence provided by the supporting claims that are located below the major claim. This set of supporting claims is arranged as a tree where the arrows denote how a set of claims support a more complex claim. For example, supporting claim number 6 is supported by the claim numbers 1 and 2. Additionally, the diagram includes the chapter containing the evidences supporting the corresponding claim..

The major claim is supported based on the evidences of six claims. Every claim is described as a set of problems: first, the problem of characterizing the reliability-related information of software components using only limited testing information (i.e. claim number 7); second, the problem of defining a composition model and a execution trace prediction model based on such characterization (i.e., claim number 6); third, the problem of modeling the expected assembly use (i.e., claim number 5); and fourth, the problem of defining a reliability model based on the information provided by the composite model as well as the operational information defined for the assembly (i.e., major claim).


REASONS/EVIDENCES Validatior Chapter 8
MAJOR CLAIM
It is feasible to produce a meaningful (e significant useful) reliability assessment of software component assemblies before their actual integration based on ( ) component abstract execution information collected in (ii) a limited number of previously executed component test cases and (iii) information about the expected assembly use
SUPPORTING CLAIM 1 INTRUMENTATION/ ABSTRACTION
Trace execution information can be collected from the testing process
REASONS/EVIDENCES Component Test Profiles Chapter 4
SUPPORTING CLAIM 6
It is feasible to collect reliability-related information from the testing process
REASONS/EVIDENCES Reliability Model Chapter 7
SUPPORTING CLAIM 2 RELIABILITY MODEL
Execution trace information allows one to define a consistent reliability model
REASONS/EVIDENCES Reliability Model Chapter 7
SUPPORTING CLAIM 3 PREDICTION
It is possible to predict execution traces from a limited set of previously executed test cases (i e , samples)
REASONS/EVIDENCES Prediction Model Chapter 6
SUPPORTING CLAIM 7
It is feasible to compose execution trace information from partial information of different components
REASONS/EVIDENCES P/C Model Chapter 6
SUPPORTING CLAIM 4 COMPOSITION
It is feasible to compose execution traces representing the real composite trace
REASONS/EVIDENCES Composition Mode Chapter 6
SUPPORTING CLAIM 5 USE MODELING
Operational information can be specified in a well-suited form that improves the reliability assessment
REASONS/EVIDENCES Assembly Operationa Profiile Chapter 5
ASSUMPTIONS

- There are only n successful test cases for the -th component involved in the assembly
- The problem of determining the operational information is out of the scope of this research work and it is assumed that the information is giver However the problem of modeling such information is addressed by this research work
- The software components considered in this research work are those that can be abstracted as input-output boxes

Figure 6.Thesis Claim Structure.

The problem of characterizing the reliability-related information of software components is addressed by defining a new artifact, namely Component Test Profile or just Test Profile (i.e., supporting claim number 6). This artifact organizes the information produced during the component testing process. It includes information such as inputs, expected outputs, and execution-related information for every performed test case.

The execution-related information is produced by instrumenting the component's source code and by monitoring the component during the test case execution (i.e., supporting claim number 1). The instrumentation is a process that inserts specific instructions in the source code for monitoring purposes. The inserted instructions send run-time information to a special purpose monitoring system that runs concurrently and independently of the tested component.

The monitored information is characterized by blocks of source code, referred to as execution nodes. Execution nodes are defined by considering the distinct execution paths that the execution of the component may follow. The output of this analysis can be visualized as a graph containing nodes and arrows denoting the distinct nodes that can be visited during the component execution. The resulting monitored information is a sequence of visited execution nodes. This sequence is called the execution trace.

The incorporation of this execution-related information into every test case improves the component characterization for reliability purposes (i.e., supporting claim number 2). This internal information provides a detailed view about what is "used" during the component execution without getting into detailed instruction level information. This approach can be considered as a gray-box approach since it not only considers external information but also considers the internal run-time behavior of the component.

The next issue focuses on the composition problem derived from the sequential assembly of components (i.e., supporting claim 4). The composition focuses on producing a model representing a composite behavior of such assembly using the same characterization defined for component test profiles. In other words, the composition output is another component test profile representing the composite behavior (i.e., assembly test profile). The composition combines execution traces since they provides the evidences used to assess the reliability of the component at the lowest level. In addition, the composition also considers the corresponding inputs that generate such execution traces since the specific sequence of execution nodes is determined by the corresponding inputs to the component.

For example, the composition of two components connected sequentially is described as follows. A test case output from the first component in the assembly is selected. Then, the model looks for the most similar test case input in the second component. "Most similar" means that such output and input can be considered the same value.

The determination of which test case or test point to use in the second component is trivial if a comprehensive set of test cases is provided**.** However, since the test profile only contains a limited set of test cases, a new sub-problem is addressed: the problem of predicting execution traces based on the information recorded in the component test profile (i.e., supporting claim 3). The problem of predicting execution traces is solved by choosing the best prediction model among distinct techniques, such as artificial neural networks, classification trees, or the best binary matching technique.

Then, the composition model for sequential assemblies of software components concatenates execution traces defined by the corresponding inputs from the first component and the predicted execution traces from the second component (i.e., supporting claim 7).

The composition for other architectural styles, different from sequential or data-flow styles, follows the same approach but it requires additional instrumentation and infrastructure that is discussed in Chapter 6.

The problem of specifying the expected assembly usage (i.e., supporting claim 5) is addressed by defining a notation using the Unified Modeling Language (UML). The notation is defined as a UML profile which includes a set of modeling constructs, definitions, and examples. This notation includes concepts such as operational modes, operations, functionalities, and data dependencies.

The reliability model combines the resulting composite model and operational information addressing the major claim. The proposed reliability model assumes that there is no knowledge about the failure distribution and every time that an execution node is visited, it can be considered as a different possible scenario. Thus, an execution node probability of failure is estimated using an estimator proposed by (Miller et al., 1992). A specific test case is defined by a sequence of execution nodes and the failure of one node impacts the entire result of the test case. The test case probability of failure is estimated by considering a sequential array of execution nodes. Then, the component probability of failure is calculated as the expected test case probability of failure.

The previous description produces a point estimation that is complemented with the confidence interval estimation. This interval estimation is done with the Bootstrap technique (Shelemyahu and Kenett 1998). This technique does not assume any distribution and it can produce interval estimation. The core part of this technique is based on resampling and finding the distribution of the point estimation.

A prediction error is provided as an additional measure for the quality of the assessment. A large error in prediction means that more test cases are required to improve the component characterization.

The reliability model can incorporate operational information to weight the reliability assessments. The operational information might divide the input domain of the assembly forming a partition. The divisions are named as sub-domains. Then, a reliability assessment for each sub-domain is performed using the information from the component test profiles. Subsequently, the probabilities associated with such sub-domains allow weighting the reliability assessment obtained for every sub-domain. As a result, more meaningful reliability assessments are produced since the more important sub-domains contribute more to the reliability assessment.

3.2 Framework Overview

The Test Profile Analysis Framework organizes systematically a set of activities and information to support early reliability assessment. The core elements of the framework are phases, tasks, artifacts, and models. Figure 7 shows a graphical representation of these elements using UML notation.

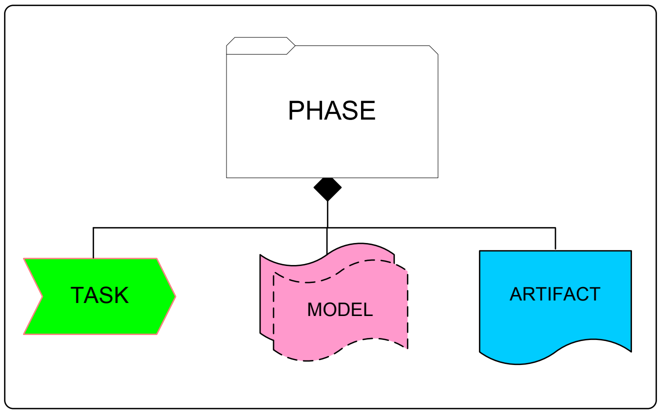


Figure 7. Framework Elements.

A *phase* is a group of tasks that generates a set of specific artifacts (i.e., deliverables). A phase is composed of tasks, artifacts, and abstract artifacts.

A *task* is a well-defined unit of work. Tasks have readiness criteria (preconditions) and completion criteria (postconditions).

An *artifact* is one of many kinds of tangible products produced during the development of software. In our context, an artifact is a specific piece of information with a concrete data representation (i.e., it has a concrete format and representation).

A *model* is an abstraction that uses mathematical language to describe the behavior, structure, or properties of specific entities or processes.

The relationship between these elements is as follows: a phase is composed of one or more tasks. Every task has a set of one or more required artifacts (i.e., inputs) and a set of produced artifacts (i.e., outputs). An artifact is described using abstracts artifacts to facilitate the manipulation and understanding of such artifact. A model provides a formal foundation within a specific task and it impacts the semantics of the artifacts.

3.3 Framework Phases

Figure 8 shows the four phases of the Test Profile Analysis Framework. The phases relate sequentially to each other meaning that the deliverables from one phase are required as input for the next phase.

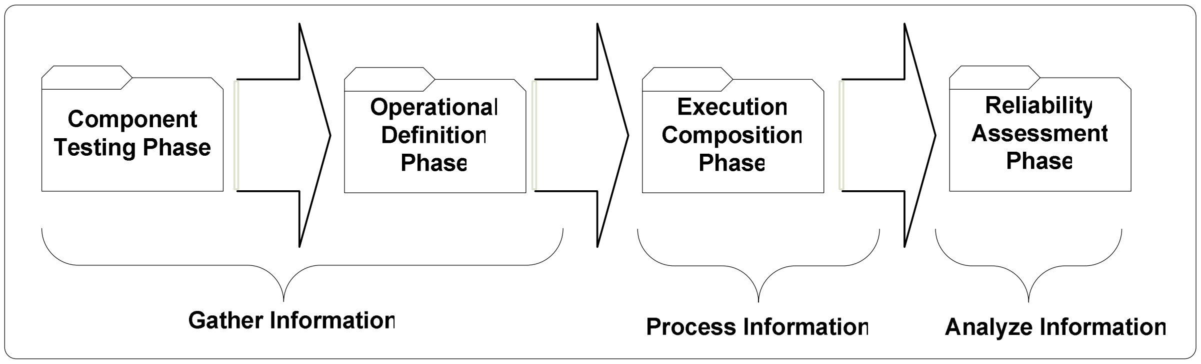

Component Testing Phase
Operational Definition Phase
Execution Composition Phase
Reliability Assessment Phase
Gather Information
Process Information
Analyze Information

Figure 8. Framework Phases.

As Figure 8 shows, the phases can be organized as a three step approach: First, all required information for the reliability assessment of the software assembly is collected. Second, the information gathered requires some processing and preparation for the reliability assessment. Finally, in the third step, the analysis of the processed information is performed.

A more descriptive representation of the framework is shown in Figure 9. Framework phases are represented by columns. The tasks are enclosed within their respective phase. The arrows are used to specify when an artifact is required as input or is produced as output. The dotted arrow denotes input artifacts and the solid arrow denotes output artifacts. Square-ended lines are used to denote that a model is required for a task. For example, the task *Create Assembly Test Profile* requires as input artifacts the *Component Test Profiles* and the *Assembly Operational Profile*. This task generates the artifact named *Assembly Test Profile*. The task requires the *Abstract Execution Composition Model* as well as the *Execution Trace Prediction Model*.

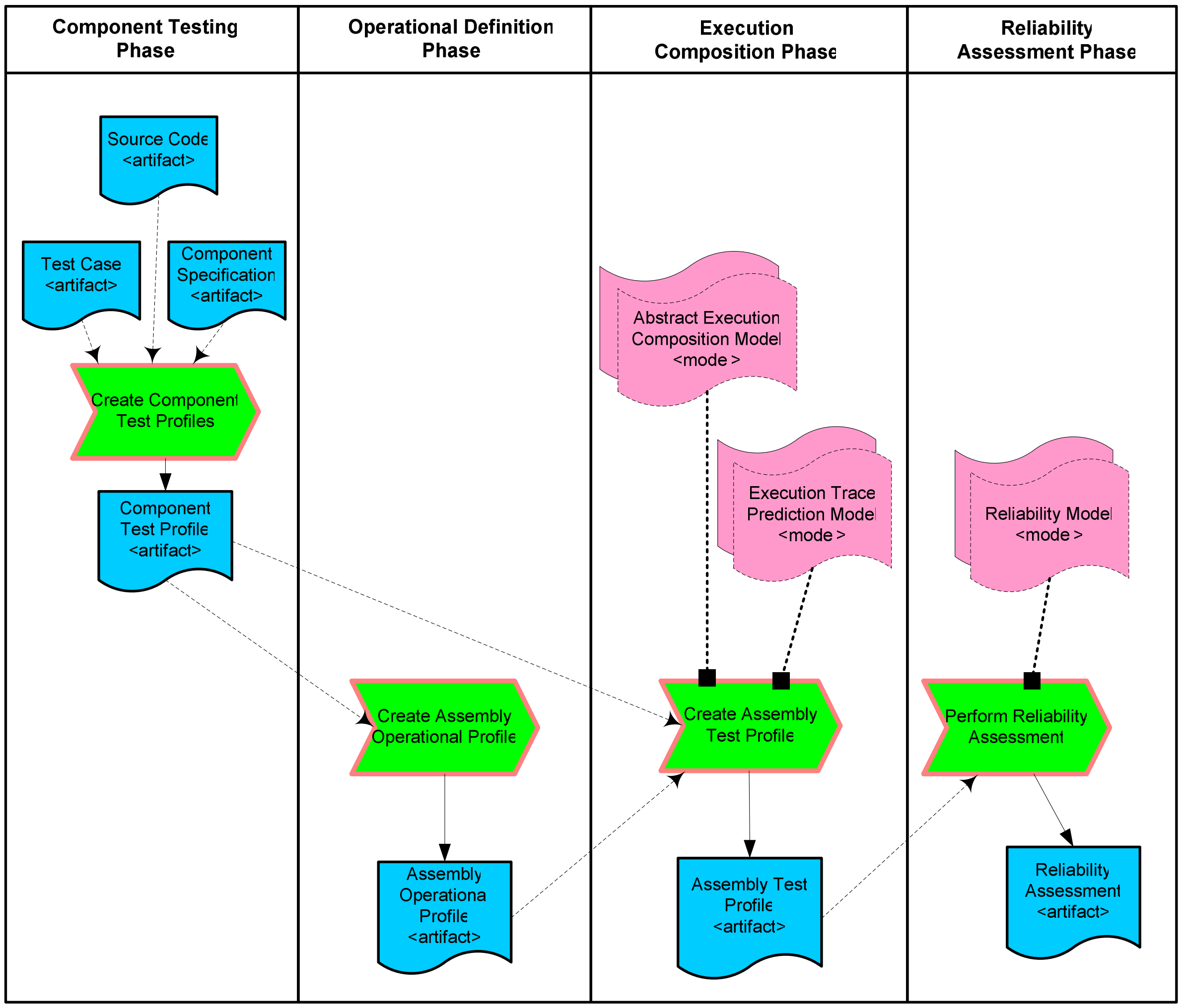
The first phase, the Component Testing Phase, can be seen as an independent phase because the component supplier might produce the required artifacts. The other phases are focused on the actual assembly evaluator.

3.3.1 Component Testing Phase

The Component Testing Phase is the first framework step. During this phase the component-related information required for reliability assessment is collected, organized, and prepared for the next phases.

The goal of the Component Testing Phase is the generation of relevant component information derived from the component testing process. This information includes a set of test cases, execution traces, and test results.

The task "*Create Component Test Profile"* is performed in this phase. The goal of this task is the generation of component test profiles. This task uses three input artifacts: Component Source Code, Component Specification, and Test Cases. The *Component Specification* is a description of the functionalities, domains, and other relevant information describing the component behavior. The *Test Case* artifact provides the usual information required to test the component (i.e., inputs and the expected outputs). The *Component Source Code* is an artifact that contains a human-readable representation of the component behavior. This representation uses notations known as programming languages.



Component Testing Phase
Operational Definition Phase
Execution Composition Phase
Reliability Assessment Phase
Source Code <artifact>
Test Case <artifact>
Component Specification <artifact>
Create Component Test Profiles
Component Test Profile <artifact>
Abstract Execution Composition Model <mode >
Execution Trace Prediction Model <mode >
Reliability Model <mode >
Create Assembly Operational Profile
Create Assembly Test Profile
Perform Reliability Assessment
Assembly Operational Profile <artifact>
Assembly Test Profile <artifact>
Reliability Assessment <artifact>

Figure 9. Framework Expanded View.

The three artifacts of the "Component Testing Phase" answer three key questions:

What are the inputs and outputs of the software component? – What should be tested?	→	Test Specification
What has been tested?	→	Test Cases
What is the execution behavior during the test case execution?	→	Execution Information

The component specification provides a point of reference which is used to generate test cases. Component source code provides the means to analyze the component behavior for instrumentation purposes. Test cases provide the means to validate the absence of failures.

Create Component Test Profile task generates *Component Test Profiles*. These artifacts consolidate the information collected after executing all test cases. This information includes the set of test cases and test results, the component specification, and the corresponding execution traces.

Chapter 4 describes in detail this phase including related tasks and artifacts.

3.3.2 Operational Definition Phase

The Operational Definition Phase is the second framework phase. During this phase, the information about the expected use of the component is defined in terms of the specified assembly requirements.

The goal of the Operational Definition Phase is the generation of a quantitative characterization about the expected usage of the assembly. The operational information generally relates assembly requirements or functionalities with a quantification about the expected use of such functionalities. This quantification is usually defined in terms of probabilities.

The task "*Create Assembly Operational Profile*" is performed during this phase. The goal of this task is to generate a specification (i.e., a model) about the operational information. The activities involved during this task include the determination of the major operations of the assembly in terms of the assembly requirements specification, the estimation of the expected use of such operations, and the definition of a model that integrates both pieces of information.

In addition, this task specifies which components are required for the assembly and how these components are connected. During this task the component are selected according the functionality of the assembly.

The task uses as input artifacts a set of *Component Test Profiles.* These artifacts describe the characteristics of the components that will be used to create the assembly.

This task generates as output the *Assembly Operational Profile* artifact. This artifact is a combination of information about what operations the assembly will provide, measurements about their expected use, and the components involved in the assembly.

Chapter 5 describes in detail this phase including related tasks and artifacts.

3.3.3 Execution Composition Phase

The Execution Composition Phase is the third framework phase. During this phase, the information from the component test profiles is combined to generate a composite model, referred to as the *Assembly Test Profile*.

The goal of the Execution Composition Phase is the manipulation and generation of a new artifact resulting from the test profile composition process. This information includes the execution-related information about the abstract execution defined for every component included in the assembly.

The task "*Create Assembly Test Profile*" is performed during this phase. The goal of this task is to create a new test profile by combining individual component test profiles. This task requires models supporting both the composition process and the execution prediction process.

The composition process uses a matching mechanism that composes two test cases from different components. The information from these two matched test cases is combined to generate a composite test case. Since component test profiles have a limited number of test cases, the framework uses prediction models associated with each component test profile. Using these prediction models the composition mechanism can predict the best matching test case.

The task requires as input artifacts a set of *Component Test Profiles* and the *Assembly Operational Profile*. The component test profiles provide the atomic elements to be composed while the Assembly Operational Profile describes what and how to compose such atomic elements.

This task generates as output the *Assembly Test Profile*. This artifact includes the same information defined for any component test profile, but it includes information about the prediction uncertainty (i.e., prediction error). This error is derived from the execution trace prediction performed during the composition.

This phase requires two models. The *Abstract Execution Composition Model* prescribes the rules and principles governing the composition process. The *Execution Trace Prediction Model* provides a statistical model supporting the execution trace prediction when there is limited information in the component test profile.

3.3.4 Reliability Assessment Phase

The Reliability Assessment Phase is the fourth framework phase. During this phase, the reliability assessment of the assembly test profile is calculated. The goal of the Reliability Prediction Phase is the estimation of the assembly probability of failure. This estimation uses the artifacts produced during previous phases.

The task "*Perform Reliability Assessment*" is performed during this phase**.** The goal of this task is to estimate the assembly probability of failure. The task requires as input artifacts the *Assembly Testing Profile* and the *Assembly Operational Profile*. This task produces the *Reliability Assessment* artifact**.** This artifact includes estimations such as the assembly probability of failure and the confidence interval of such estimation.

This phase requires a *Reliability Model* supporting the reliability assessment. This model is based on all evidences provided in the assembly test profile (i.e., test cases, execution traces, and domains) and the information contained in the assembly operational profile (i.e., operations and expected usage probabilities).

3.4 Models Supporting the Framework

The framework requires specific models for some of its tasks. Having a model supporting a task is an advantage since the task performance can be improved as better models are proposed or refined. Furthermore, the framework can be tailored to assess other system properties by collecting different information in the component test profiles and defining new models that analyze such information.

The models provided in the framework answer three important questions:

How to compose/ combine information from the component test profiles?	→	Abstract Execution Composition Model
How to solve the issue of the limited amount of information?	→	Execution Trace Prediction Model
How to assess the information of a specific attribute?	→	Reliability Model

The first question refers to the need for well-defined rules to manipulate and combine information from two or more component test profiles. The composition model proposed in Chapter 6 specifies the rules to find matching test cases and combine information such as execution traces, inputs, and outputs.

The second question refers to the need of execution prediction models to enable the composition process. The prediction model produces the matching value required to guarantee the composition. The model can be generated using techniques such as artificial neural networks, classification trees, or the best binary matching technique.

The third question is related to the reliability assessment. The reliability model proposed in Chapter 7 bases the estimation on information from the component test profiles, the assembly test profile, and the assumptions defined for the framework. The model assesses the reliability of a single component. This component might be a single component or a composite component obtained from the component test profile composition.

The three models discussed above are defined formally. The Execution Composition Model is defined as a simple composition algebra covering basic primitives and a composition operator. The Trace Execution Prediction Model is selected from three prediction models having a well-known formalization. The Reliability Model is an architectural – input domain model. This model weights its estimations with information from the Assembly Operational Profile.

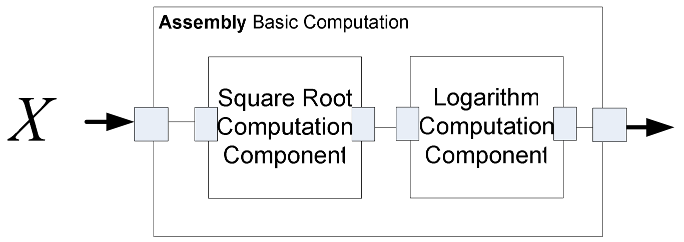
3.5 Basic Case Study

This section presents a basic case study that is used to facilitate the understanding and to illustrate the concepts, models and activities of the framework, which are discussed in detail in the following chapters.

The basic example is an assembly for performing the following computation:

$$Y = \log\left(\sqrt{X}\right)$$

The assembly is composed of two components: the square root computation component and the logarithm computation component. The assembly of these components follows a data-flow architectural style, as shown in Figure 10.


Assembly Basic Computation
X
Square Root Computation Component
Logarithm Computation Component
Y
Figure 10. Basic Case Study Assembly.

According to the component specifications, the first component in the assembly requires as input an integer value ranging from 0 to 65535 and the output is a real value ranging between 0 and 255.9980. Similarly, the second component receives as input a real value ranging between 0 and 3.402823 x 10^{38}.

This basic case study will be used to illustrate the concepts and procedures described in the following chapters.

3.6 Summary

This chapter has described the framework proposed for the reliability assessment of software assemblies. The chapter includes a discussion of the rationale supporting the framework as well as the claims supporting every framework phase.

The next chapter will address the claim about the feasibility of collecting execution traces from the component testing process.

Chapter 4
Modeling and Organizing Testing Information

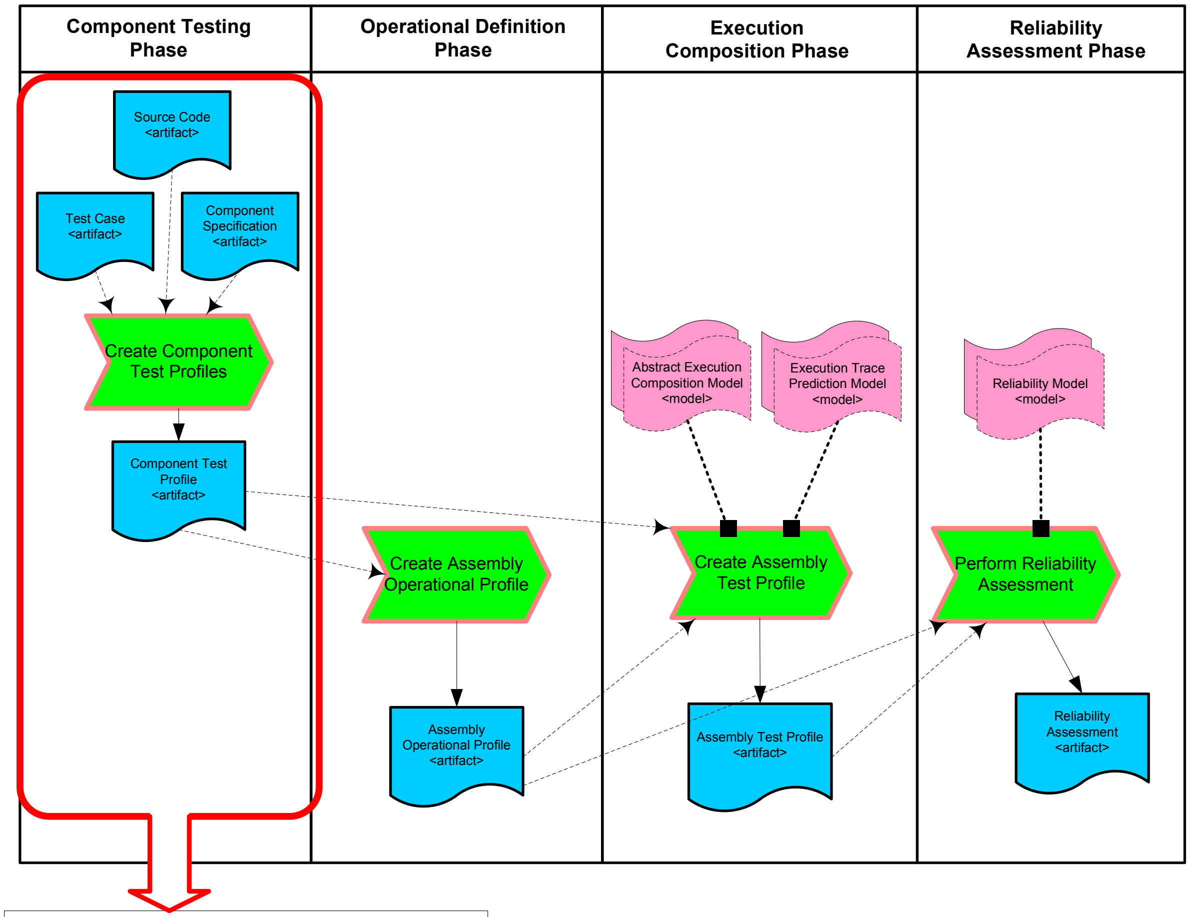
This chapter addresses the supporting claim: "*Execution trace information can be collected from the testing process.*" To support this claim, two important issues have to be addressed: the categorization of information that is used and produced during the testing process and the artifact to organize such kind of information.

The Component Testing Phase is the proposed solution to the issues mentioned above. Figure 11 shows the location of this phase in the framework. Component Test Profiles are the output artifacts produced in this phase. These artifacts are the practical solution supporting the claim addressed in this chapter.

This chapter is organized as follows: First, a detailed description of the problem is presented. Second, the rationale supporting the solution is explained. Third, a description of the task and artifacts used through the component testing phase is provided. Fourth, a set of examples to illustrate the proposed solution is presented. Fifth, an overview of the tools required to implement the solution is described. Finally, a discussion of the results, limitations, and future extensions is included.

4.1 Problem Description

Software testing is a technique that provides information to assess the reliability of a product. Reliability is an attribute that requires information about the existence or absence of failures. A failure is the inability of a system or component to perform its required functions within specified performance requirements (IEEE 1998). The evidence of failures is generated during the execution of test cases.


Component Testing Phase
Operational Definition Phase
Execution Composition Phase
Reliability Assessment Phase
Source Code <artifact>
Test Case <artifact>
Component Specification <artifact>
Create Component Test Profiles
Component Test Profile <artifact>
Create Assembly Operational Profile
Assembly Operational Profile <artifact>
Abstract Execution Composition Model <model>
Execution Trace Prediction Model <model>
Create Assembly Test Profile
Assembly Test Profile <artifact>
Reliability Model <model>
Perform Reliability Assessment
Reliability Assessment <artifact>

SUPPORTING CLAIM 1: INTRUMENTATION/ ABSTRACTION
Trace execution information can be collected from the testing process.

Figure 11. Framework Expanded View with Focus on the First Phase.

The complexity of software testing comes from the testing space size (i.e., the size of the input domains). The number of different inputs that are possible to test is usually enormous. For example, consider a simple function that has two positive integer input parameters (i.e., parameters ranging from 0 to 65535). The number of different possible input values is defined as the set of all possible combinations of two integer values (i.e., the number of all possible combined values is 65535^2).

In practice, testing process has additional important issues to consider beyond the complexity previously mentioned**.** From the product perspective, two issues require special attention during the testing process: the generation of testing data and the determination of testing outcomes (Jorgensen 1995) (Hoffman 1998) (Utting et al., 2006) (i.e., the problem of determining when a test case is correct, incorrect, or undetermined). From a process perspective, the issues that require special attention during the testing process are those associated with the use, organization, and management of information (e.g., configuration management, storage management, and test execution).

The issues mentioned above define a complex scenario where any additional effort to gather more information may impact the time and resources of the testing process. Any decision to modify the testing process will add overhead to the project. For example, consider a testing process having 700 test cases to execute. If an additional inspection is included for every test case and the time required for an inspection is *t*, then, the project will require 700*t* additional time units. The tradeoff here is the cost of additional time versus the saving derived from the benefit of such inspections.

The problem addressed in this chapter is the definition of a strategy to collect execution trace information during the testing process.

The generation of execution trace information requires addressing issues such as selecting the information to be recorded in the execution trace, defining the runtime infrastructure that is required to monitor such information, and implementing the tools that are required to instrument the source code before testing the software.

Storing and manipulating execution traces is another issue to consider since the size and number of such traces might consume a considerable amount of resources. This issue is related to the use of artifacts and the tools to manage such artifacts.

The approach to solve the issues mentioned above is described in the next section.

4.2 Solution Approach

This section describes the solution proposed to address the problem of generating execution traces during the testing process. The solution includes the use of a new artifact and a strategy to collect execution information.

4.2.1 The Test Profile Artifact

A new artifact is proposed to solve the problem of organizing the testing information. This artifact identifies an information structure for storing, organizing, and managing the testing information as well as the required execution trace information. This new artifact is named Test Profile. A test profile is an artifact that stores the actual information and meta-information about the testing process. A representation of the conceptual elements stored in a test profile is shown in Figure 12.

Figure 12 shows a metamodel where metaclasses represent data elements stored within the test profile. This class diagram is created using the UML notation (Jacobson et al., 1998). A class diagram contains boxes representing entities. The lines connecting these boxes represent relationships between the entities. The term metaclass refers to a special kind of entity that is capable of being instantiated into different occurrences. For example, a State metaclass may be instantiated into class instances such as Texas, Arizona, and so on.

The diagram specifies that a test profile is composed of test data (i.e., a set of test cases), test meta-data (i.e., descriptions about the information stored within the test profile), and test results (i.e., outcome of each test case and the corresponding execution information).

Execution information describes the run-time behavior of the system or component during the test case execution. Additional run-time information can be included in the execution information by extending the current meta-model, as shown in Figure 12 (Jacobson et al., 1998).

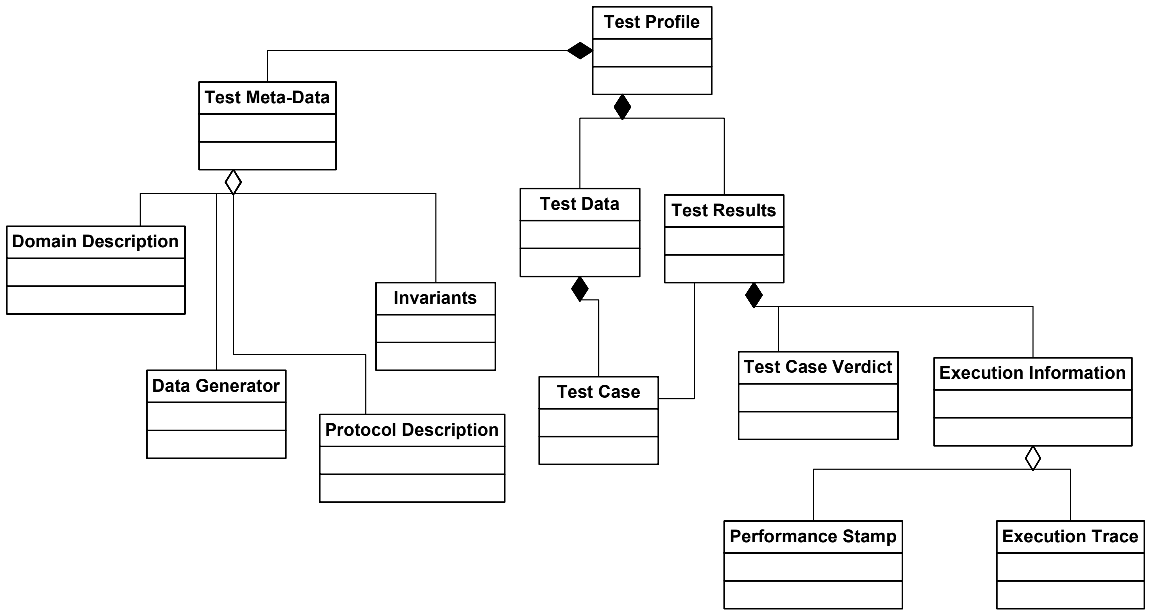


Figure 12. Test Profile Meta-Model.

One additional characteristic of the test profile is that it might contain meta-information about test cases. This additional information includes meta-information describing invariants over the test cases, protocol descriptions associated with the set of test cases, descriptions about the input and output domains, and data generators defining a compact representation of the data used in the test cases.

This metamodel is designed to support different testing scenarios and provide information for different purposes. There is no constraint on what information to use or include in test profiles, the information selection depends on the specific analytical needs. Since our interest focuses on reliability assessment, the minimum information that it is required includes the test cases, the results, and the execution trace information. Examples of test profiles can be found in Section 4.4.

4.2.2 Execution Trace Infrastructure

The execution trace monitoring uses an abstract model of the source code. This model is represented with a graph, referred to as an execution graph. A node in the execution graph represents a block of one or more source code instructions. Edges connecting nodes represent different execution paths derived from the conditionals and loops within the source code. This model is built using control-flow analysis (Nielson et al., 2006) (Allen 1970). This high-level abstraction reduces the infrastructure complexity for collecting and analyzing execution traces.

The procedure to produce and store execution traces has two steps: The source code instrumentation of the system under test and the execution monitoring performed during the testing process.

The source code instrumentation is a process performed to add special purpose instructions within the actual system source code. This addition can be considered as source code refactoring (Fowler et al., 1999). These instructions are designed to have a minimum impact on the system execution performance. The added instructions produce the list of nodes visited during the test case execution. Figure 13 illustrates the instrumentation process that ends with the generation of a binary version of the system under test. The source code instrumentation may be performed manually or automatically. A discussion about the implementation of the instrumentation is presented in Section 4.5.1.

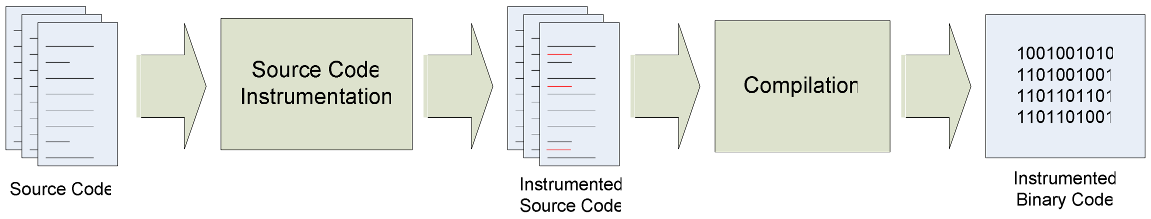


Figure 13. Source Code Instrumentation.

The source code instrumentation is performed by the developer before starting the testing process. Depending on the specific methodology or development process, the source code instrumentation may be performed by a different person within the development team.

The execution monitoring infrastructure required to collect the traces could be included within the component under test or be implemented as an external system. The disadvantages of including this runtime infrastructure within the component are the execution overhead and the difficulty to differentiate the failures generated between the infrastructure and the component. The strategy followed in this work implements the runtime infrastructure as an external system. This strategy isolates, to some degree, the impact of failures of such infrastructure and the overhead generated within the system under test.

4.3 Solution Description

This section describes how the ideas discussed in the previous section are implemented into the proposed framework. First, a description of the artifacts required and produced during this phase is provided. Next, a description of the task required for the component test profile construction is explained.

4.3.1 Artifacts Description

The input artifacts required in this phase are the component source code, a set of test cases, and the component specification. The output artifact is a set of one or more component test profiles.

The *Component Source Code* artifact is a version of the component source code at test time. The *Test Case* artifact is a description of the test case. It includes a set of test inputs, execution conditions, and expected results. The test case is designed for a particular objective, such as to test a particular program path or to verify compliance with a specific requirement. In other words, a test case is a specific scenario where a specific functionality, behavior, or property is validated against a known result.

The set of test cases provides part of the evidence that is used to assess the reliability of the component. This evidence provides the certainty that the component is failure-free in the inputs specified by the test cases.

The *Component Specification* artifact describes the component functionality. The level of formality of this description may vary. Functionality signatures and a description of the goal of such source of functionality are the minimum elements required. The component specification provides a reference for evaluating the subdomain partitioning and the randomness of test cases.

The *Component Test Profile* artifact is the primary source of information to determine the component reliability. This artifact is the core element in the framework since it collects most of the required information used for the reliability assessment. The component test profile includes information on what was tested, what was the test result, and what was the executable behavior of the component for each test case. In other words, the component test profile has the inputs, the outputs, and the corresponding execution trace for each test case.

A template for every artifact is provided in Appendix B.

4.3.2 Task: Create Component Test Profiles

The *Create Component Test Profile* task is performed during the first phase. The purpose of this task is the generation of component test profiles.

The activities involved in this task are the instrumentation of the component source code, the testing of the given component using all the test cases, and the organization of all the information produced during the testing process.

The instrumentation process uses static code analysis. Static code analysis is the analysis of a software source code that is performed without actually executing the program. In most cases, the analysis is performed on the source code; in other cases, it is performed on the object code. There are diverse analyses focusing on different aspects of the source code (e.g., control-flow analysis (Allen 1970), data-flow analysis (Forgacs 1996), type analysis (Palsberg 2001)).

The sophistication of static analysis varies depending on the goal of such analyses. For example, analysis can be used to highlight possible coding errors or to produce an abstract model used to prove the absence of defects such as memory leaks or deadlocks.

The instrumentation technique selected in this work uses principles from control-flow analysis. Our instrumentation procedure finds blocks defined by the different execution paths within a source code. The determination of a block is based on recognizing control-flow structures such as conditionals, loops, and alternatives. Section 4.5.1 provides details about the instrumentation process.

The instrumentation requires an infrastructure for monitoring the testing process. This monitoring process is implemented by an external system which functions as an observer. The monitoring infrastructure records blocks that are visited during a particular test case execution. Section 4.5.2 gives details on the monitoring system.

4.4 Examples

This section describes two examples to show how the task defined in this phase is performed. Two components are analyzed and their corresponding test profiles are produced.

4.4.1 Square Root Component Test Profile

This example describes a simple component that computes the square root of an integer value. The artifacts provided for the construction of the component test profile are:

- Component Specification
- Test Cases
- Component Source Code

The component specification is shown in Table 6. The table lists the signature of the component functionality and the corresponding domain description. The domain description is defined in two parts: the theoretical specification defined for the domain and the actual implemented ranges.

Component Specification		
Component Name	Square Root Component	
Component Functionality	**Functionality Signature**	**Domain Description**
Square root computation function	Sqroot(*input*): *output* *input*: Parameter whose square root will be computed *output*: Return value that represents the computed square root value	**Theoretical** $input \in \aleph$ $output \in \Re$ **Implemented** $input \in [0...65535]$ $output \in [0,255.998047]$

Table 6. Square Root Component Specification.

This specification may contain more detailed and formal information about the component. The level of formality and the amount of information specified depend on the purpose of the component.

A partial set of test cases, for this component, is listed in Table 7. This table lists a set of random values defined for testing the square root component. The *Input Value* column contains the values to the square root component. The *Expected Output Value* column contains the corresponding values used to verify the correctness of the computation performed by the component.

Test Cases		
Test Case Number	**Input Value**	**Expected Output Value**
1	1	1
2	43	6.557438
3	49	7
4	55	7.416198
5	56	7.483315
6	58	7.615773
7	84	9.165152
8	95	9.746795
9	106	10.29563
10	117	10.81665
12	130	11.40175
13	134	11.57584
.	.	.
.	.	.
.	.	.

Table 7. Partial List of Test Cases for the Square Root Component.

The component source code is shown in Table 8. The programming language used to implement the computation is Visual Basic.

Component Source Code

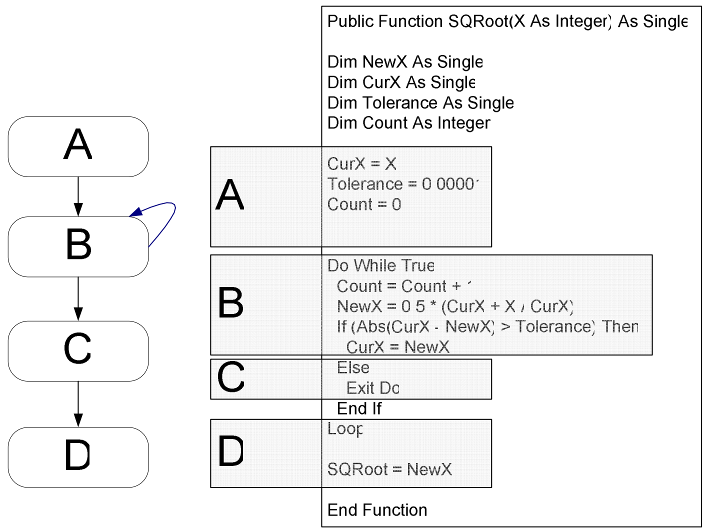
Public Function SQRoot(X As Integer) As Single
 	
Dim NewX As Single
Dim CurX As Single
Dim Tolerance As Single
Dim Count As Integer
 	
CurX = X
Tolerance = 0.00001
Count = 0
 	
Do While True
  Count = Count + 1
  NewX = 0.5 * (CurX + X / CurX)
  If (Abs(CurX - NewX) > Tolerance) Then
    CurX = NewX
  Else
    Exit Do
  End If
Loop
 	
SQRoot = NewX
 	
End Function

Table 8. Square Root Component Source Code.

The previous artifacts are used to generate the component test profile. As explained earlier, the Create Component Test Profiles requires the instrumentation of the source code.

The instrumentation performs an analysis over the source code structure. The analysis produces the execution graph. This graph denotes the different execution paths that a program may follow. Figure 14 shows the execution nodes assigned for the source code shown in Table 8. The figure also shows the graph representing the program control flow.

Control-flow analysis uses well-known automatic techniques applied in compiler technology (Srikant and Shankar 2002). These techniques include advanced algorithms performing sophisticated tasks such as recognizing logical structures in a source code file, optimizing the source code, or verifying type-safe assignments.



```
Public Function SQRoot(X As Integer) As Single

Dim NewX As Single
Dim CurX As Single
Dim Tolerance As Single
Dim Count As Integer

CurX = X
Tolerance = 0 0000'
Count = 0

Do While True
  Count = Count + '
  NewX = 0 5 * (CurX + X / CurX)
  If (Abs(CurX - NewX) > Tolerance) Then
    CurX = NewX
  Else
    Exit Do
  End If
Loop

SQRoot = NewX

End Function
```

Figure 14. Execution Nodes defined for the Square Root Component.

The source code instrumentation adds a set of instructions to the source code as shown in Table 9.

Component Source Code
Public Function SQRoot(X As Integer) As Single
Dim NewX As Single
Dim CurX As Single
Dim Tolerance As Single
Dim Count As Integer
Start_Test_Case(X)
CurX = X
Tolerance = 0.00001
Count = 0
Visited_node("A")
Do While True
Count = Count + 1
NewX = 0.5 * (CurX + X / CurX)
If (Abs(CurX - NewX) > Tolerance) Then
CurX = NewX
Visited_node("B")
Else
Visited_node("C")
Exit Do
End If
Loop
SQRoot = NewX
Visited_node("D")
End_Test_Case(X)
End Function

Table 9. Square Root Component Instrumented Source Code.

Table 9 shows the instrumented source code. The added instructions are highlighted using bold format. These instructions were implemented as an external monitoring function that runs at the same time for every test case execution. This example was instrumented manually.

The execution of all test cases is the next task after the source code instrumentation. The information gathered during the testing process is included in the corresponding Component Test Profile. Table 10 shows part of the resulting Component Test Profile produced after executing all test cases.

Component Test Profile		
Component Name	Square Root Component	
Component Functionality	**Functionality Signature**	**Domain Description**
Square root computation function	Sqroot(*input*): *output* *input*: Parameter whose square root will be computed *output*: Return value that represents the computed square root value	**Theoretical** $input \in \aleph$ $output \in \Re$ **Implemented** $input \in [0...65535]$ $output \in [0,255.998047]$

Test Data

Test Case Number	**Input Value**	**Expected Output Value**	**Execution Trace**	**Test Result**
1	1	1	ABCD	Correct
2	43	6.557438	ABBBBBBBCD	Correct
3	49	7	ABBBBBBBCD	Correct
4	55	7.416198	ABBBBBBBCD	Correct
5	56	7.483315	ABBBBBBBCD	Correct
6	58	7.615773	ABBBBBBBCD	Correct
7	84	9.165152	ABBBBBBBBCD	Correct
8	95	9.746795	ABBBBBBBBCD	Correct
9	106	10.29563	ABBBBBBBBCD	Correct
10	117	10.81665	ABBBBBBBBCD	Correct
11	123	11.09054	ABBBBBBBBCD	Correct
12	130	11.40175	ABBBBBBBBCD	Correct
13	134	11.57584	ABBBBBBBBCD	Correct
14	150	12.24745	ABBBBBBBBCD	Correct
. . .	. . .	. . .	. . .	. . .

Table 10. Square Root Component Test Profile.

Table 10 shows two levels of information. One level corresponds to the meta-data information about the component that includes the domain description and the functionality description. The second level includes all test data, test results, and their corresponding execution trace information.

4.4.2 Face Detection Component Test Profile

This example describes a more complex component. This component is capable of extracting a section of an image containing information that is interpreted as a face. The artifacts provided for the construction of the component test profile are:

- Component Specification
- Test Cases
- Component Source Code

This component requires an image as input data. However, the physical or natural properties associated with such image can not be used easily to characterize every test case (i.e., physical information does not provide enough information to differentiate every image).

The strategy for overcoming this problem is to define a differentiation ontology associated with the application domain. This technique is explained in Section 5.2.2. In this case, the set of attributes associated with the image are shown in Figure 15.

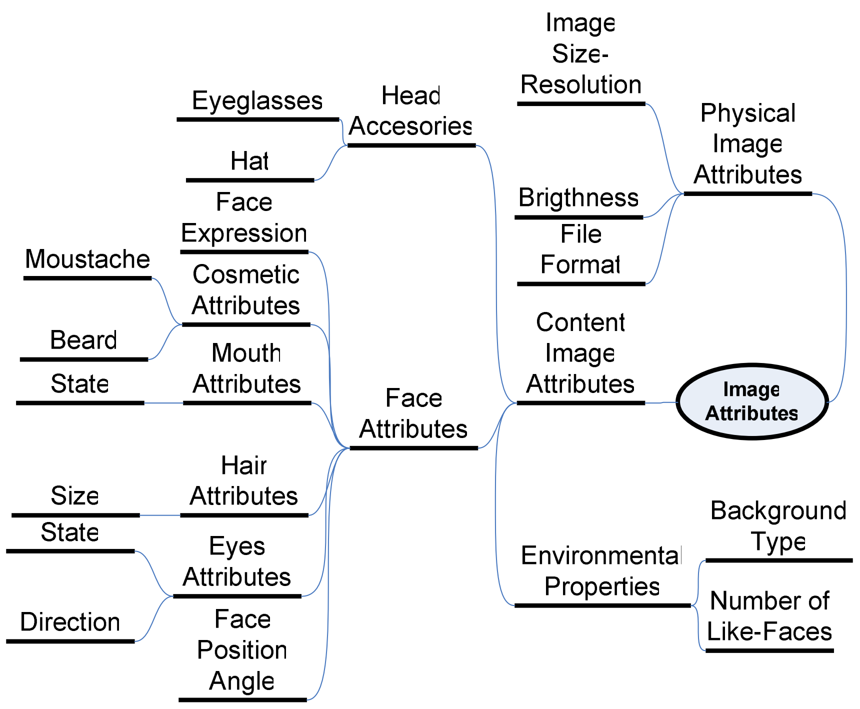


Figure 15. Ontology for the Face Detection Component Input Domain.

Figure 15 shows a diagram describing a set of attributes to differentiate the images used in the test cases. The ellipse denotes the root of all attributes. The final leaves denote specific attributes (e.g., eyeglasses, beard, brightness, among others.) while intermediate leaves denote groups of attributes (e.g., cosmetic attributes, content image attributes, among others). Table 11 shows a description of all the attributes defined in the ontology.

Attribute	**Description**
Format	Image file format, such as JPG, BMP, GIF
Brightness	Image brightness ranging from 0 to 255
Width	Image width ranging from 1 to 1024 pixels
Height	Image height ranging from 1 to 1024 pixels
Number of like faces	Number of like-faces in the image
Background type	Background complexity characterized by three possible values: simple, medium, and complex
Face position	Face position alignment characterized by four possible values: left, right, straight, up, and down
Eyes state	Eyes state characterized by three possible values: open, semi closed, and closed
Eyes direction	Eyes direction characterized by four possible values: left, right, straight, up, and down
Hair size	Hair size characterized by none, short, normal, long, and extreme long
Mouth state	Mouth state characterized by three possible values: one, semi closed, and closed
Moustache	Moustache presence, characterized by two possible values: yes, and no
Beard	Beard presence, characterized by two possible values: yes and no
Face expression	Face expression characterized by four possible values: angry, happy, sad, and normal
Eyeglasses	Eyeglasses presence, characterized by two possible values: yes and no
Hat	Hat presence, characterized by two possible values: yes and no
Gender	Gender characterized by two possible values: male and female

Table 11. Attribute Description for the Face Detection Component.

Then, every test case is defined as an input image and its set of attributes. Table 12 shows a test case and its corresponding attributes.

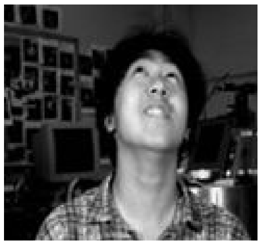
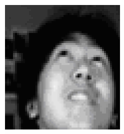
Test Case	01_031	
	INPUT	OUTPUT
Preview		
Image File Name	Img_0323.jpg	Img_0323.pic
Format	BMP	PIC
Width	128	80
Height	120	80
Number of like faces	1	1
Background type	Complex	NA
Face position	Up	Up
Eyes state	Open	Open
Eyes direction	Up	Up
Hair size	Normal	Normal
Mouth state	Semi closed	Semi closed
Moustache	No	No
Beard	No	No
Face expression	Happy	Happy
Eyeglasses	No	No
Hat	No	No
Gender	Male	Male
Test Detection	NA	True

Table 12. Test Case Defined for the Face Detection Component.

The component specification is shown in Table 13. This specification is based on the previous ontology and the input-output attribute transformation. An attribute transformation describes how an attribute is changed after the component performs an operation.

Component Specification		
Component Name	Face Detection Component	
Component Functionality	**Functionality Signature**	
Face-like detection	Face_Detection(*raw_image*): *face_found, detection* *raw_image*: Image parameter that is used as source for the face detection *face_found*: Image that is returned and it contains the face detected *detection*: Return value that specifies if the detection was successful	
ATTRIBUTE TRANSFORMATION		
Attribute	**Input Domain**	**Output Domain**
Format	{BMP, JPG, GIF}	{PIC}
Width	1 < width < 35536	width-output ≤ width
Height	1 < height < 35536	height-output ≤ height
Number of like faces	5	1
Background type	{simple, medium, complex}	{simple, medium, complex}
Face position	{left, right, straight, up, down}	{left, right, straight, up, down}
Eyes state	{open, semi closed, closed}	{open, semi closed, closed}
Eyes direction	{left, right, straight, up, down}	{left, right, straight, up, down}
Hair size	{none, short, normal, long, extreme long}	{none, short, normal, long, extreme long}
Mouth state	{open, semi closed, closed}	{open, semi closed, closed}
Moustache	{yes, no}	{yes, no}
Beard	{yes, no}	{yes, no}
Face expression	{angry, happy, sad, normal}	{angry, happy, sad, normal}
Eyeglasses	{yes, no}	{yes, no}
Hat	{yes, no}	{yes, no}
Gender	{male, female}	{male, female}
Test Detection	NA	{success, failure}

Table 13. Face Detection Component Specification.

The Component Test Profile produced after the testing process includes all the previous information (i.e., test cases and component specification) as well as the collected execution traces and the test case outcomes. Table 14 shows a partial list of the additional information included in the component test profile of this component.

Test Case Number	**Test Case Verdict**	**Execution Trace**
01_031	Success	A1A2A5A7C7C8C9D2D3D3D3D3D3D3D3D3D3D3D3D3D3D4D5D5D5D5D5D5D5D6D7D6D9E1E3E4
01_032	Success	A1A2A5A7C7C8C9D2D3D3D3D3D3D3D3D3D3D3D3D3D3D4D5D5D5D5D5D5D5D5D5D6D7D6D7D9D8E1E3E4
01_033	Success	A1A2A5A7C7C8C9D2D3D3D3D3D3D3D3D3D3D3D3D3D3D4D5D5D5D5D5D5D6D7D6D7D9D8E1E3E4
01_034	Success	A1A2A5A7C7C8C9D2D3D3D3D3D3D3D3D3D3D3D3D3D3D4D5D5D5D5D6D7D6D9E1E3E4
01_035	Success	A1A2A5A7C7C8C9D2D3D3D3D3D3D3D3D3D3D3D3D3D3D4D5D5D6D7D9E1E3E4
01_036	Success	A1A2A5A7C7C8C9D2D3D3D3D3D3D3D3D3D3D3D3D3D3D4D5D5D5D5D6D7D6D9E1E3E4
01_037	Success	A1A2A5A7C7C8C9D2D3D3D3D3D3D3D3D3D3D3D3D3D3D4D5D5D5D5D5D6D7D9E1E3E4
01_038	Success	A1A2A5A7C7C8C9D2D3D3D3D3D3D3D3D3D3D3D3D3D3D4D5D5D5D5D5D5D5D5D6D7D6D9E1E3E4
.	.	.
.	.	.
.	.	.

Table 14. Partial Information of the Face Detection Component Test Profile.

4.5 Tool Support and Implementation Issues

This section discusses relevant issues about the tools that were used to perform the source code instrumentation. Additionally, the most important aspects of the monitoring system used for recording the execution information are discussed.

4.5.1 Source Code Instrumentation

The source code instrumentation may be performed manually or automatically. One of the goals of the instrumentation process is to minimize the impact of such activity on the testing process. Automatic analysis tools are a key element to reduce the overhead.

Since the execution abstraction is based on execution paths, the instrumentation procedure can be performed by analyzing the source code. This analysis focuses on finding the different alternatives where the execution may branch in two or more different paths. Thus, the source code analyses tool looks for branching or looping statements.

The tool that performs the automatic source code instrumentation for Java programs are developed using the Eclipse JDT framework. It provides APIs to manipulate a given Java source code, detect errors, perform compilations, and launch programs (Foundation). Eclipse's JDT has its own Document Object Model (DOM) in the same spirit of the well-known XML DOM: the Abstract Syntax Tree (AST).

The source code navigation is simplified. It is similar to the XML document navigation used by the AST API. This simplification reduces the analysis complexity since these mechanisms hide the complexity of scanning and parsing the source code.

Figure 16 shows the typical workflow followed when AST is used (Kuhn and Thomann 2007). This flow starts by providing a component source code (i.e., step 1). Next, the source code is parsed with the class org.eclipse.jdt.core.dom.ASTParser (i.e., step 2). The product from this step is the abstract syntax tree of the source code (i.e., step 3). Depending on the specific needs, the AST may be manipulated by directly modifying the AST or by noting the modifications in a separate protocol. This protocol is handled by an instance of ASTRewrite (i.e., step 4a and 4b). Next, the modification in the source code are applied (i.e., step 5). Finally, it is required to use a wrapper for the source code (i.e., step 6).

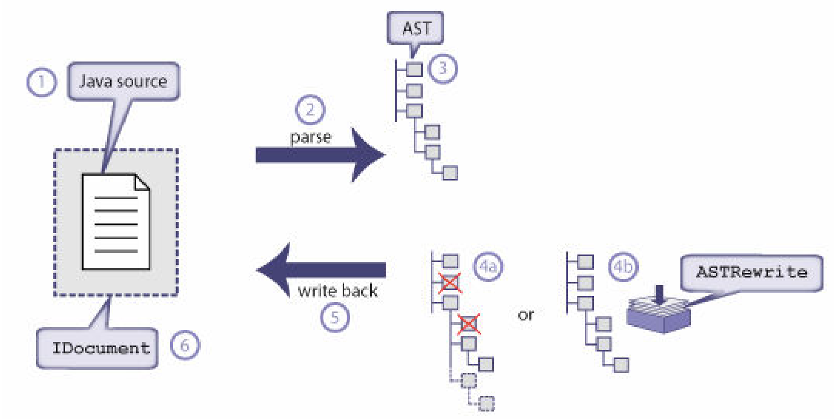


Figure 16. Typical Workflow using the AST (Kuhn and Thomann 2007).

Using this API, the source code instrumentation for java programs is easily implemented. However, the instrumentation for other systems having distinct programming language was performed manually.

4.5.2 Execution Monitoring

The execution monitoring is performed by an external system that is running at the same time as the component under test. This system is called the Execution Trace Monitoring System (ETMS). The architecture of this monitoring system is shown in Figure 17.

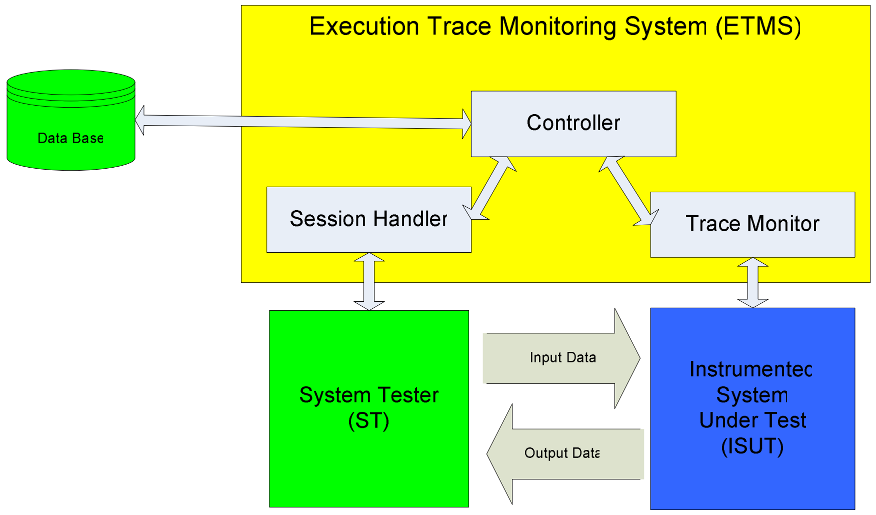


Figure 17. Execution Trace Monitoring System.

The main ETMS components are the controller, the session handler, and the trace monitor component. The session handler component is the interface with the system that tests the component. This component also manages different sessionS for different test cases. The trace monitor component is the listener of all visited execution nodes. This component collects this information during the test case execution and organizes such information. The controller component is the core element of the system. This component coordinates the other components and interacts with the persistent repository (i.e., the database). The bidirectional arrows in the diagram denote information and control exchange.

4.6 Discussion

This chapter has described two elements addressing the supporting claim: "*Execution trace information can be collected from the testing process.*" The first element, the Component Test Profile artifact, provides support for storing, organizing, and analyzing test execution information. The second element, the execution monitoring infrastructure provides a run time environment to collect execution traces.

The cost of building test profiles can be reduced if most of the information produced during the testing process is reused. The part of the component test profile where the component functionality is specified can be obtained from the source code or the regular system documentation. If this is the case, then the additional required effort required is the instrumentation.

The use of instrumented code is valid when it can be assumed that this additional code does not have side effects or does not impact the actual execution. This assumption needs to be analyzed on a case by case basis since there are technological platforms where the additional code may have a considerable impact. For example, there are embedded systems where the size of the source code is an important limitation in the system under development. Additional statements may impact this type of constraints.

There are cases where the operating system is able to create profiles during the execution of any program. These profiles have been used mostly for debugging, performance assessment (Ofelt and Hennessy 2000), and program maintenance (Reps et al., 1997). A future research direction is to extend one of such profilers to address the monitoring of execution traces.

The current approach to abstract the execution is based on grouping statements based on control-flow criteria. There are some drawbacks in this approach that must be considered for future improvements. For example, the control-flow analysis does not consider the number of statements defined within any block; it is possible to have a considerable number of statements within one block and to have another block with just one statement. From a reliability perspective, there are more chances to find a failure in the block having more number of statements than in that having just one statement.

4.7 Summary

This chapter presented evidence supporting the claim: "*Execution trace information can be collected from the testing process.*" The first phase of the Test Profile Analysis Framework provides the solutions of the problems derived from this claim**.**

The solution includes the definition of a new artifact that is produced during the component testing process as well as the implementation of tools which partially automate the task defined in the Component Testing Phase.

The contributions of this chapter include:

- A definition of a new artifact, the Component Test Profile, which collects information from the testing process. This artifact is based on a metamodel that can be extended to fit other needs of information.
- A model providing an abstract view of the component's execution during the testing process. This model is based on previous works in the area of control flow analysis.
- A solution to the problem of automatic source code instrumentation for Java programs. This solution is based on using libraries that provide a high level view of any source code program similarly to an XML document.
- The design and implementation of a system that is capable of monitoring instrumented component source code during the component testing process.

The next chapter will address the claim about the need for techniques to model operational information to facilitate the reliability assessment.

Chapter 5

Modeling Operational Information

This chapter addresses the supporting claim: "*Operational information can be specified in a well-suited form that improves the reliability assessment.*" Two major issues have to be addressed: a notation to specify operational information and a technique to model complex domains. The evidence that supports this claim is described in this chapter.

The Operational Definition Phase is the proposed solution to the issues mentioned above. Figure 18 shows the location of this phase in the framework. The output artifact produced in this phase is the Assembly Operational Profile. This artifact is the practical solution supporting the claim addressed in this chapter.

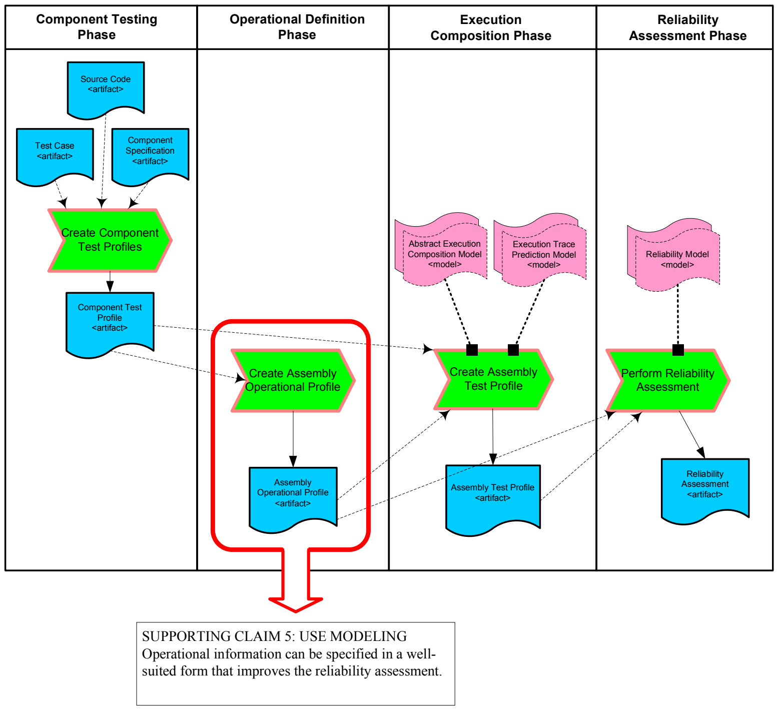

Component Testing Phase
Operational Definition Phase
Execution Composition Phase
Reliability Assessment Phase
Source Code <artifact>
Test Case <artifact>
Component Specification <artifact>
Create Component Test Profiles
Component Test Profile <artifact>
Create Assembly Operational Profile
Assembly Operational Profile <artifact>
Abstract Execution Composition Model <model>
Execution Trace Prediction Model <model>
Create Assembly Test Profile
Assembly Test Profile <artifact>
Reliability Model <model>
Perform Reliability Assessment
Reliability Assessment <artifact>
SUPPORTING CLAIM 5: USE MODELING
Operational information can be specified in a well-suited form that improves the reliability assessment.

Figure 18. Framework Expanded View with Focus on the Second Phase.

The content of this chapter is structured as follows: First, a detailed description of the problem is presented. Second, the rationale behind the solution is explained. Third, a description of the task and artifacts used in this phase is presented. Fourth, a set of examples is presented to illustrate the solution proposed in this chapter. Fifth, a description of the notation to specify operational information and the technique to characterize complex domains are described. Finally, results, limitations and future extension of the proposed solution are discussed.

5.1 Problem Description

Software reliability estimations are usually weighted using information representing the expected use of the system. These conditions are described using an artifact called the operational profile. Software reliability engineering (Musa 2004b) proposes the use of operational information (i.e., quantitative characterization of the expected use of the system) to guide the testing process as well as the estimation of the software reliability.

Although there is a general consensus regarding the benefits of using operational profiles, there is no standard form for modeling or documenting such kind of information. There is a consensus about the importance and relevance during the development process but the only agreed characteristic is the quantitative role that operational profiles should have.

The lack of a standard approach for specifying operational profiles has been addressed by (Gittens et al., 2004) who proposed an "extended operational profile model." The extension proposes to include information such as the environmental conditions associated with the expected use and the data constraints associated with the functionalities. This work, however, proposes a theoretical extension rather than providing practical experiences.

Furthermore, some domains are difficult to characterize since their physical representations do not have enough semantic content. This lack of semantic content makes it difficult to differentiate elements in such domains. For example, the physical representation of an image does not provide enough information to differentiate images for face recognition purposes. This issue requires techniques to characterize such kind of domains beyond the notation used for specifying them.

The issues mentioned before suggest the need for both, a method or technique, to characterize complex domains and a notation to model operational information addressing the usage heterogeneity found in several systems.

The problem addressed in this chapter is the definition of an operational modeling solution (i.e. notation) and a technique to support the use of such a notation to be used in the framework. This notation should include a way for specifying the components in the assembly. It is assumed that a description of the assembly (i.e., an assembly specification) and the quantitative information required for the operational modeling are available.

The next section describes the approach for solving the issues mentioned above.

5.2 Solution Approach

The solution includes a notation to model operational information and a technique to characterize complex domains.

5.2.1 Modeling Operational Information: UML Operational Characterization Profile

The need for a standard notation to model operational information is addressed by creating an extension for the Unified Modeling Language (UML), known as UML Profiles (OMG 2002). The extension described in this chapter is named as *UML Operational Characterization Profile*.

UML is an industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems standardized by the Object Management Group (OMG 2007). UML simplifies the complex process of software design by using "blueprints" for software construction.

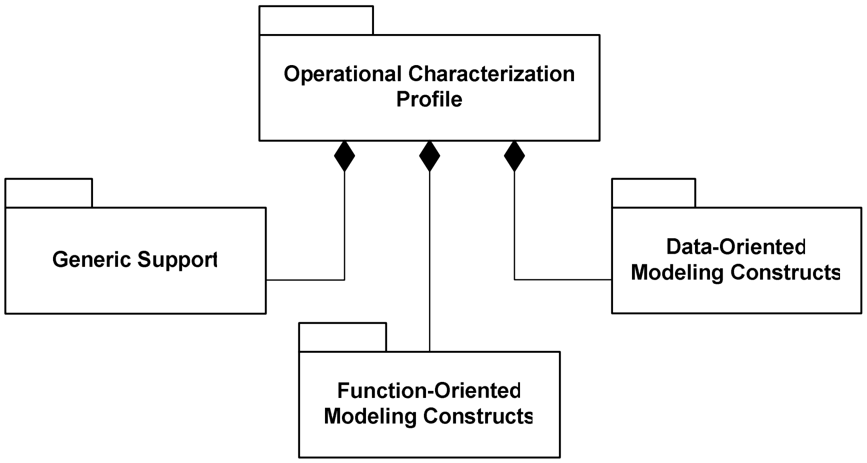
A UML profile is a modeling extension of UML focusing on a specific domain. This kind of extensions is allowed in the UML language. Examples of UML profiles are the UML Testing Profile (OMG 2005b), UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (OMG 2006b), UML Profile for Schedulability, Performance and Time (OMG 2005a), UML Profile for System on a Chip (SoC) (OMG 2006c), among others.

A UML profile is defined by a set of domain-oriented concepts (i.e., represented by metaclasses labeled with specific stereotypes), a set of relationships among these concepts, a set of rules about how these concepts may be combined to represent models, and proper semantics required for such models. All this information is specified by using UML and the Object Constraint Language (OCL) (OMG 2006a).

UML is capable of modeling and specifying operational information by using basic modeling constructs. However, there is a need for standardized constructs and notations that can help to facilitate the analysis of operational information.

Operational information can be characterized from different perspectives such as a data-oriented perspective, a function-oriented perspective, or some combination of these The selection of a perspective depends on the specific system implementation.

The UML Operational Characterization Profile is organized in three logical groups of concepts that are stored in three different UML packages: A package for data-related information, a package for functional information, and a package for generic support. The package organization is shown in Figure 19. This figure uses UML notation to denote the package organization (i.e., the lines with a diamond specify that the package connected at one end is part of the package on the solid-diamond side of the line.


Operational Characterization Profile
Generic Support
Data-Oriented Modeling Constructs
Function-Oriented Modeling Constructs
Figure 19. Package structure within the UML Operational Characterization Profile.

The Data-Oriented Modeling Constructs package contains modeling constructs to model operational information at the data level, the Function-Oriented Modeling Constructs package contains modeling constructs to model operational information at the function level, and the Generic Support package contains generic construct providing support and consistency to all other packages.

The most important modeling constructs defined within the data-oriented package are the domain and the data constraint. A data constraint represents a condition that defines a subset over the entire domain associated with some specific data. A domain is a set of all possible values that any particular component may require or produce.

Similarly, the most important modeling constructs defined in the function-oriented package are the operational mode, the operation, and the function. An operational mode is a set of major logical operations that are constrained by specific conditions. An operation is a major logical system task. A function is an atomic functionality performed by a system. The combination of one or more functions defines an operation.

In the same way, the most important modeling constructs defined in the generic support package are the operational profile, the scenario, and the environmental constraint. An environmental constraint is a condition about the parameters defining the system environment (i.e., the conditions about the system execution). A scenario is a typical interaction between the user and the system or between two software components.

The previous modeling constructs (i.e., metaclasses) are used to create models representing operational information. Section 5.5.1 details the UML profile and its organization. Appendix C includes the complete UML Profile specification.

5.2.2 Modeling Complex Operational Domains

Some domains have semantic information that is not represented in the data physical storage. Images are examples of such kind of domains. The physical attributes of an image are the size and the value for every pixel within the image. Face recognition applications uses images as input parameters. It is not feasible to use physical attributes to differentiate the images contained in the test cases.

We propose a method to address the problem of modeling complex domains by constructing a differentiation ontology. A differentiation ontology has a set of domain-related attributes that are used to differentiate elements from such domain. This method, called the Domain Characterization Method (DCM), is composed of seven steps arranged in an iterative cycle as shown in Figure 20.

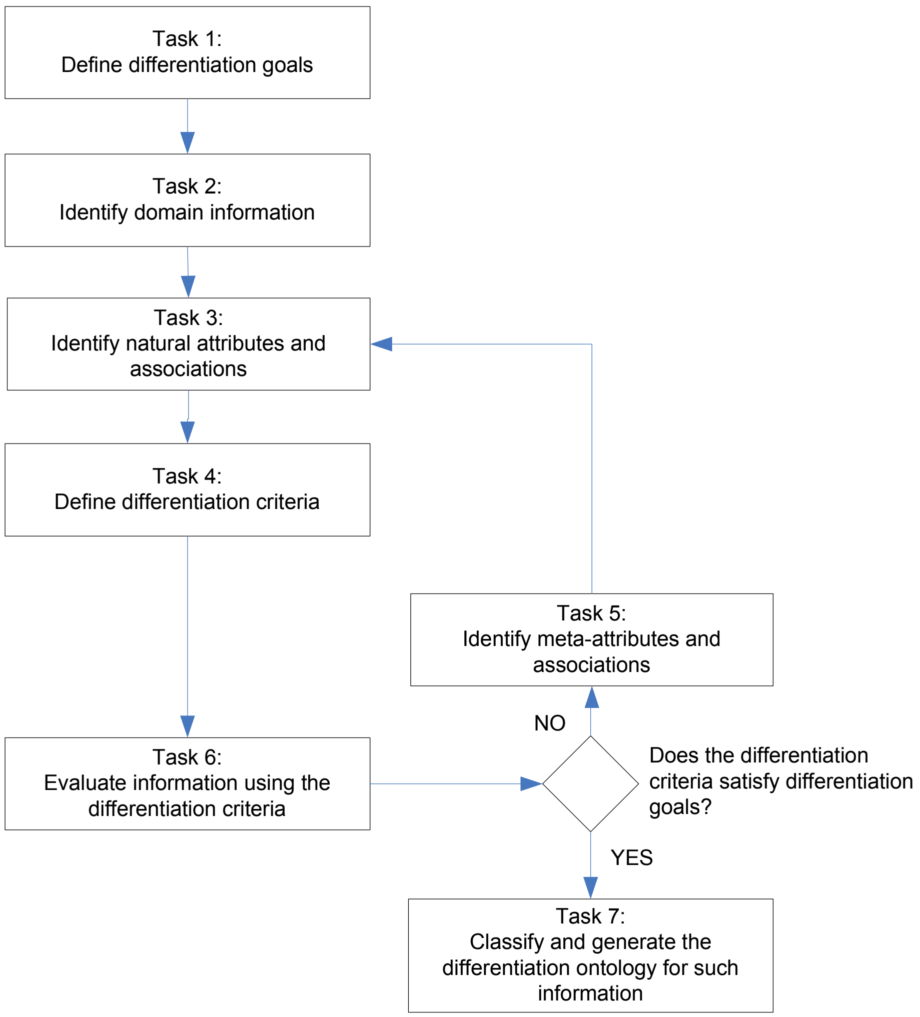

Task 1:
Define differentiation goals
Task 2:
Identify domain information
Task 3:
Identify natural attributes and associations
Task 4:
Define differentiation criteria
Task 5:
Identify meta-attributes and associations
NO
Task 6:
Evaluate information using the differentiation criteria
Does the differentiation criteria satisfy differentiation goals?
YES
Task 7:
Classify and generate the differentiation ontology for such information

Figure 20. Steps in the Domain Characterization Method.

The DCM follows an approach based on cyclic refinements guided by the achievement of differentiation goals. Differentiation criteria represent a set of attributes which are used to classify information for differentiation purposes. A more detailed discussion of DCM is presented in Section 5.5.2.

5.3 Solution Description

This section describes how the ideas in the previous section are implemented in the framework. First, a description of the artifacts is provided. Next, a description of the task required for the assembly operational profile construction is explained.

5.3.1 Artifacts Description

The inputs artifacts required in this phase are a set of component test profiles. The output artifact is the assembly operational profile. The component test profiles are the building elements required to define the assembly.

The *Assembly Operational Profile* artifact contains information regarding the components that are required for the assembly, the functionalities provided by the assembly, and the quantitative characterization of the usages of such functionalities.

The assembly operational profile is a key artifact in the framework because it contains the information to weight the reliability assessment. Domains with more expected operational use have more weight in the final reliability estimate. This operational-based weighting produces more useful estimations from a pragmatic perspective, since the reliability estimate depends on the expected usages of the assembly.

A template for every artifact is provided in Appendix B.

5.3.2 Task: Create Assembly Operational Profile

The *Create Assembly Operational Profile* task is performed during the second phase of the framework. The purpose of this task is the generation of the operational information required for the reliability assessment as well as the specification of the components included in the assembly.

The activities involved in this task are the specification of operational information using the *UML Operational Characterization Profile* and the characterization of domains. It is assumed that quantitative information required for the operational model is available (i.e., the subdomain operational probability is known). This task can not be performed automatically since it requires the construction of models rather than the processing of information.

5.4 Examples

This section describes three examples to illustrate the activities performed in the phase. The first example focuses on an assembly composed of one component. The second example is an assembly representing a security monitoring system. The technique to model complex operational domains is used in this example. The third example is an assembly representing a web-based library system.

5.4.1 Square Root Component Operational Profile

This example presents the operational profile of a basic assembly having one component. This assembly requires a data-oriented characterization since there is only one functionality or operation defined for such assembly. Table 15 shows the assembly operational profile. This representation has a corresponding UML model which is shown in Figure 21.

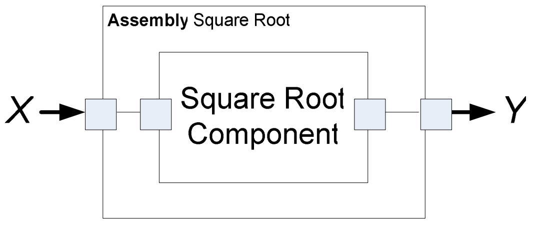
Assembly Operational Profile					
Assembly Name	Square Root Assembly				
Purpose	Estimate the square root of the most frequent entries in the system.				
Assembly Specification					
Quantitative Model					
Domain Description	**Subdomain Description (*X*)**				**Comments**
		Lower	**Upper**	**Probability**	
Theoretical $X \in \aleph$ $Y \in \Re$	Subdomain 1	0	1000	0.15	
	Subdomain 2	1001	5000	0.4	
Implemented $X \in [0...65535]$ $Y \in [0,255.998047]$	Subdomain 3	5001	20000	0.35	
	Subdomain 4	20001	65535	0.1	

Table 15. A Data-oriented Assembly Operational Profile.

Table 15 shows the information derived from the operational information defined in the operational model shown in Figure 21. The first part of the table shows descriptive items such as the assembly name, the assembly purpose, and the assembly specification (i.e., required components). The second section, named as quantitative model, contains the most relevant operational information such as domain description, subdomain descriptions, and their corresponding operational probabilities.

Figure 21 shows the corresponding UML model which is based on the UML Operational Characterization Profile. The model includes information that is not shown in the assembly operational profile. For example, the model shows the hierarchical relationship between the modeling constructs such as the relationship between operation and operational profile, the relationship between domain and subdomains. The constraint defined for the measurement consistency model assumes that the sum of all probabilities is equal to 1.

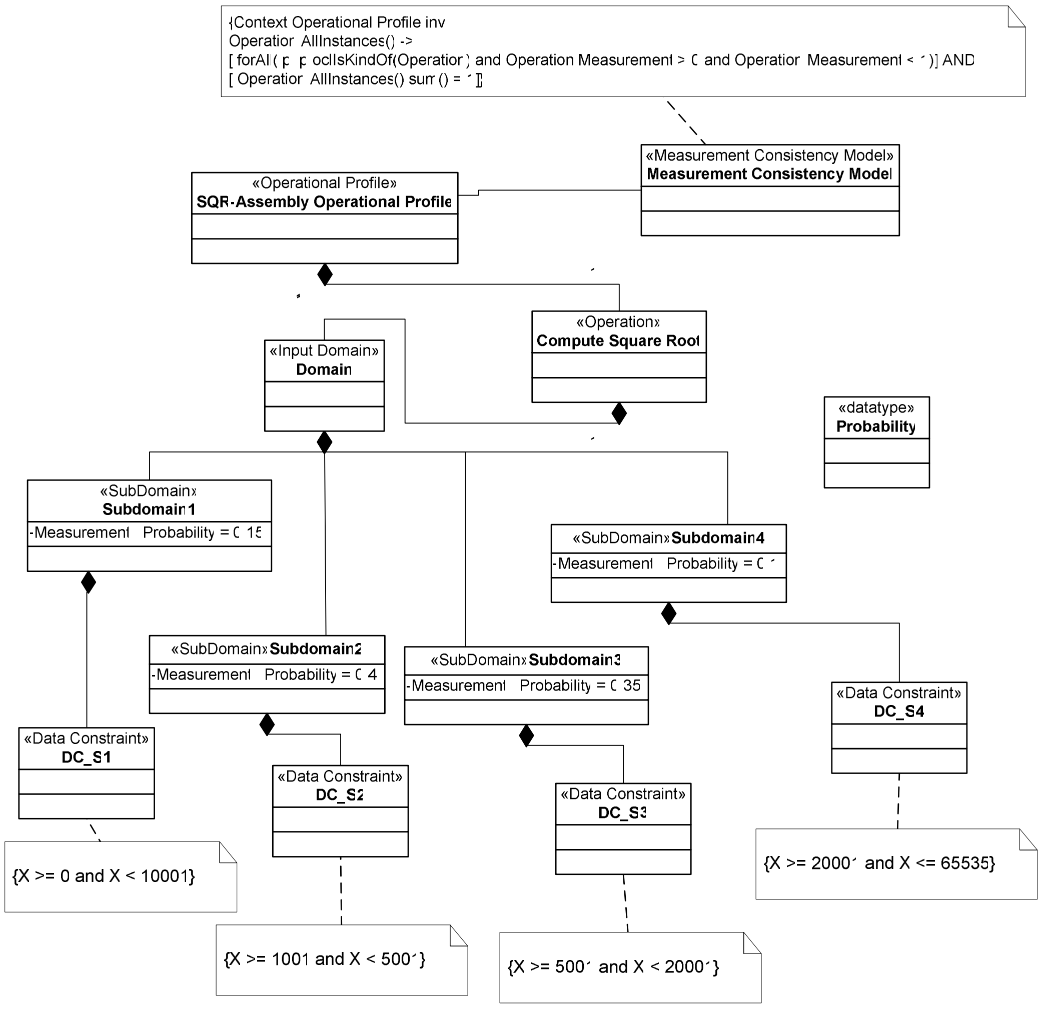

{Context Operational Profile inv
Operatior AllInstances() ->
[forAll(p p oclIsKindOf(Operatior) and Operation Measurement > C and Operatior Measurement < ´)] AND
[Operatior AllInstances() sum () = ´]}
«Measurement Consistency Model»
Measurement Consistency Model
«Operational Profile»
SQR-Assembly Operational Profile
«Operation»
Compute Square Root
«Input Domain»
Domain
«datatype»
Probability
«SubDomain»
Subdomain1
-Measurement Probability = C 15
«SubDomain» Subdomain4
-Measurement Probability = C ´
«SubDomain» Subdomain2
-Measurement Probability = C 4
«SubDomain» Subdomain3
-Measurement Probability = C 35
«Data Constraint»
DC_S1
«Data Constraint»
DC_S2
«Data Constraint»
DC_S3
«Data Constraint»
DC_S4
{X >= 0 and X < 10001}
{X >= 1001 and X < 500´}
{X >= 500´ and X < 2000´}
{X >= 2000´ and X <= 65535}

Figure 21. UML Model defined for the Operational Profile shown in Table 15.

5.4.2 Security Monitoring System Operational Profile

This example describes the operational profile specified for a system that is represented by an assembly. The goal of this system is to monitor the user access to different secure locations.

This assembly uses as input images that are complex to characterize using only their physical attributes. This issue is addressed using the technique to model complex operational domains. The resulting differentiation ontology is shown in Figure 22. The attributes defined for this ontology were obtained from experts in the area of face recognition and relevant characteristics in the operational environment of the assembly. For example, experts describe that image features such as eyes and mouth represent relevant attributes associated with face recognition algorithms. Operational environment has relevant attributes such as the use of eye glasses and the hair size. The final ontology combines attributes from these two sources.

The interpretation of symbols used in Figure 22 is the same as Figure 15 in Chapter 4. The ellipse denotes the root of all attributes. Final leaves denote specific attributes (e.g., eyeglasses, beard, and brightness) while intermediate leaves denote groups of attributes (e.g., cosmetic attributes and content image attributes). The difference between Figure 22 and Figure 15 is the number of attributes that impact the data differentiation.

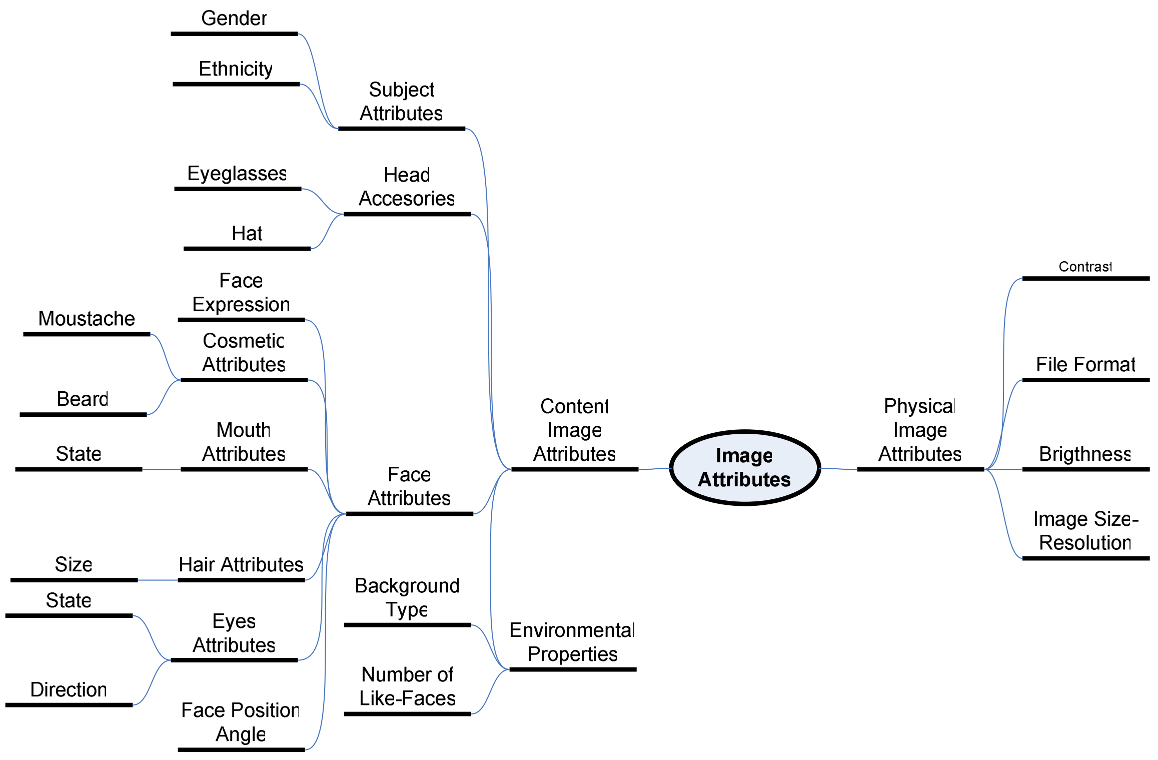
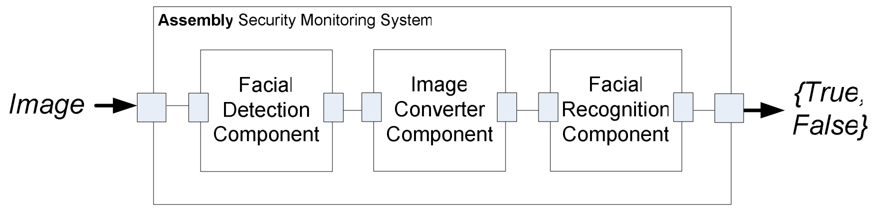


Figure 22. Differentiation Ontology defined for the Image Input Domain.

The assembly operational profile defined for this application is shown in Table 16. This table shows in tabular form the operational model depicted in Figure 23.

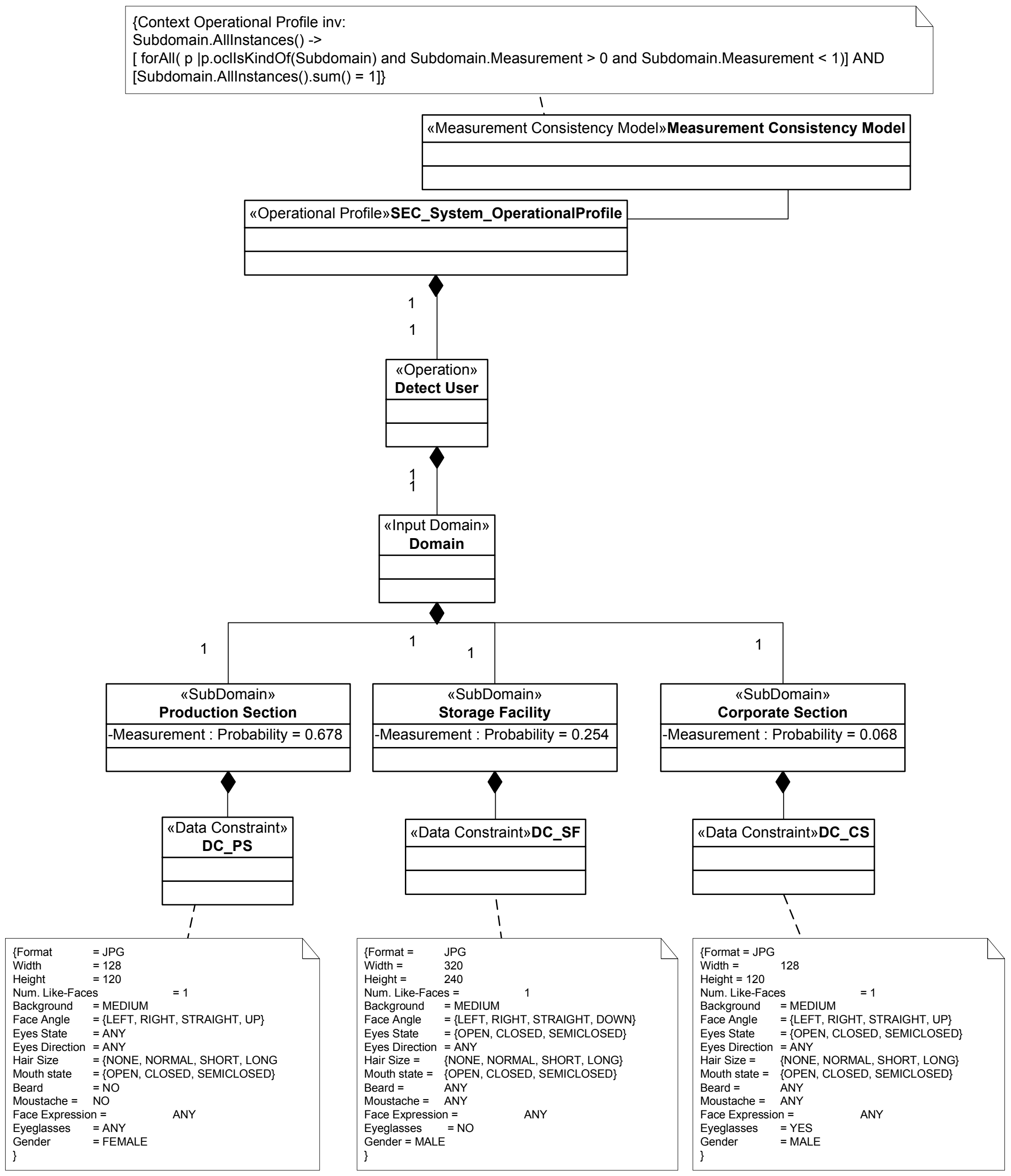
Assembly Operational Profile	
Assembly Name	Security Monitoring System
Purpose	Detect if users are allowed to stay at particular locations.
Assembly Specification	 Assembly Security Monitoring System Image Facial Detection Component Image Converter Component Facial Recognition Component {True, False} When the output is True, the user is allowed to stay at the location.

Quantitative Model			
Domain Description	**Subdomain Description**		**Comments**
	Subdomain Name	**Probability**	
Image	Corporate Location	0.254	This subdomain corresponds to the corporate offices.
	Storage Location	0.068	This subdomain corresponds to the storage facility.
	Production Location	0.678	This subdomain corresponds to the production section.

Subdomain Characterization

	Corporate Location	**Production Location**	**Storage Location**
Format	JPG	JPG	JPG
Width	128	128	320
Height	120	120	240
Num. Like-Faces	1	1	1
Background	MEDIUM	MEDIUM	MEDIUM
Face Angle	{LEFT, RIGHT, STRAIGHT, UP}	{LEFT, RIGHT, STRAIGHT, UP}	{LEFT, RIGHT, STRAIGHT, DOWN}
Eyes State	{OPEN, CLOSED, SEMICLOSED}	ANY	{OPEN, CLOSED, SEMICLOSED}
Eyes Direction	ANY	ANY	ANY
Hair Size	{NONE, NORMAL, SHORT, LONG}	{NONE, NORMAL, SHORT, LONG	{NONE, NORMAL, SHORT, LONG}
Mouth state	{OPEN, CLOSED, SEMICLOSED}	{OPEN, CLOSED, SEMICLOSED}	{OPEN, CLOSED, SEMICLOSED}
Beard	ANY	NO	ANY
Moustache	ANY	NO	ANY
Face Expression	ANY	ANY	ANY
Eyeglasses	YES	ANY	NO
Gender	{MALE}	{FEMALE}	{MALE}

Table 16. An Assembly Operational Profile containing a Complex Operational Domain.


{Context Operational Profile inv:
Subdomain.AllInstances() ->
[forAll(p |p.oclIsKindOf(Subdomain) and Subdomain.Measurement > 0 and Subdomain.Measurement < 1)] AND
[Subdomain.AllInstances().sum() = 1]}
«Measurement Consistency Model»Measurement Consistency Model
«Operational Profile»SEC_System_OperationalProfile
1
1
«Operation»
Detect User
1
1
«Input Domain»
Domain
1
1
1
1
«SubDomain»
Production Section
-Measurement : Probability = 0.678
«SubDomain»
Storage Facility
-Measurement : Probability = 0.254
«SubDomain»
Corporate Section
-Measurement : Probability = 0.068
«Data Constraint»
DC_PS
«Data Constraint»DC_SF
«Data Constraint»DC_CS
{Format = JPG
Width = 128
Height = 120
Num. Like-Faces = 1
Background = MEDIUM
Face Angle = {LEFT, RIGHT, STRAIGHT, UP}
Eyes State = ANY
Eyes Direction = ANY
Hair Size = {NONE, NORMAL, SHORT, LONG
Mouth state = {OPEN, CLOSED, SEMICLOSED}
Beard = NO
Moustache = NO
Face Expression = ANY
Eyeglasses = ANY
Gender = FEMALE
}
{Format = JPG
Width = 320
Height = 240
Num. Like-Faces = 1
Background = MEDIUM
Face Angle = {LEFT, RIGHT, STRAIGHT, DOWN}
Eyes State = {OPEN, CLOSED, SEMICLOSED}
Eyes Direction = ANY
Hair Size = {NONE, NORMAL, SHORT, LONG}
Mouth state = {OPEN, CLOSED, SEMICLOSED}
Beard = ANY
Moustache = ANY
Face Expression = ANY
Eyeglasses = NO
Gender = MALE
}
{Format = JPG
Width = 128
Height = 120
Num. Like-Faces = 1
Background = MEDIUM
Face Angle = {LEFT, RIGHT, STRAIGHT, UP}
Eyes State = {OPEN, CLOSED, SEMICLOSED}
Eyes Direction = ANY
Hair Size = {NONE, NORMAL, SHORT, LONG}
Mouth state = {OPEN, CLOSED, SEMICLOSED}
Beard = ANY
Moustache = ANY
Face Expression = ANY
Eyeglasses = YES
Gender = MALE
}
Figure 23. Operational Model Defined for the Security Monitoring System Assembly.

The explanation of the information provided in Figure 23 and Table 16 is similar to the square root example. However, this example includes the use of the differentiation ontology to characterize every subdomain.

5.5 Tool Support and Implementation Issues

This section provides a more detailed description of the UML Operational Characterization Profile and a description of the technique used to characterize complex domains.

5.5.1 UML Operational Characterization Profile

The use of UML profiles starts by creating a document that defines the profile. The elements included in a UML profile definition are an overview of the profile structure, a list of concepts defined for the domain, the structure of meta-classes supporting such concepts, the organization of metaclasses and their relationships, a description of every metaclass and its constraints, and a set of illustrative examples.

The UML Operational Characterization Profile has a set of stereotypes for each UML metaclass. Additionally, this profile has been designed to define a set of new stereotyped modeling elements instead of defining new kinds of diagrams. This approach allows the use of almost all other UML modeling capabilities. This approach is achieved by defining a new generalization element used to extend the semantic capabilities of any classifier (defined in the UML Kernel package). The generalization element is referred to as *OperationalizableElement*. This metaclass allows the incorporation of measurements representing operational information, as shown in Figure 24.

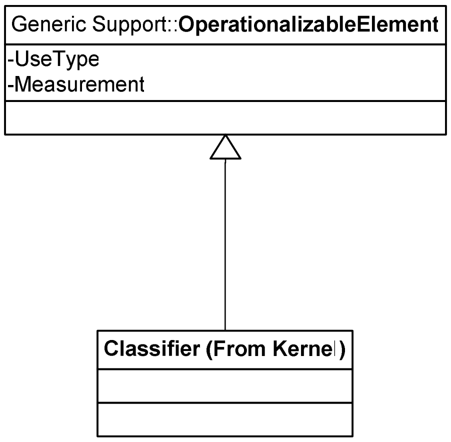


Figure 24. Partial Meta-Model Showing the Incorporation of Operational Attributes.

The consequence of this extension is that almost any UML model may be "operationalizable." This means that any UML diagram can be "semantically enriched" with measurements about the expected or intended use of the entity described in the diagram. The consistency and coherence of any UML operationalized model depends on the specific case.

The UML Operational Characterization Profile is organized in two logical groups of concepts that are stored in three different packages: Package for data-related information (i.e., Figure 26), package for functional information (i.e., Figure 27), and a package providing generic support (i.e., Figure 25).

Figure 25 shows the package containing modeling constructs that are general to the other packages. The descriptions of concepts defined within this package are:

- An *Environmental Constraint* is a condition about the parameters defining the system environment (i.e., the conditions about the system execution).

- The *Measurement Consistency Model* represents a set of rules defined over measurement attributes defined within a model for some modeling constructs. The consistency rules are defined with *Formation Rules* and *Invariants*.
- An *Operational Profile* is a quantitative specification about the expected use of a system. The specification can be defined at different levels: operation level, data level, or a combination of both levels.
- The *OperationalizableElement* is used to extend the modeling capabilities for the measurement specification. The attributes defined in this class, which are extended to all classifier modeling constructs, allow the specification of constraints and invariants over any model that uses this feature.
- A *Scenario* is a typical interaction between the user and the system or between two software components. A scenario can be described as a set of scenarios following a structured organization such as a state chart.

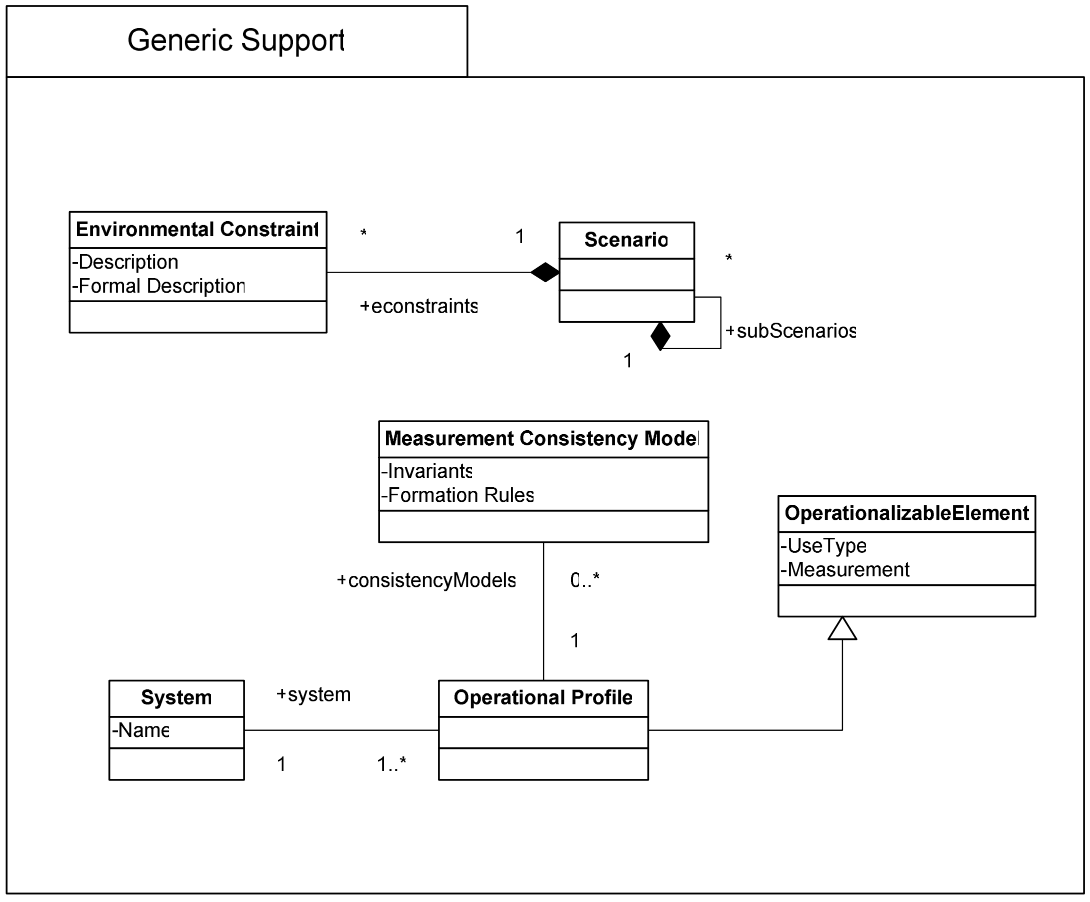


Figure 25. Generic Support Package.

Figure 26 shows the package containing modeling constructs that are data-related. These construct are related to those defined within other packages. The descriptions of concepts defined within this package are:

- A *Data Constraint* represents a condition that defines a subset over the entire domain associated with specific data.
- A *Domain* is a set of all possible values that any particular program might require or generate.
- A *Subdomain* is a subset of values from any specific domain. Usually, a set of subdomains forms a partition over a specific domain.
- An *Input Domain* is the domain containing all the input values defined for a specific function, operation, or program.
- An *Output Domain* is the domain containing all the output values defined for a specific function, operation, or program.

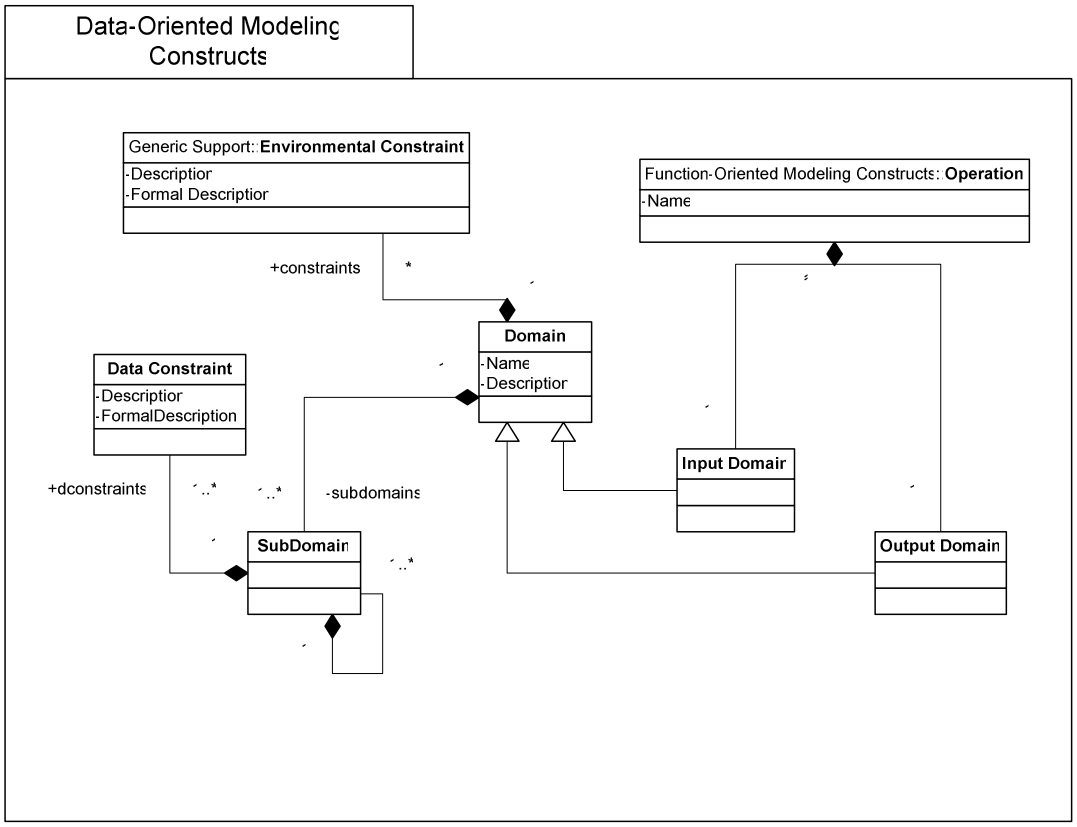


Figure 26. Data-oriented Modeling Constructs Package.

Figure 27 shows the package containing modeling constructs that are function-related. These constructs are related to those defined within other packages. The descriptions of concepts defined within this package are:

- An *Operational Mode* is a set of major logical operations. Usually there is a small number (e.g., from 2 to 8) of operational modes defined for a system.
- An *Operation* is a major system logical task performed by the system.
- An *Operation Group* is a set of associated operations based on predefined criteria.
- A *Function* is an atomic functionality performed by a system. The combination of one or more functions defines an operation.

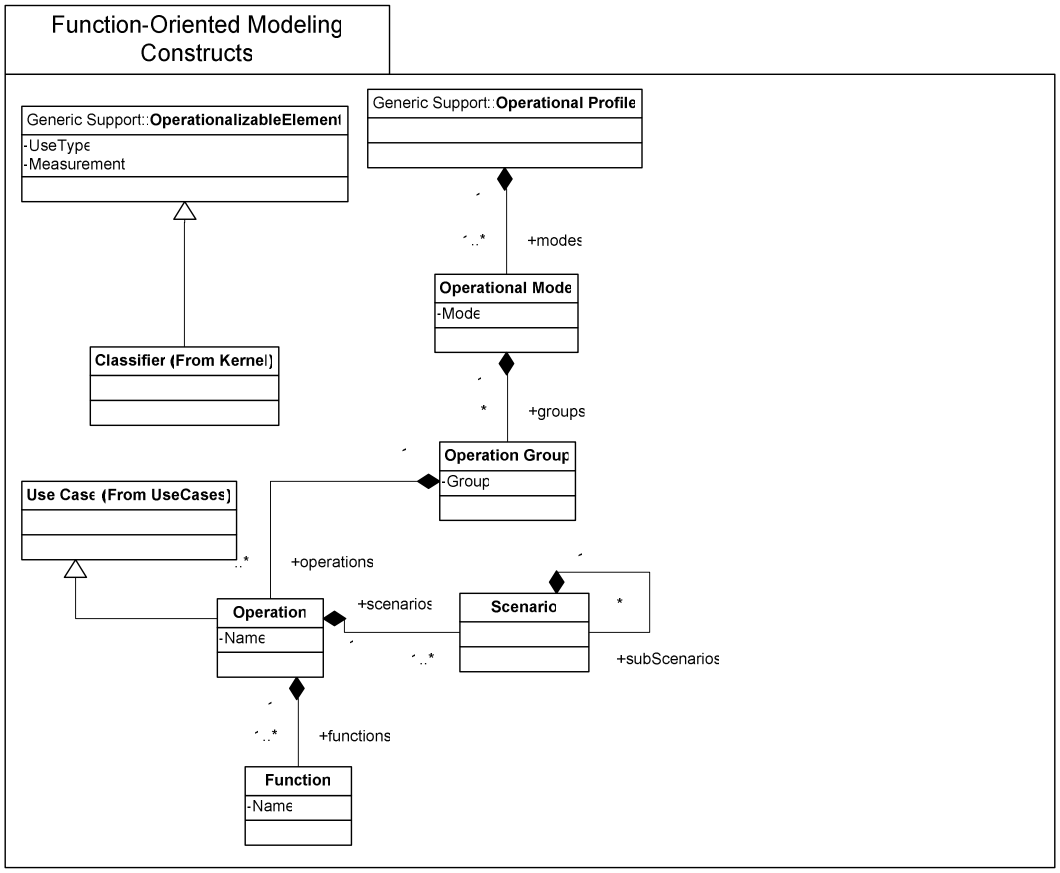


Figure 27. Function-oriented Modeling Constructs Package.

These modeling constructs provide support to specify operational models as discussed in Section 5.4. More technical details about the operational profile can be found in Appendix C.

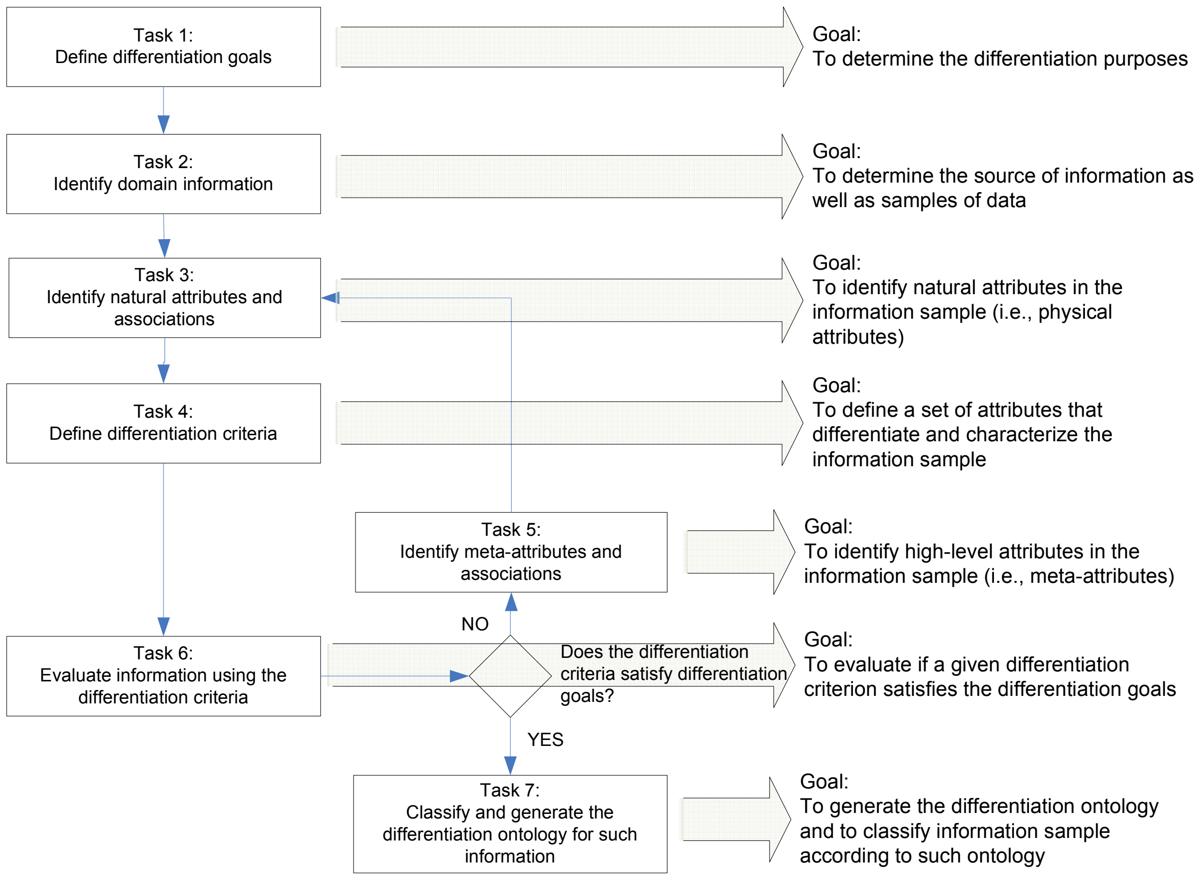
5.5.2 The Domain Characterization Method

The Domain Characterization Method (DCM) is a technique to characterize complex information into a suitable, observable, and differentiable form. This method is based on building differentiation ontology. This ontology is oriented to provide a set of attributes to differentiate data for a specific domain.

The strategy proposed to address such limitation is by defining meta-information about the data (e.g., meta-information about the image). This meta-information allows the characterization and differentiation among images. The required additional information depends on the specific application domain. For example, the meta-information required for an image-based application for face recognition is different from the information required for image-based application for volumetric change detection.

The strategy is based on creating an ontology associated with the application domain. The key part of this ontology is its orientation to differentiate data. This differentiation is helpful for operational modeling, data generation, and simulation. This ontology is referred to as the *differentiation ontology*.

An ontology is a specification of a conceptualization of a knowledge domain (Gomez-Perez et al., 2003). An ontology is a controlled vocabulary that describes objects and the relations between them in a formal way. It has a grammar for using the terms in the vocabulary and to express something meaningful within a specified domain of interest. Ontology development is a popular technique associated with knowledge management. There are several methodologies for building ontologies. The most representative methodologies are the Cyc Method (Lenat and Guha 1990), the Uschold and King's method (Uschold and Gruninger 1996), Grüninger and Fox's Methodology (Gruninger and Fox 1995), and Methontology (Fernandez-Lopez et al., 1997). The strategy proposed to address the problem in this section is inspired on the Methontology conceptualization activity. The result from this strategy is the Domain Characterization Method. DCM has seven steps with each one addressing a specific goal. Figure 28 shows the seven steps and their corresponding goal.



Task 1:
Define differentiation goals
Goal:
To determine the differentiation purposes
Task 2:
Identify domain information
Goal:
To determine the source of information as well as samples of data
Task 3:
Identify natural attributes and associations
Goal:
To identify natural attributes in the information sample (i.e., physical attributes)
Task 4:
Define differentiation criteria
Goal:
To define a set of attributes that differentiate and characterize the information sample
Task 5:
Identify meta-attributes and associations
Goal:
To identify high-level attributes in the information sample (i.e., meta-attributes)
NO
Task 6:
Evaluate information using the differentiation criteria
Does the differentiation criteria satisfy differentiation goals?
Goal:
To evaluate if a given differentiation criterion satisfies the differentiation goals
YES
Task 7:
Classify and generate the differentiation ontology for such information
Goal:
To generate the differentiation ontology and to classify information sample according to such ontology

Figure 28. Domain Characterization Method and the goals defined for every task.

The first step defines the goals of the ontology. These goals guide the refinement process (i.e., iterations) since they provide the stopping criteria. For example, taking as example the face recognition system, the goal of the ontology is the differentiation of input information in order to characterize three different scenarios where the system is used (i.e., operational information).

The second step collects the information that is manipulated and stored in the system. Using the same example, the information collected is a set of image files.

The third step analyzes the information to find the attributes associated with their physical representation. These attributes are referred to as natural attributes since they are naturally observed in such information. For example, the natural attributes for an image are the image width, height, size, and the values for every pixel within the image.

The fourth step defines the differentiation criteria by identifying the set of attributes used to differentiate all information items. Depending on the set of attributes found during the corresponding iteration, the criteria may be composed of natural as well as meta-attributes. Consider the example of the image during the first iteration. In this case, the only defined attributes are natural, therefore the differentiation criteria is defined using these attributes.

The fifth step is postponed when differentiation criteria do not satisfy the differentiation goals. This step creates a high-level structure of attributes. In the image example, meta-attributes may include attributes such as the number of people in the image, background type, and face position.

The sixth task evaluates the feasibility and the effectiveness of the differentiation criteria. Following the example, using any combination of natural attributes does not provide means to characterize scenarios, as required in the first step.

The iteration process continues until a suitable differentiation criterion based on a set of natural and meta-attributes is defined. Finally, the seventh step classifies the information and generates the additional information if those are not provided.

5.6 Discussion

This chapter has described two elements addressing the supporting claim: "*Operational information can be specified in a well-suited form that improves the reliability assessment.*" The first element, the notation, provides a standard notation for creating operational models of systems. The second element provides a technique to overcome the problem of modeling complex domains.

The operational modeling notation helps the reliability assessment framework since it provides a domain-specific notation. The use of such notation, however, is not a mandatory requirement. This notation helps since it may be implemented in some case tools allowing the use of UML profiles and performing analysis over these models.

The practical use of the proposed UML is limited to those kinds of systems that can be modeled using the proposed modeling constructs (i.e., data-oriented and function-oriented systems). The current version of this profile does not cover more complex specifications such as process-oriented systems.

The assembly operational profile is an extended operational profile since it includes additional information about the components and their connections within the assembly as well as the operational information defined for such assembly. This extension reduces the work required to manage additional artifacts (e.g., assembly specification).

The Domain Characterization Method yielded good results in the experiments performed. The main difficulty in using this technique is the determination of the differentiation criteria. Determining if the resulting characterization does really determine or influence the component behavior is out of the scope of this dissertation. However, the best theoretical characterizations are those that determine the best attributes which also have an impact on the software execution. Future research will address this issue.

5.7 Summary

This chapter has presented the evidence supporting the claim: "*Operational information can be specified in a well-suited form that improves the reliability assessment.*" The solution to the problems derived from this claim is instantiated in the second phase of the Test Profile Analysis Framework**.**

The solution includes the use of the assembly operational profile that is specified during the assembly definition process. The notation uses the modeling constructs defined in the UML Operational Characterization Profile.

The contributions derived from this chapter include:

- A notation, called the UML Operational Characterization Profile, to specify operational information.
- A basic technique to characterize complex domains where the physical stored information does not provide enough information to differentiate and manipulate distinct data items for testing purposes.

The next chapter will address the claim about the need for formal models to perform the execution trace composition and the execution trace prediction.

Chapter 6

Composing and Predicting Execution Traces

This chapter addresses the supporting claim: "*It is feasible to compose execution trace information from partial information of different components.*" The evidence to support this claim is provided by the claims: *"It is feasible to compose execution traces that represent the real composite trace"* and *"It is feasible to predict execution traces from a limited set of previously executed test cases"*. The evidences supporting these claims include two major issues: predicting execution traces based on limited information and defining a composition model based on such execution traces.

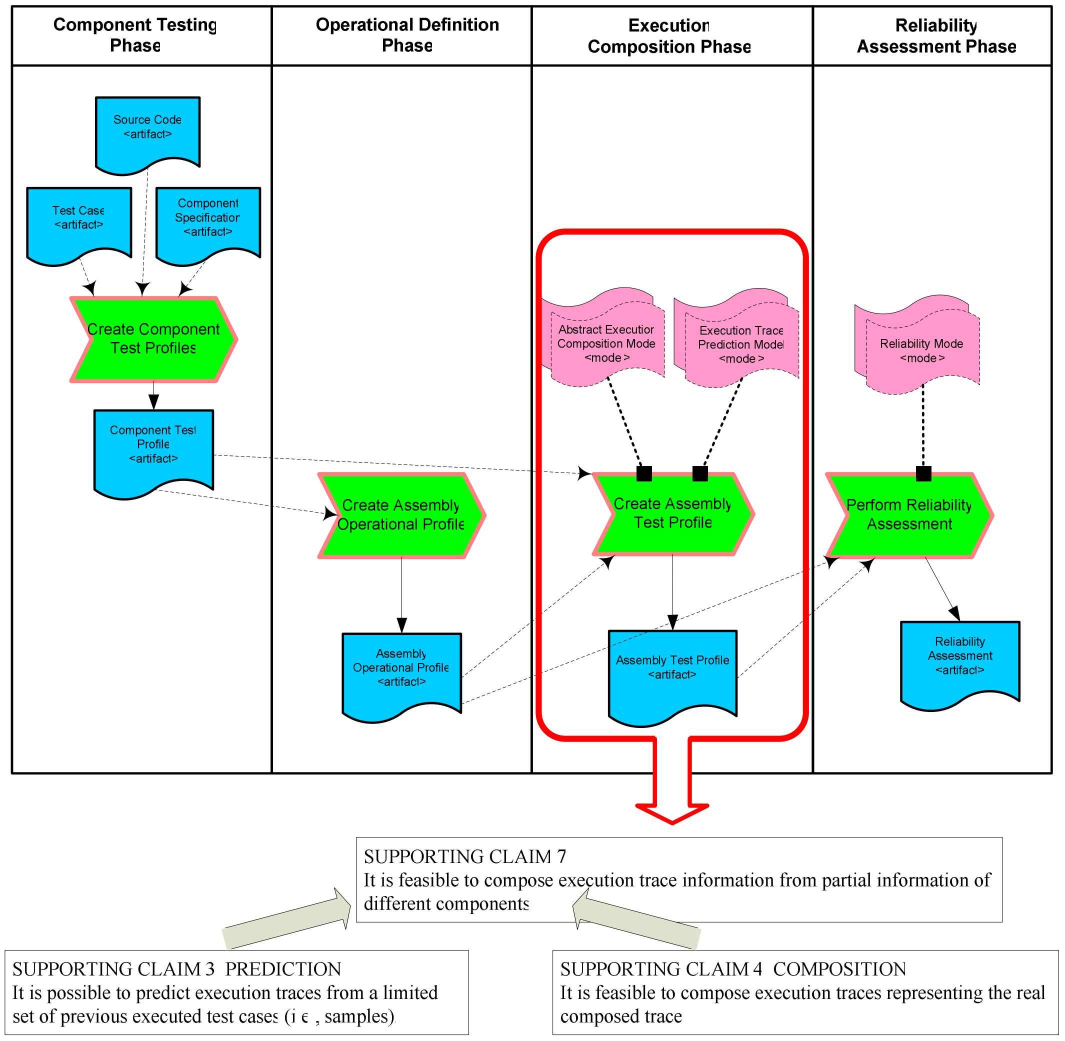
The Execution Composition Phase is the proposed solution to the issues mentioned above. Figure 29 shows the location of this phase in the framework. The Assembly Test Profile is the output artifact produced by this phase. This artifact is the practical solution supporting the claim addressed in this chapter.

The content of this chapter is structured as follows: First, a detailed description of the problem is presented. Second, the rationale for the solution is explained. Third, a description of the task, models, and artifacts used through the framework's phase are presented. Fourth, a description of the formal models supporting the phase is provided. Fifth, a set of examples is presented to illustrate the solution proposed for the problem addressed in this chapter. Sixth, the implementation of the solution is described. Finally, a discussion about the results, limitations, and future extension of the proposed solution is presented.

6.1 Problem Description

Two problems have to be solved to support the claim: the problem of having limited information and the problem of how to connect execution information from different component test profiles. These problems are important since their solution provides the foundation to consolidate the reliability-related information provided by component test profiles.

The problem of the composition of component execution traces into assembly execution traces is addressed by defining a composition model. There are several works that address the model-based composition. The models are based on different formalisms such as statecharts (Lüttgen et al., 2000), Petri-nets (Anisimov et al., 2001), process algebras (Wallnau et al., 2002b), Predicate Transition Nets and Temporal Logic (Yujian et al., 2006). The main goal of these formal models is the definition of a clear and unambiguous executable semantics. Examples of applications of formal semantics are model checkers (Gao et al., 2006), simulators (Narayanan and McIlraith 2002), and test case generators (Amyot et al., 2005).


Component Testing Phase
Operational Definition Phase
Execution Composition Phase
Reliability Assessment Phase
Source Code <artifact>
Test Case <artifact>
Component Specification <artifact>
Create Component Test Profiles
Component Test Profile <artifact>
Create Assembly Operational Profile
Assembly Operational Profile <artifact>
Abstract Execution Composition Model <model>
Execution Trace Prediction Model <model>
Create Assembly Test Profile
Assembly Test Profile <artifact>
Reliability Model <model>
Perform Reliability Assessment
Reliability Assessment <artifact>

SUPPORTING CLAIM 7
It is feasible to compose execution trace information from partial information of different components

SUPPORTING CLAIM 3 PREDICTION
It is possible to predict execution traces from a limited set of previous executed test cases (i.e., samples)

SUPPORTING CLAIM 4 COMPOSITION
It is feasible to compose execution traces representing the real composed trace

Figure 29. Framework Expanded View with Focus on the Third Phase.

The key part of any formalism resides in the basic information used to define the model. For example, some test case generators use a formal model based on extended state machines that represent message interactions between components. In this case, the basic information is the message interactions and the ordering defined between them.

In this research, the basic information is the execution-related information included in the component test profiles (i.e., execution traces). Then, the composition model uses such execution traces.

Intuitively, the composition problem is addressed by matching test cases. Figure 30 shows the composition process that matches test cases between the output value and the input value of two component test profiles. The ideal composition is achieved if the set of test cases in a component test profile covers the entire domain. However, the real component test profiles have a limited number of test cases (c.f., Chapter 4). This limited number of test cases might not have the corresponding test cases required for the composition process. This research addresses this issue by defining prediction models of component test profiles.

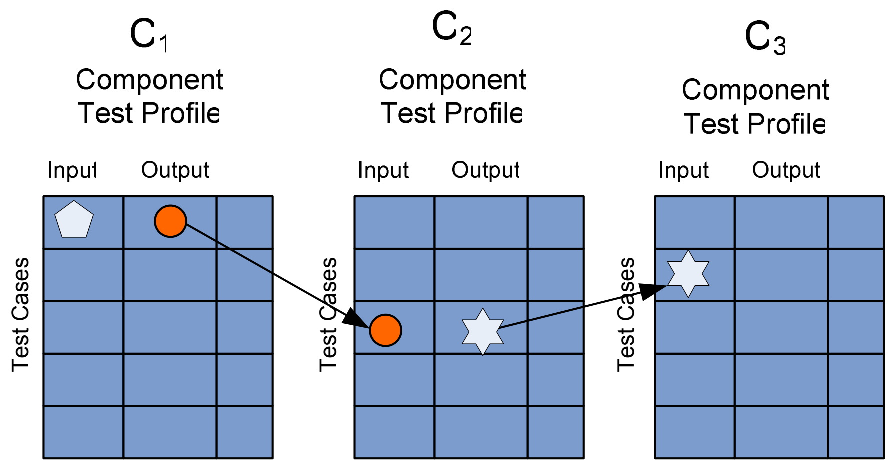


Figure 30. Component Test Profile Composition Process.

Prediction models are a popular issue addressed by several research areas such as statistics, engineering, chemistry, social sciences, and biology. Several works on prediction models have been developed for different scenarios. Some of these works are based on regression models, classification trees, Bayesian techniques, and artificial neural networks (Duda et al., 2000) among others.

The problems addressed in this chapter are the definition of an execution trace prediction model based on the information provided in the component test profile and the definition of a composition model which combines the individual component test profile information into an assembly test profile. The approach to solve these issues is described in the next section.

6.2 Solution Approach

The solutions proposed to address the problems described in Section 6.1 are described as follows: First, the proposed models to predict execution-related information are described. These models use different techniques such as artificial neural network, classification trees, and the best binary matching technique. Second, the composition model is presented. This composition model assumes the existence of the prediction model. Otherwise, a correct composition can not be guaranteed. A correct composition means that the composition result represents an acceptable approximation of the real composition.

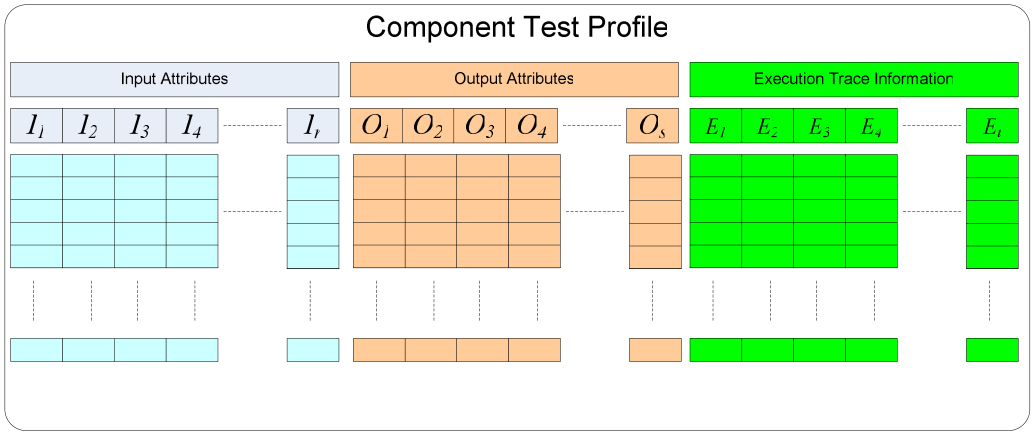
6.2.1 Predicting Execution Traces

The strategy to overcome the limited information problem is based on defining a prediction model. This model uses the inputs, outputs, and execution traces to train the prediction model.

The prediction model requires predicting execution node frequencies rather than predicting the entire execution trace. This characteristic comes from the information required by the reliability model (c.f., Chapter 7).

Another issue to consider is the characteristics of the data used for the prediction. The data is taken from the component testing profiles (c.f., Chapter 4) which is composed of input data, output data, and execution traces. The most simple and easy data to characterize are the execution node frequencies. The high heterogeneity that both input and output data may have requires a detailed analysis.

In general, as shown in Chapter 5, any domain is suitable for being characterized as a set of attributes. Using this characterization, any component test profile is organized as illustrated in Figure 31. This figure shows a diagram where the information about the test profile is represented as a set of tables having *r* input attributes, *s* output attributes, and *t* execution nodes defined for the component. The execution trace information stores the frequency found for a particular execution node for a specific execution trace. The frequency for the E_j execution node is denoted by $|E_j|$.



Component Test Profile
Input Attributes
Output Attributes
Execution Trace Information
I_1 I_2 I_3 I_4 I_r
O_1 O_2 O_3 O_4 O_s
E_1 E_2 E_3 E_4 E_t

Figure 31. Component Test Profile Information Required for Prediction.

The input and output attributes contain diverse data. This diversity is found in all different domain attributes. For example, it is possible to have one attribute with an Integer domain (i.e., 1 to 65535) or the case when the range is defined as a set of nominal values (i.e., gender attribute). This prediction model should account for this heterogeneity.

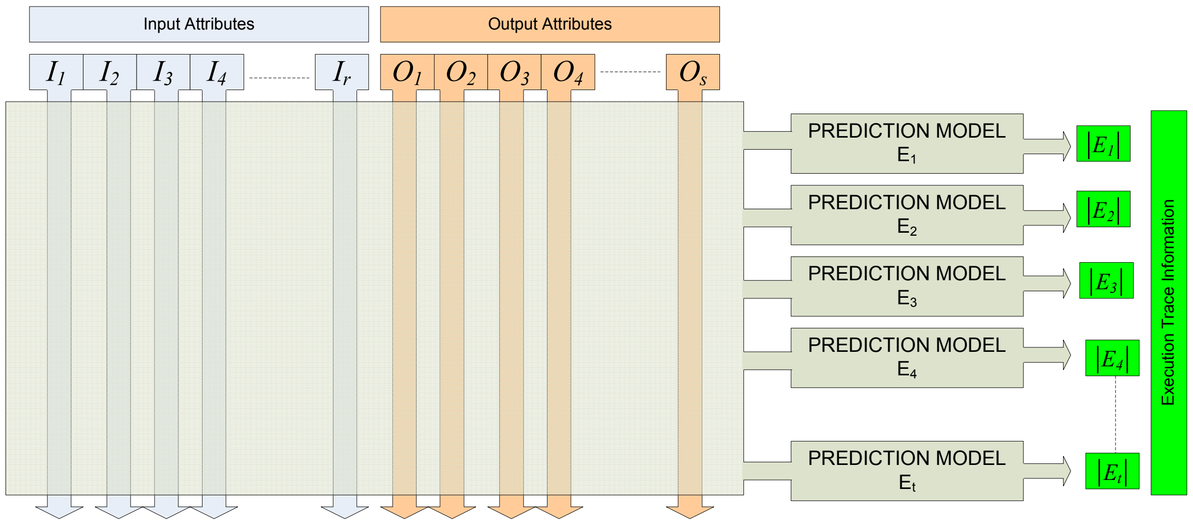


Figure 32. Execution Node Prediction Approach.

The attribute heterogeneity issue is addressed by selecting the best prediction model for each execution node frequency as shown in Figure 32. The prediction models considered in this research include artificial neural networks, classification trees, and a qualitative technique developed for cases where none of the two previous models provide good predictive results. This qualitatively technique is called the Best Binary Matching Technique.

The prediction goodness of artificial neural networks is determined by estimating the mean square prediction error using a validation set. This validation set is a sample of the original data set that is not used in the model training. Similarly, the accuracy of classification trees is assessed by estimating the error in the classification. The selection of the model, however, requires an analysis on a case by case basis since each model has configuration parameters modifying the model performance.

6.2.2 Composing Execution Traces

The execution composing problem uses the execution-related information (i.e., execution traces). Intuitively, the assembly execution trace is derived from the execution traces from the components in the assembly. For a simple case, where two components are connected sequentially, the resulting assembly execution trace is defined as the concatenation of the execution traces of those two components. The key issue in the composition is the characterization of conditions allowing this concatenation.

The proposed formal composition model uses a matching mechanism to characterize the composition of two test cases. The matching mechanism is defined as a predicate over the output domain and the input domain of two component test profiles. In other words, the composition of two test cases is allowed when the predicate over these test cases is true.

6.3 Solution Description

This section describes how the ideas in the previous section are implemented in the framework. First, a description of the artifacts is provided. Next, a description of the task required for the assembly test profile construction is explained.

6.3.1 Artifacts Description

The input artifacts required in this phase are a set of component test profiles and the assembly operational profile. The information required from the assembly operational profile is the specification of which components are required for the assembly. The output artifact is the Assembly Test Profile.

The *Assembly Test Profile* artifact contains the results of the composition of the component test profiles. This artifact is similar to the component test profile but it includes additional information concerning the composition process and the uncertainty of prediction models.

A template for every artifact is provided in Appendix B.

6.3.2 Task: Create Composite Test Profile

The *Create Assembly Test Profile* task is performed during the third phase of the framework. The purpose of this task is the generation of the assembly test profile (i.e., composite test profile) required for the reliability assessment. The assembly test profile can be seen as a single component.

The activities involved in this task are the selection of the best prediction model for every component test profile and the composition process supported by the prediction models. The first activity analyses the uncertainty of different prediction techniques in selecting the most appropriate prediction model for each component test profile. The second activity defines the matching criteria for performing the composition process.

6.4 Formal Models

This section provides a description of the formal models required for composing and predicting execution traces. Since prediction models are based on different techniques, this section only outlines the most important characteristics of such kind of techniques. This research proposes the Best Binary Matching technique for cases where the other techniques do not provide good results. The composition model is described by formalizing several items that are used in the test profiles. The key part of this formalization is defined by the composition rule which defines the matching criteria.

6.4.1 Execution Trace Prediction Model

The techniques proposed in this work to address the prediction problem are based on artificial neural networks, classification trees, and a qualitative model named as the Best Binary Matching Technique. This section provides an overview of such techniques.

Artificial Neural Networks
An artificial neural network (ANN) or commonly just neural network (NN) is an interconnected group of artificial neurons (or nodes) that uses a mathematical model for information processing based on a connectionist approach to computation. A neural network has an input layer, processing layers and an output layer (Duda et al., 2000). Figure 33 shows a typical representation of an artificial neural network.

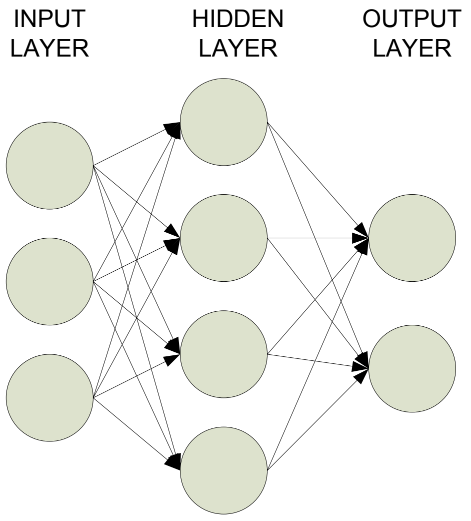


Figure 33. Representation of an Artificial Neural Network.

The information in a neural network is distributed throughout the processing layers. Each processing layer is made up of many nodes. Each node simulates a neuron by its interconnection to other nodes. The arrangement of outputs forms the final output from the neural network. The neural network is taught by correcting the false or undesired outputs from a given input. Training results in a different set of nodes being used and in an adjustment in the weights of the interconnections to other nodes.

The weights and adjustments are similar to a least mean squares fit of a line to a number of points. Just as statistical analysis reveals underlying patterns in a collection of data, a neural network locates consistent patterns in a collection of data, based on a set of predefined criteria.

The input layer is connected to the raw data. The middle (or hidden) layer is used to consolidate the results of the input layer, and the output layer provides the results of the consensus propagated from the previous layers. In practice, there are usually many more nodes in the input layer than in the middle layer, and the number of nodes in the output layer usually corresponds to the number of possible outcomes.

Detailed information about ANN can be found in (Duda et al., 2000).

Classification Trees
Classification trees (CT) have been used widely in operation research for decision analysis. However, from other perspectives, a classification tree is a prediction model that has been used for data classification or data mining. More descriptive names for such tree models are decision tree or reduction tree. In these tree structures, leaves represent classifications and branches represent conjunctions of features that lead to those classifications (Duda et al., 2000).

Classification trees are popular mechanisms since they are simple to understand and interpret. Additionally, classification trees use a white box model. If a given result is provided by a model, the explanation for the result is easily replicated by simple mathematics. Similarly to the ANN, this kind of models does not have any assumptions about the input data.

A classification tree is built through a process known as binary recursive partitioning. This is an iterative process of splitting the data into partitions, and then splitting it up further on each of the branches.

Figure 34 shows an example of a classification tree (XLMiner). The procedure to build such tree starts with the data or the training set. The classification label (e.g., "purchaser" or "non-purchaser") is known and it is pre-classified for every record in the training set. All of the records in the training set are grouped together as a table. The algorithm then systematically tries breaking up the records into two parts, examining one variable at a time and splitting the records on the basis of a dividing line in that variable (e.g., income > $75,000 or income <= $75,000). The goal of this process is to attain a homogeneous set of labels (e.g., "purchaser" or "non-purchaser") in each partition. This splitting or partitioning is then applied to each of the new partitions. The process continues until no more useful splits can be found. The heart of the algorithm is the rule that determines the initial split rule.

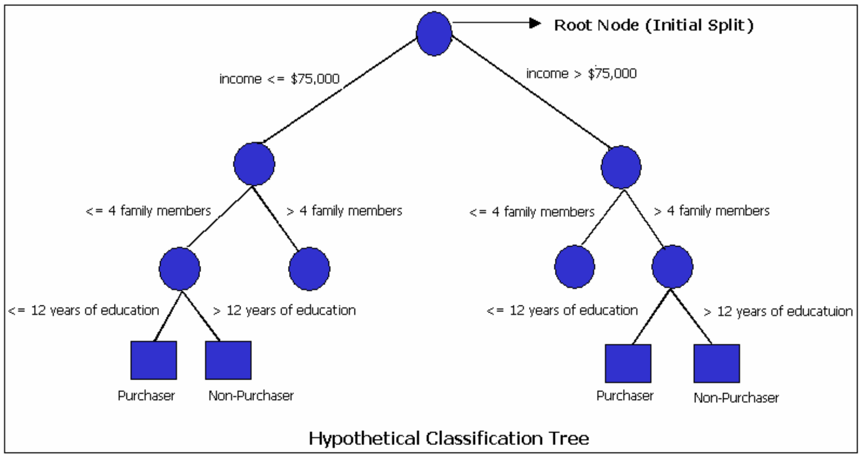


Figure 34. Example of a Classification Tree.

Detailed information about CT can be found in (XLMiner 2007) and (Duda et al., 2000).

Best Binary Matching Technique
The Best Binary Matching Technique (BBMT) is proposed for cases when ANN and CT do not provide good results. This technique finds the most similar point in a given set of points. This technique creates a binary characterization of such points and then it determines the most similar point by computing the minimum distance between those points.

This technique works better on cases where there is a considerable amount of categorical attributes defined for every point. Numerical attributes are grouped into discrete groups. For example, Table 17 shows a data set of points characterized by two categorical attributes and one numerical attribute.

Attribute			Output
Age	Gender	Country	Numerical
33	Male	Mexico	4
25	Female	USA	3
12	Male	Korea	8
.	.	.	.
.	.	.	.
.	.	.	.

Table 17. Data Set Example for BBMT.

The BBMT requires transforming the information in the data set into a binary representation. This transformation generates new columns for every categorical value of every categorical attribute. Numerical attributes are transformed into categorical values by defining classes[4]. The resulting binary representation of the information shown in Table 17 is presented in Table 18. In this case, the Age attribute was transformed into seven classes.

Attribute											Output
Age G1	Age G2	Age G3	…..	Age G7	Gender Male	Gender Female	Country Mexico	Country USA	Country Korea	….	Numerical
0	0	1	.	0	1	0	1	0	0	…	4
0	1	0	.	0	0	1	0	1	0	…	3
1	0	0	.	0	1	0	0	0	1	…	8
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.

Table 18. Binary Representation of the Data shown in Table 17.

Then, the prediction looks for the most similar point that matches the corresponding input point. The criterion for selecting the most similar point is based on finding the minimum number of unmatched values (or bits). This similarity criterion may represent a trivial

[4] The number of groups follows a pragmatic justification since it is easy to manage smaller data sets.

solution. However, the criterion could be improved by performing more complex analysis to find the most significant attributes and, then, use those attribute only.

We include an experimental metric to measure the relative error found in the point selection process[5]. This measure is based on the number of bits that are different between the matched points (i.e., the distance between those points). Intuitively, the smaller the distance the higher the confidence in the value used for such input. We call this distance as the *non-normalized selection error*.

The *non-normalized selection* error for two test points tp_j and tp_k from different test profiles is defined as

$$\delta\,(tp_j \bullet tp_k) = distance(tp_j.o \text{ XOR } tp_k.i)$$

where $tp_j.o$ denotes the binary value for the output for the *j-th* test point and $tp_k.i$ denotes the input binary value for the *k-th* test point, and distance is defined as the number of bits having value 1. These values are produced by the XOR operation.

This metric could be used as a measure of the cumulative computed selection error for more than two sequentially-composed components. This metric is biased by the range of values for each sub-domain. One way to overcome this situation is by normalizing the values of this metric to values referenced to the maximum value found.
Let D denote the set of all distance values produced by the best matching criterion between two test profiles. The distance represents the number of different bits. The normalized selection error for any distance $\delta_k \in D$ is defined as

$$\bar{\delta}_k = \frac{\delta_k}{max\{\delta_j | \forall j \in \mathbf{D}\}}$$

Next, this metric needs to be extended to measure the cumulative error produced in the test point selection among several test profiles. We define the normalized cumulative selection error, $\bar{\Delta}$, as the average of all normalized selection errors found in a composite test point:

$$\bar{\Delta}(tp_1 \bullet tp_2 \bullet tp_3 \bullet \ldots \bullet t_r) = \frac{1}{r-1} \sum_{i=2}^{r} \bar{\delta}(tp_{i-1}.o, tp_i.i)$$

Then, the *i-th* subdomain cumulative error selection is the average error computed from those composite test points located in the *i-th* subdomain. The sub-domain cumulative error selection is defined as

$$\bar{\Delta}(s_i) = \frac{1}{n_i} \sum_{m \in s_i} \bar{\Delta}(m)$$

where s_i denotes the *i-th* subdomain and m denotes a test point in s_i.

[5] We use the term "experimental" since our research is still looking for the statistical significance of such measure. The current use of such measure is just for descriptive purposes.

The domain cumulative error in selection can be estimated using a weighted sum of the corresponding subdomain cumulative errors and the subdomain operational probabilities.

6.4.2 Abstract Execution Composition Model

The abstract execution model is defined formally by abstracting any system using the information contained in its test profile. This section uses the term system, but it can be interchanged with the term component without any problem.

The strategy followed to describe the composition model is by defining the basic building block used for the formalization. Then, a set of composition rules are stated for the composition process.

The basic building blocks defined for this model are:

- **Execution Nodes**. A finite set E of labeled execution nodes.
- **Basic Data Domains**: A set B_D of basic data domains. These domains represent basic data types such as

$$B_D = \{Integer, Real, Character, \textbf{SEQ}(Character)\}^{6}.$$

The following definitions provide the characterization of the data contained within test profiles.

<u>Definition 6.1.</u> A **Constrained Basic Data Domain**, R_D, is a subset of a basic data type defined in the set of basic data domains. This subset is defined as

$$R_D = \{x \mid \textbf{\textit{istypeof}}(x) \in B_D \wedge \textbf{\textit{pred}}(x)=\textbf{TRUE})\}$$

where x represents any element in the subset satisfying two conditions. The first condition states that the data type associated with such element is contained in the set of basic data domains (i.e., using the ***istypeof*** predicate). The second condition focuses on the specific constraints associated with each subset. These constraints are defined using the ***pred*** predicate.

<u>Definition 6.2.</u> A **Complex Data Domain**, C_D, is a tuple of constrained basic data domains such as

$$C_D = <rd_1, rd_2, rd_3,\ldots,rd_n>$$

where $rd_1, rd_2, \ldots rd_n$ denotes constrained basic data domains.

[6] The expression, **SEQ***(Character),* denotes all possible sequences of characters. In the computer jargon this is known as the String data type.

<u>Definition 6.3.</u> An **Input Domain**, $\mathcal{I}$, is a tuple of complex data domain such as

$$\mathcal{I} = <cd_1, cd_2, cd_3, \ldots, cd_r>$$

where $cd_1, cd_2, \ldots cd_r$ denotes complex data domains.
Similarly, an **Output Domain**, $\mathcal{O}$, is a tuple of complex data domain such as

$$\mathcal{O} = <cd_1, cd_2, cd_3, \ldots, cd_s>$$

where $cd_1, cd_2, \ldots cd_s$ denotes complex data domains.

<u>Definition 6.4.</u> An **Input Domain Instance**, I, is a tuple such as

$$I = <i_1, i_2, i_3, \ldots, i_r>$$

where $i_1 \in cd_1, i_2 \in cd_2, i_3 \in cd_3, \ldots i_r \in cd_r$. This tuple represents a set of actual values (or instances) defined for every constrained basic data domain. The definition for the Output Domain Instance, O, follows a similar definition. An **Output Domain Instance**, O, is a tuple such as

$$O = <o_1, o_2, o_3, \ldots, o_s>$$

where $o_1 \in cd_1, o_2 \in cd_2, o_3 \in cd_3, \ldots o_s \in cd_s$.

<u>Definition 6.5</u>. A Test Profile, T_P, is a tuple such as

$$T_P = <\mathcal{P}, \mathcal{I}, \mathcal{O}, E>$$

where $\mathcal{I}$ denotes the input domain and $\mathcal{O}$ the output domain. E denotes a set of labeled execution nodes and $\mathcal{P}$ denotes a set of tuples such as

$$\mathcal{P} = <I, O, \mathcal{E}>$$

where I is an input domain instance, O is an output domain instance, and $\mathcal{E}$ represents a sequence of labeled execution nodes where $\mathcal{E} \in \mathbf{SEQ}(E)$.

Next, additional definitions are provided to characterize the matching of information between instances of input and output domains.

<u>Definition 6.6</u>. Domain Matching Condition. Given an input domain $\mathcal{I}$ and an output domain $\mathcal{O}$, it is said that these domains satisfy a **domain matching condition,** denoted by $\mathcal{I} \Rrightarrow \mathcal{O}$**,** when for every complex data domain, cd_i in $\mathcal{I}$, there is an equivalent complex data domain cd_j in $\mathcal{O}$. Two complex data domains are equivalent when $istypeof(cd_i) = istypeof(cd_j)$.

<u>Definition 6.7</u>. Instance Equivalence Condition. Given an input domain instance I and an output domain instance O such that $\mathcal{I} \Rrightarrow \mathcal{O}$, it is said that these instances satisfy an **instance equivalence condition,** denoted by $I \approx O$**,** if for every instance i_k and its corresponding match o_j, the predicate $covered(i_k, o_j)$ = TRUE.

The predicate covered (i_k, o_j) is defined as a function that returns true if i_k is mapped to o_j and returns false otherwise. The predicate *covered* plays an important role in the composition since it determines which instance can be used for the composition.

The composition mechanism is defined as follows. We use the notation $T_i(\mathcal{I})$ and $T_i(\mathcal{O})$ to denote the corresponding input and output domains of the T_i test profile.

<u>Definition 6.8</u>. Execution Trace Composition. Given two execution node sequences, e_1 and e_2, the composition of sequences, denoted by $e_1 \oplus e_2$, is defined as the concatenation of both execution sequences. Formally, the composition of two execution traces, such as $e_1 = a_1a_2a_3...a_n$ and $e_2 = b_1b_2b_3...b_m$, is defined as $e_1 \oplus e_2 = a_1a_2a_3...a_nb_1b_2b_3...b_m$.
The properties of this execution trace composition operator are

- Antisymmetric, $e_1 \oplus e_2 \neq e_2 \oplus e_1$
- Identity, $e_1 \oplus \varepsilon = \varepsilon \oplus e_1 = e_1$, where ε is an empty trace.

The next rule is defined following an algorithmically oriented approach.

<u>Test Profile Composition Rule</u>. Given two test profiles,

$$T_j = <\mathcal{P}_j, \mathcal{I}_j, \mathcal{O}_j, E_j>, \quad T_k = <\mathcal{P}_k, \mathcal{I}_k, \mathcal{O}_k, E_k>,$$

where the output domain of the j^{th}-profile and the input domain of the k^{th}-profile satisfy the domain matching condition $T_j(\mathcal{O}) \Rrightarrow T_k(\mathcal{I})$. Then, the executable composition of two test profiles, denoted by $T_j \rightarrow T_k$, is defined as follows:

- A new test profile, T_{jk}, is defined. The elements of this test profile are instantiated as follows:

  $$T_{jk} = <\mathcal{P}_{jk}, \mathcal{I}_j, \mathcal{O}_k, E_{jk}>$$

  where

  $E_{jk} = E_j \cup E_k$, representing the union of both labeled execution node sets.

  $\mathcal{P}_{jk}$, representing the composite tuples $\mathcal{P}_{jk} = <I_c, O_c, \mathcal{E}_c>$ from both test profiles. This set of tuples is generated following the next algorithm:

  - For every $p \in \mathcal{P}_j$, find a tuple $q \in \mathcal{P}_k$ satisfying the instance equivalence condition, such that $p.O \approx q.I$.
  - Add a new tuple $<i_c, o_c, e_c>$ in $\mathcal{P}_{jk}$ such that

    $i_c = p.i$

    $o_c = q.o$

    $e_c = p.e \oplus q.e$

    where $p.i$ denotes the input domain instance in p, $q.o$ denotes the output domain instance in q, and $p.e$ and $q.e$ denote the corresponding execution trace.

6.5 Example

This section presents examples to illustrate how the prediction and composition models are applied. This section uses the mathematical computation assembly defined in previous chapters. A more elaborate example is presented in Chapter 8.

6.5.1 Prediction Models for the Square-Root Computation Component

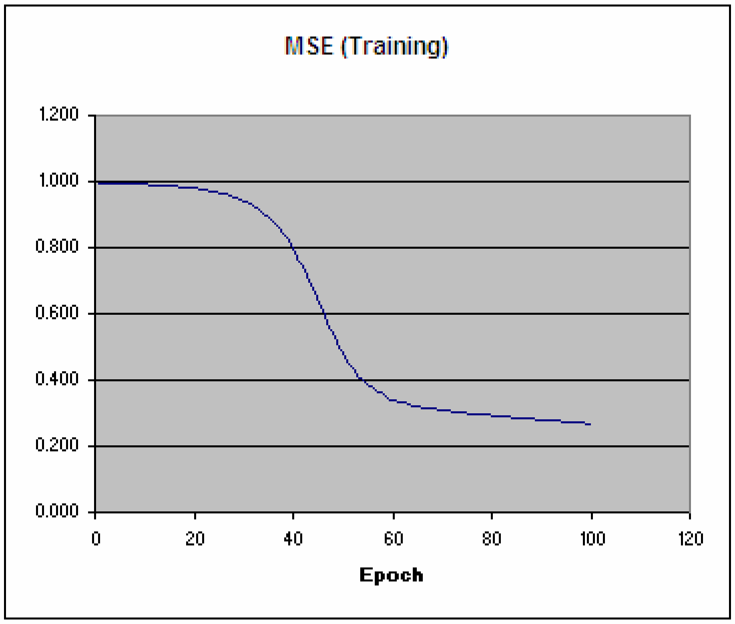
The data provided in the square-root computation component test profile is shown in Table 19.

INPUT	OUTPUT	|A|	|B|	|C|	|D|
60662	246.2965698	1	13	1	1
55440	235.4570007	1	13	1	1
48820	220.9524841	1	12	1	1
22426	149.7531281	1	12	1	1
42661	206.5453949	1	12	1	1
64047	253.0750885	1	13	1	1
10035	100.1748505	1	11	1	1
30172	173.7008972	1	12	1	1
23309	152.6728516	1	12	1	1
58607	242.0888214	1	13	1	1
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Table 19. Data Required for the Square-Root Computation Component Prediction Model.

In this case, the component test profile requires only one prediction model since execution node B is the only node that shows variability. The other nodes keep a constant value through all test cases.

Artificial Neural Networks provides a good prediction model due to the characteristics of the data defined for this component. This component has only one numerical attribute as input and there were a considerable number of test points for the ANN training. Figure 35 shows the Mean Squared Error (MSE) found during the training and validation of the ANN defined for the B execution node. In this case the model was trained using a random sample containing 600 test cases. Two hidden layers were specified and 100 training cycles (i.e., epochs) were used. The errors shown in this model do not have a significant impact on the prediction since the mean square error is less than 1 occurrence. This prediction error is usually incorporated into the assembly test profile.



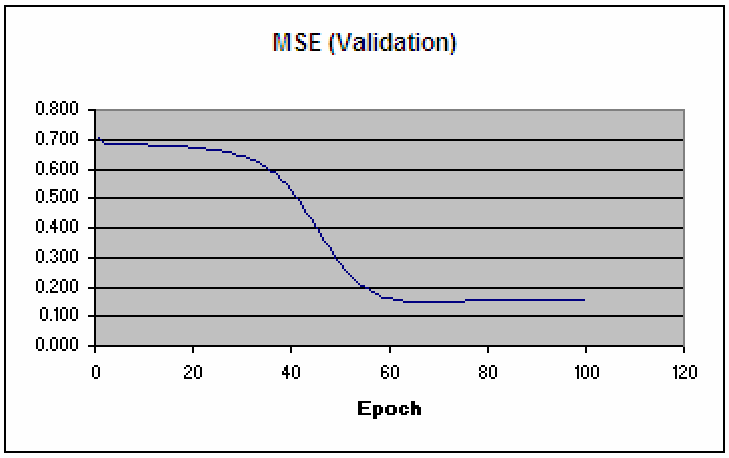


Figure 35. MSE Found in ANN Training and Validation Phases.

6.5.2 Mathematical Computation Assembly Composition

This example uses the basic example describing the assembly composed of the square root and logarithm computation components.

We use two component test profiles to illustrate the concepts defined previously. The assembly in the example has two components; one computes the square root of integer values and the second computes the logarithms of real values, denoted as C_{sqr} and C_{log} respectively. This assembly is described in Section 3.5 and the square root computation component test profile is described in Section 4.4.1.

The component test profile for the C_{sqr} component is defined as follows

$$
\begin{aligned}
&TP_{sqr} = <\mathcal{P}_{sqr}, \mathcal{I}_{sqr}, \mathcal{O}_{sqr}, E_{sqr}>, \text{ where}\\
&\quad E_{sqr} = \{\text{A, B, C, D}\}\\
&\quad \mathcal{I}_{sqr} = << i_{sqr} \in \text{Integer} >>\\
&\quad \mathcal{O}_{sqr} = << o_{sqr} \in \text{Real} \wedge [o_{sqr} > 0 \text{ and } o_{sqr} < 300] >>\\
&\quad \mathcal{P}_{sqr} = \{<1,1,\text{ABBBCD}>, <16, 4, \text{ABBBBCD}>, \ldots.\}
\end{aligned}
$$

Similarly, the component test profile for the C_{log} component is defined as follows

$$
\begin{aligned}
&TP_{log} = <\mathcal{P}_{log}, \mathcal{I}_{log}, \mathcal{O}_{log}, E_{log}>, \text{ where}\\
&\quad E_{log} = \{\text{H, I, J, K}\}\\
&\quad \mathcal{I}_{log} = << i_{log} \in \text{Real} >>\\
&\quad \mathcal{O}_{log} = << o_{log} \in \text{Real} >>\\
&\quad \mathcal{P}_{log} = \{<1,0, \text{HIJK}>, <3.5, 0.544, \text{HIJJJK}>, \ldots.\}
\end{aligned}
$$

The composition requires meeting the *domain matching condition*, $\mathcal{I}_{log} \Rrightarrow \mathcal{O}_{sqr}$. This condition is satisfied since *istypeof*(Real) = *istypeof*(Real). Next, we need to state the *instance equivalence condition* by defining the predicate *covered.* In this case, the predicate *covered* is defined as follows

$$covered(i, j) = \begin{cases} \text{TRUE} & |i - j| < \xi \\ \text{FALSE} & Otherwise \end{cases}$$

This predicates is simple since it only compares the numeric distance between two elements (i.e., the value selected is the one that is less than a threshold value denoted as ξ).

Finally, using the previous information, the composite test profile is described as

$TP_{sqr\text{-}log} = <\mathcal{P}_{sqr\text{-}log}, \mathcal{I}_{sqr\text{-}log}, \mathcal{O}_{sqr\text{-}log}, E_{sqr\text{-}log}>$, where

$E_{sqr\text{-}log} = \{A, B, C, D, H, I, J, K\}$

$\mathcal{I}_{sqr\text{-}log} = << i_{sqr\text{-}log} \in \text{Integer} >>$

$\mathcal{O}_{sqr\text{-}log} = << o_{sqr\text{-}log} \in \text{Real} >>$

$\mathcal{P}_{sqr\text{-}log} = \{<1{,}0, \text{ABBBCDHIJK}>, \; \ldots.\}$

In practice, the instance meeting the equivalence condition is substituted by using the predicted test case or test point.

6.6 Tool Support and Implementation Issues

The tool performing the prediction and composition is implemented as a set of scripts in the Microsoft Access System (MS Access). MS Access (Friedrichsen 2003) is a relational database management system developed by Microsoft. This system combines the relational Microsoft Jet Database Engine with a graphical user interface. Additionally, Microsoft Access can use data stored in Access/Jet, Microsoft SQL Server (Nicol and Albrecht 2002), Oracle, or any ODBC-compliant data container (Geiger 1995). One advantage of MS Access is that professional applications can be built in a relatively short time.

The powerful statistical environment R is used to implement the prediction models. R is a language and environment for statistical computing and graphics (Maindonald and Braun 2006) (CRAN 2006). It is a GNU project which is similar to the S language and environment. R provides a wide variety of statistical packages (e.g., linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, and clustering, among several others) and graphical techniques.

6.7 Discussion

This chapter has described elements addressing the claims *"It is feasible to compose execution traces that represent the real composite trace"* and *"It is feasible to predict execution traces from a limited set of previously executed test cases"*. Such elements include a composition model for the first claim and a prediction model for the second.

The proposed composition model is based on abstracting any software component using its inputs and outputs. This approach is suitable for a wide range of systems, but there are cases where highly interactive systems require a more sophisticated composition model, including concurrency and parallelism support mechanisms such as Petri-Nets or Process Algebras.

The proposed approach to address the prediction issue relies on the nature of the information contained in the component test profile. This issue should be addressed on a case by case basis until we can incorporate more assumptions in the data, such as randomness and a specific degree of source code coverage provided by techniques such as code coverage analysis. Code coverage analysis is the process of finding areas of a program not exercised by a set of test cases, creating additional test cases to increase coverage, and determining a quantitative measure of code coverage (Fenton and Pfleeger 1998) (Jorgensen 1995).

Our research uses the same prediction model in each component. The prediction model, however, may be improved by generating an independent prediction model for every execution node instead of using the same technique for all of them. One issue to consider when moving in this direction is the computational cost required for the selection of the best prediction model.

The integration of both models (i.e., prediction and composition models) proposed in this dissertation is not completely transparent, since the composition model assumes the existence of the matching condition that is implemented in the framework using a prediction value. An improvement in such model is the redefinition of the matching condition using composition operators for stochastic information. This improvement requires the definition of the stochastic information in terms of the prediction models required for the composition.

6.8 Summary

This chapter has presented the evidence supporting the claim: "*It is feasible to compose execution trace information from partial information of different components.*" The evidence supporting this claim is given by the evidence supporting the claims: *"It is feasible to compose execution traces that represent the real composite trace"* and *"It is feasible to predict execution traces from a limited set of previously executed test cases"* The third phase of the Test Profile Analysis Framework contains the solutions to the problems derived from this claim**.**

The solution includes the use of a composition model to characterize the execution-related information. Additionally, the solution illustrates how prediction models can be defined based on component test profile information.

The contributions of this chapter include:

- An execution composition model based on a data matching abstraction. This model composes the information of different component test profiles assembled sequentially.
- An approach for defining prediction models for execution node frequencies. This approach uses techniques such as artificial neural networks and classification trees.
- A prediction technique, named as Best Binary Matching Technique, designed for cases when statistical prediction models do not provide good prediction results.

The next chapter will address the claim about the need for formal models to perform the reliability assessment of assembly test profiles.

Chapter 7

Prediction Reliability Model

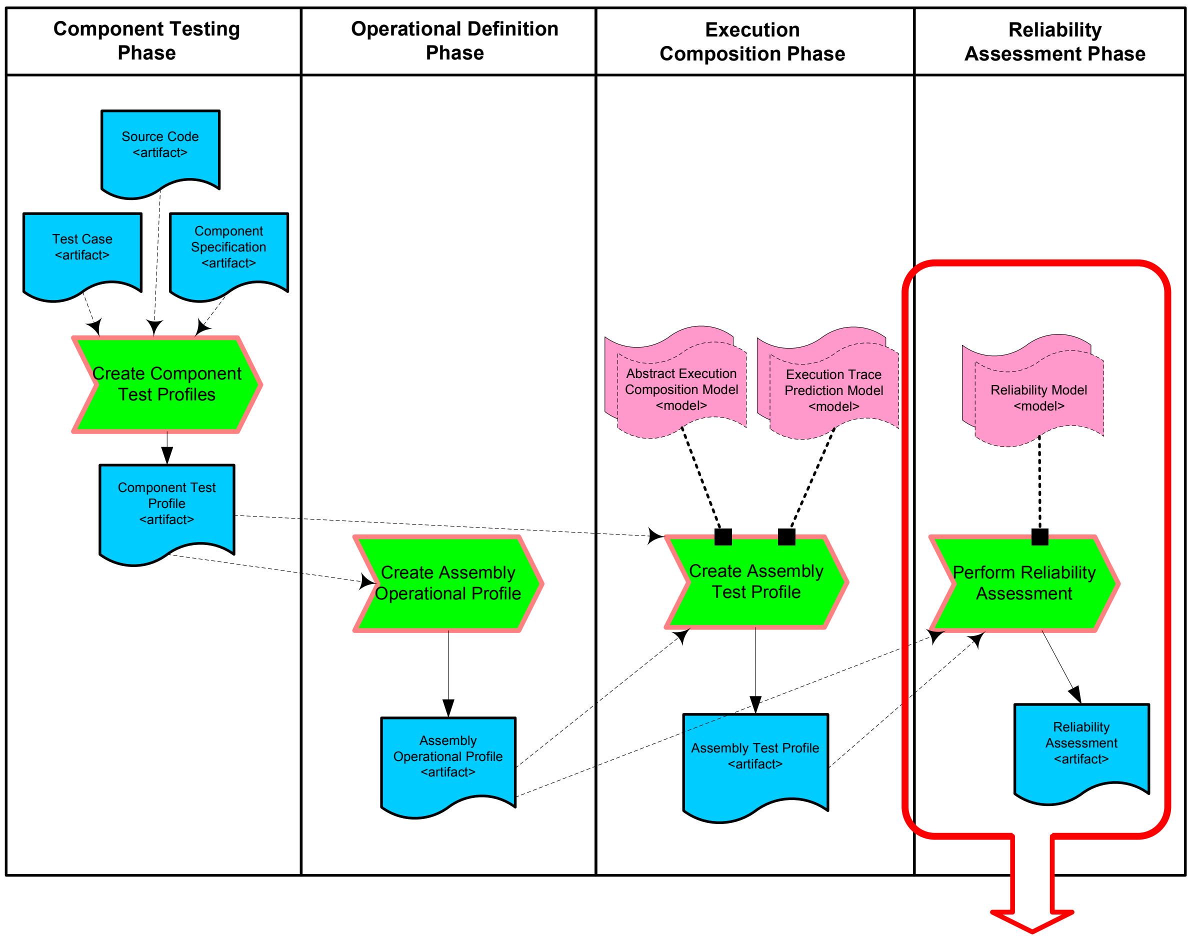
This chapter addresses the supporting claim: "*It is feasible to collect reliability-related information from the testing process.*" This claim is decomposed into two supporting claims: *"Trace execution information can be collected from the testing process"* that was discussed in Chapter 4 and *"Execution trace information allows one to define a consistent reliability model."* To support the latter claim, an important issue is addressed: defining a reliability model based on the information provided by the component test profiles. The Reliability Assessment Phase corresponds to the solution addressing such issue. Figure 36 shows the context of the Reliability Assessment phase as well as its required and produced artifacts.

This chapter is organized as follows: First, a detailed description of the problem is presented. Second, the rationale for the solution is explained. Third, a description of the task, models, and artifacts used in the reliability assessment phase is presented. Fourth, an ad-hoc reliability model is described. Fifth, a set of examples to illustrate the proposed solution are presented. Sixth, an overview of the implementation is described. Finally, a discussion of the results, limitations, and future extension of the proposed solution is included.

7.1 Problem Description

The problem addressed in this chapter is the definition of a reliability model that uses the information from the component and assembly test profiles. This is an important problem since its solution provides the theory required for the assembly reliability assessment.

Specifically, supporting claim 2 prescribes the use of execution traces to create a "consistent" reliability model. In this research, the term "consistent" implies two aspects: First, the model produces an estimate constrained by a point of comparison. This point of comparison is determined by the assembly operational profile. Second, the model produces an estimate using internal execution information. Therefore, the estimate provides a better characterization of the component behavior.


Component Testing Phase
Operational Definition Phase
Execution Composition Phase
Reliability Assessment Phase
Source Code <artifact>
Test Case <artifact>
Component Specification <artifact>
Create Component Test Profiles
Component Test Profile <artifact>
Create Assembly Operational Profile
Assembly Operational Profile <artifact>
Abstract Execution Composition Model <model>
Execution Trace Prediction Model <model>
Create Assembly Test Profile
Assembly Test Profile <artifact>
Reliability Model <model>
Perform Reliability Assessment
Reliability Assessment <artifact>

SUPPORTING CLAIM 2: RELIABILITY MODEL
Execution trace information allows to define a consistent reliability model.

Figure 36. Framework Expanded View with Focus on the Fourth Phase.

Claim number 2 can be decomposed in two problems. The first problem addressed in this chapter is related to the use of execution traces to produce a reliability estimate improving the black-box models. Black-box reliability models do not consider any internal information of the system. These models consider only the inputs tested and the failures found. For example, the model proposed by (Miller et al., 1992) produces an estimate using the number of failures found and the number of test cases performed. Other works, classified as input-domain models, consider not only the number of test cases performed but also the size and randomness of the input domain defined for the system (Nelson 1978).

The problem addressed in this chapter is related to the incorporation of operational information into the reliability assessment. The use of operational information to weight reliability estimations is a well-known strategy in Software Reliability Engineering (Musa 2004b). Other works involving experiences in using operational profiles are described elsewhere (Musa 1993) (Woit 1993). Examples of models describing operational information are presented in Chapter 5.

The reliability model proposed in this chapter assumes that all test cases in any component test profile are successful test cases (i.e., no failure are found). Moreover, the failure independence between execution nodes is also assumed.

The approach to solve the issues mentioned above is described in the following section.

7.2 Solution Approach

The solution approach is described as follows: First, the recursive composition approach is discussed. Second, the structure of the information contained within component test profiles that is used by the reliability model is explained. Third, the hierarchical approach used to define the reliability model is discussed. Finally, the strategy for estimating the confidence interval is described.

7.2.1 Recursive Composition Approach

Chapter 6 describes the composition model for sequential assemblies of components. The resulting composite model (i.e., assembly test profile) can be seen as another component test profile which includes additional information describing the prediction uncertainty. Figure 37 shows a graphical representation of this property. For example, the assembly C_c located at the bottom of the figure has three components (i.e., C_i, C_j, and C_k). The resulting assembly test profile can be represented as another component, named as C_c. The next upper assembly, named as C_2, has three components (i.e., C_a, C_b, and C_c). The same explanation follows for the upper levels until the entire system can be seen as the component S_1.

Using this approach, it is possible to define complex systems that are modeled as sequentially recurrent assemblies of components. This recursive property of the composition is also incorporated into the reliability model. Therefore, the reliability model uses the component test profile as a basic unit of abstraction.

7.2.2 Structure for the Reliability-related Information

The reliability model proposed in this chapter uses the structure of the information provided by assembly test profiles and the assembly operational profile. Figure 38 shows a diagram representing this structure. Rectangles denote basic information items such as execution nodes, execution traces, subdomains, and domains. Rectangles are connected vertically by lines denoting existence constraints. For example, the two lower boxes and their corresponding connection represent the fact that an execution trace is composed of one or more execution nodes. Similarly, a subdomain is composed of one or more execution traces. At the top level, the domain is composed of one or more subdomains and it represents the entire space of possible behavior or inputs for any component operation. Execution nodes and execution traces are gathered from component test profiles. The domain and subdomains are gathered from the assembly operational profile. Figures illustrating the ideas are shown at the right side.

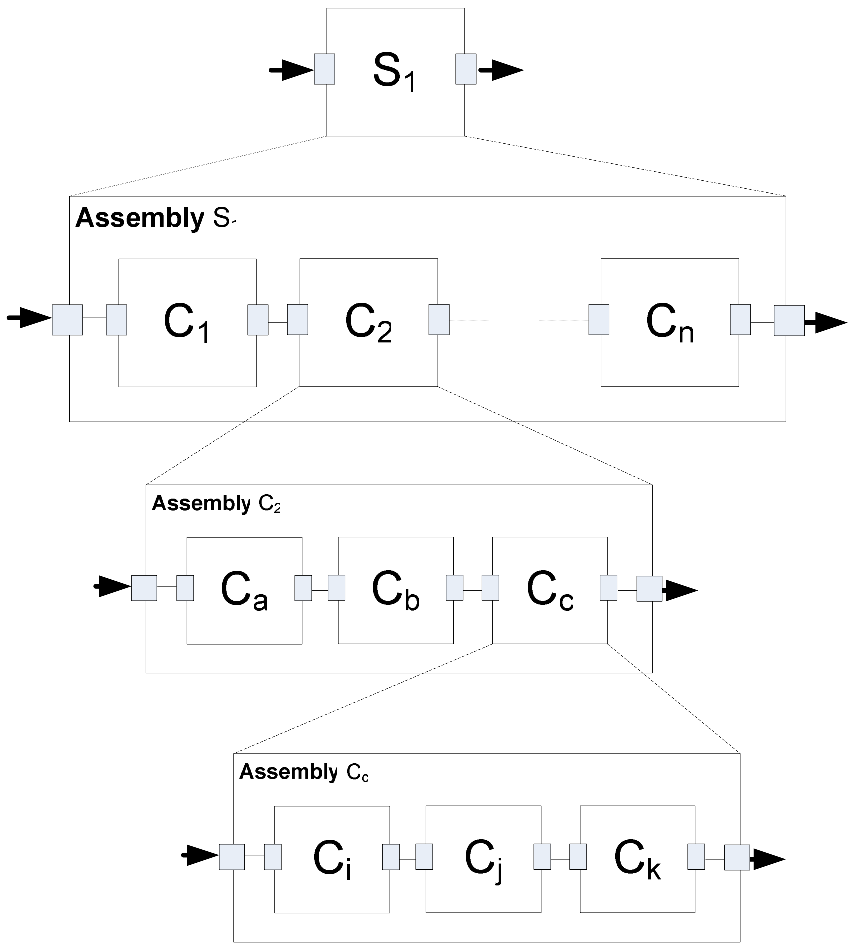


Figure 37. Assembly Abstraction Approach.

This structure is preserved in the reliability estimation. In other words, the reliability model uses this hierarchical structure. First, the execution node probability of failure is estimated. Next, the execution trace probability of failure is estimated using execution node probabilities of failure. Following the same approach, the subdomain probability of failure is estimated using the execution traces probabilities of failures. Finally, the domain probability of failure is estimated using the subdomain probabilities of failure.

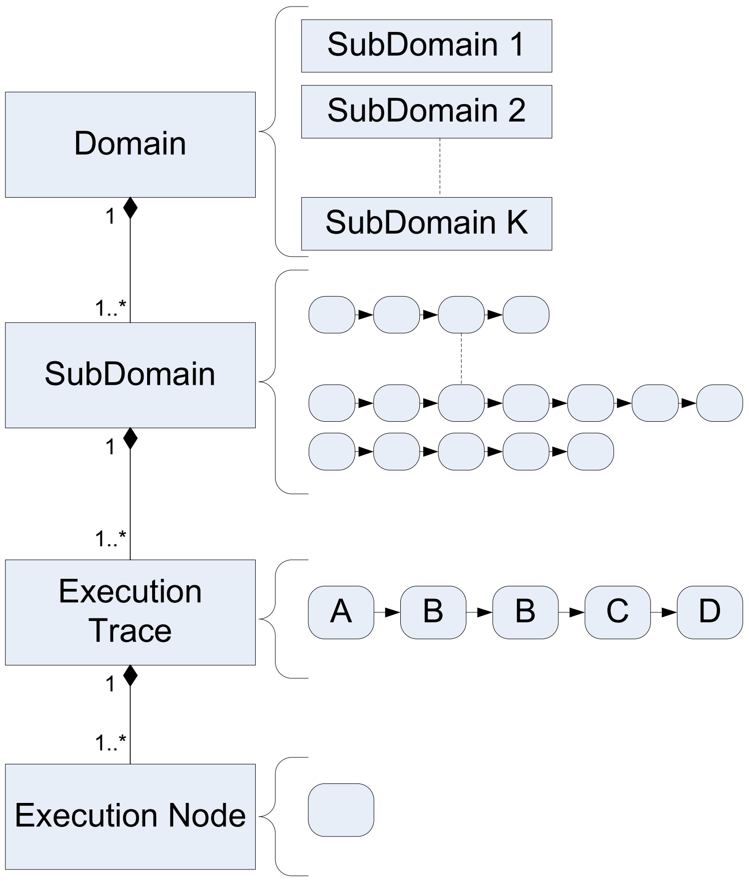


Figure 38. Reliability-related Information Structure.

7.2.3 Probability of Failure based on Execution Knowledge

The core part of the reliability model relies on the assumption that any execution node occurrence represents a distinct scenario where such execution node is tested. Such assumption is based on the idea that occurrences are produced by different test case inputs. There are cases where such assumption might not hold such as the interruption pooling mechanism where there is a loop over the same instruction waiting for an external response.

This model does not assume any distribution for the execution node probability of failure. The probability of failure is calculated using a Bayesian Estimator proposed by (Miller et al., 1992). This Bayesian approach allows the incorporation of prior assumptions into the test results but our model does not incorporate any prior assumption. This Bayesian estimator can be seen as a black-box approach to estimate the execution node probability of failure since there is no additional information within the execution node.

The estimation of the execution node probability of failure is similar to the problem of estimating the probability of finding a black ball after drawing *t* white-balls from an urn with replacement. This problem dates back to Laplace's work, when he derived the formula[7] (1):

[7] This is a special case for the Bayesian approach where the prior Beta Distribution has the parameters a=1, b=1.

$$\hat{\theta} = \frac{1}{t+2} \qquad (1)$$

The execution node probability of failure is estimated using (1). A detailed explanation of this formula can be found in (Miller et al., 1992).

Execution traces are the next information level according to the information structure described in Section 7.2.2. An execution trace represents an array of components connected sequentially. Intuitively, a failure in any execution node produces a failure in the corresponding execution trace. In other words, the execution trace is connected forming a series structure. Based on these ideas, the execution trace probability of failure is defined as

$$\text{Execution trace probability of failure} = 1 - (\text{Prob. of no failure in all execution nodes}) \qquad (2)$$

The probability of no failure in all execution nodes contained in the corresponding execution trace is estimated by using basic probability rules

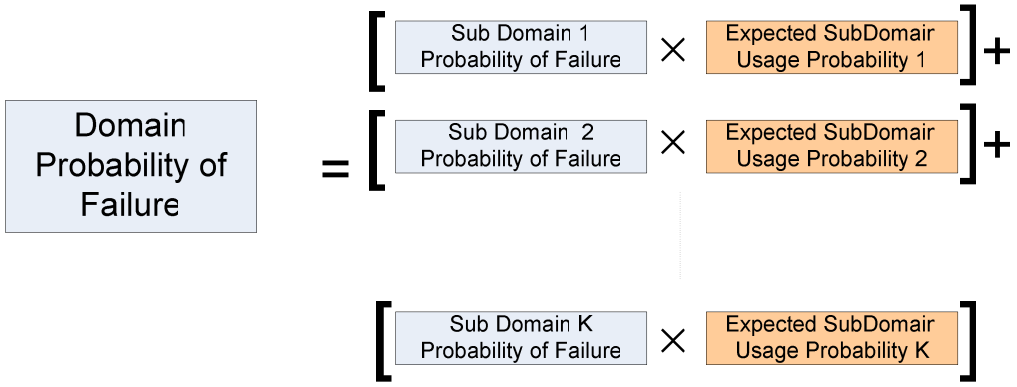
$$\text{Prob. of no-failure in all execution nodes} = \begin{array}{l} (1\text{- Prob. of failure in first execution node}) \times \\ (1\text{- Prob. of failure in second execution node}) \times \\ (1\text{- Prob. of failure in third execution node}) \times \\ \quad . \\ \quad . \\ (1\text{- Prob. of failure in last execution node}) \end{array} \qquad (3)$$

Subdomain is the next information level. In this case, the subdomain probability of failure is calculated as the mean of the all execution trace probability of failures contained in such subdomain. The use of this estimate is based on the assumption that the points in a given subdomain characterize the behavior of such subdomain.

The domain probability of failure is calculated in a similar way as in the subdomain probability of failure. However, at this step the probability calculation is slightly modified to incorporate operational information, as discussed in the next section.
A more detailed description of the reliability model is presented in Section 7.4.1.

7.2.4 Incorporating Operational Information

The approach to incorporate operational information is by calculating a weighted sum of the corresponding subdomain probabilities of failure with the probabilities representing the expected subdomain usage. Figure 39 shows a graphical representation of the incorporation of subdomain operational probabilities into the domain probability of failure estimation.


Domain Probability of Failure
=
Sub Domain 1 Probability of Failure
×
Expected SubDomain Usage Probability 1
+
Sub Domain 2 Probability of Failure
×
Expected SubDomain Usage Probability 2
+
Sub Domain K Probability of Failure
×
Expected SubDomain Usage Probability K
Figure 39. Incorporating Subdomain Usage Probabilities into the Reliability Estimation.

7.2.5 Point Estimation and Interval Estimation

The ideas mentioned in previous sections emphasize how point estimation is performed. This estimation is complemented with interval estimation which provides additional information about the estimation variance.

Bootstrap Sampling or Bootstrapping (Shelemyahu and Kenett 1998) is a well-known technique to perform interval estimation. Bootstrapping is a computer-intensive statistical technique. The payoff for such intensive computations is freedom from two major limiting factors that have dominated classical statistical theory since its beginning: the assumption that the data conform to a bell-shaped curve, and the need to focus on statistical measures whose theoretical properties can be analyzed mathematically. The name "bootstrap" was derived from an old saying about pulling oneself up by one's bootstraps. In this case, bootstrapping means redrawing samples randomly from the original sample with replacement.

The basic idea of Bootstraping is shown in Figure 40. This technique is used to assess the statistical accuracy of sample data (i.e., statistics of sample). The assessment is performed by taking *N* bootstrap samplings and computing the statistics from each bootstrap sampling. The values of bootstrap statistics are used to evaluate the statistical accuracy of the original sample statistics (i.e., by building the distribution of such sample statistics). There are distinct approaches to calculate a confidence interval. Our model determines the non-parametric interval estimation by selecting the 5th and 95th percentile of the empirical distribution produced by resampling. These two percentiles formed the limits for the 90% bootstrap percentile confidence interval.

A more detailed description of the Bootstrap technique can be found in (Shelemyahu and Kenett 1998) and in (Cheng 1995).

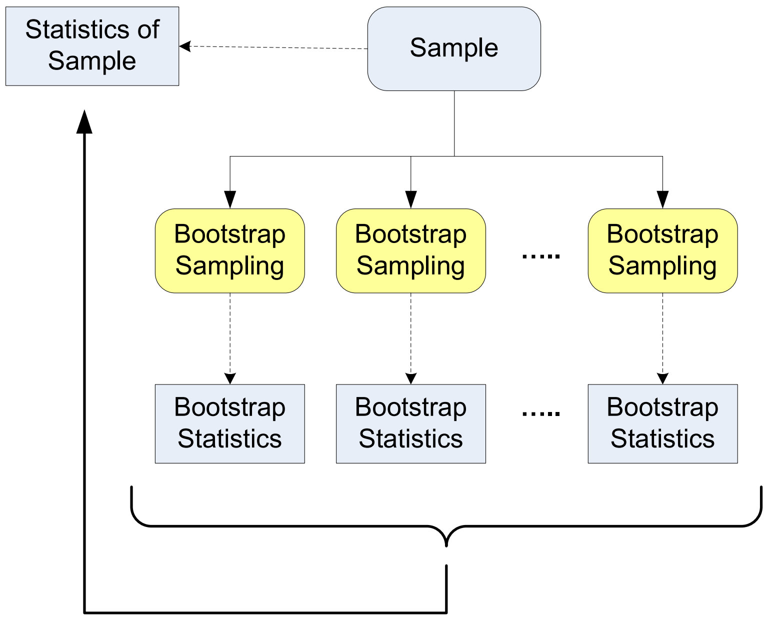


Figure 40. Bootstrap Sampling Approach.

7.3 Solution Description

This section describes how the ideas discussed in the previous section are instantiated into the proposed reliability analysis framework. A description of the artifacts and the task required for this phase are presented.

7.3.1 Artifacts Description

The input artifacts required in this phase are the assembly test profile and the assembly operational profile. The output artifact is the reliability assessment. The *Assembly Test Profile* artifact represents the composite behavior of the assembly, and the *Assembly Operational Profile* provides the point of reference used to weight the reliability assessment.

The *Reliability Assessment* artifact contains the results of the reliability assessment including both point and interval estimations.

A template for every artifact is provided in Appendix B.

7.3.2 Task: Perform Reliability Assessment

The *Perform Reliability Assessment* task is done during the fourth framework’s phase. The purpose of this task is the calculation of the assembly reliability estimate and the confidence interval based on the corresponding component test profiles and the assembly operational profile.

The activities involved in this task include the data processing required to perform the reliability assessment. This processing includes the estimation of execution node frequencies, execution node probabilities of failure, test case probabilities of failures, subdomain probabilities of failures, and the domain probability of failure. Data processing requires tools to perform the computations automatically. Specifically, the Bootstrap technique requires a considerable amount of computational resources.

7.4 Formal Model

This section describes the formal reliability model. This description uses the definitions given in Section 7.2. The reliability model is described using a set of complementary definitions addressing all levels of information described in Section 7.2.2.

7.4.1 Reliability Model

The following definitions provide the abstract representation of the information required for the model. This model uses some of the formalization described in Chapter 6 to define an ad-hoc reliability model on the top of the composition model.

Definition 7.1 (same as Definition 6.5)**.** A Test Profile, T_P, is a tuple such as

$$T_P = <\mathcal{P}, \mathcal{I}, \mathcal{O}, E>$$

where $\mathcal{I}$ denotes the input domain and $\mathcal{O}$ the output domain. E denotes a set of labeled execution nodes and $\mathcal{P}$ denotes a set of tuples such as

$$\mathcal{P} = <I, O, \mathcal{E}>$$

where I is an input domain instance, O is an output domain instance, and $\mathcal{E}$ represents a sequence of labeled execution nodes where $\mathcal{E} \in \mathbf{SEQ}(E)$. Where $\mathbf{SEQ}(E)$ represents the set of all possible execution traces. The cardinality of $|\mathcal{P}| = \mathcal{N}$, represents the number of test points or test cases in the test profile.

Definition 7.2. A Component Operational Profile, O_P, is a set of tuples such that

$$O_P = <D_i, p_i>$$

where D_i is a set and p_i is the corresponding probability such that $D_i \subset \mathcal{I}^*$, where $\mathcal{I}^*$ denotes the set of all possible input domain instances defined for the input domain $\mathcal{I}$ and

$$\sum_{i=1}^{K} p_i = 1 \qquad\qquad \bigcup_{i=1}^{K} D_i = \mathcal{I}^*$$

where $D_i \cap D_j = \emptyset$, for any i, j such that $i \neq j$ and $i, j \in \{1,2,\ldots K\}$.

<u>Definition 7.3</u>. The Execution Node Occurrence for the *i-th* Execution Node in the *j-th* subdomain, *v(i, j)* , is defined as:

$$v(i,j) = \sum_{m \in D_j} freq(m,i)$$

where *freq(m,i)* denotes the total number of times that the *i-th* execution node is found in the *m-th* element (i.e., *m* represents a test point within the subdomain).

<u>Definition 7.4</u>. The Probability of Failure for the *i-th* Execution Node in the *j-th* subdomain, $\hat{\phi}_k$, where *k* = *<i, j>*, is estimated by using the Bayesian Estimator (Miller 1992):

$$\hat{\phi}_k = \frac{1}{v(i,j)+2}$$

where *v(i, j)* denotes the *i-th* execution node occurrence found in the *j-th* subdomain.

<u>Definition 7.5</u>. The Probability of Failure for the *j-th* test point, $\hat{\tau}_j$, is estimated by:

$$\hat{\tau}_j = 1 - \prod_{k \in G.j} (1 - \hat{\phi}_k)$$

where *G.j* denotes the multiset representing the execution nodes found in *j-th* test point (i.e., execution trace).

Definition 7.6. The Probability of Failure for the *i-th* subdomain, $\hat{\theta}_i$, is estimated by:

$$\hat{\theta}_i = \frac{1}{n_i}\left(\sum_{m \in D_i} \tau_m\right)$$

where n_i denotes the number of points found in the *i-th* subdomain, $n_i = |D_i|$. This probability represents the simple average of all test point probabilities of failure.

Definition 7.7. The Domain Probability of Failure, $\hat{\theta}$, is estimated by:

$$\hat{\theta} = \sum_{i=1}^{K} \hat{\theta}_i p_i$$

where p_i denotes the *i-th* subdomain operational probability. This probability is estimated by computing a weighted sum of all subdomain probabilities of failure.

7.5 Example

This section describes a basic example to illustrate how the reliability model works. A square root computation component is assessed using the model proposed in this chapter.

7.5.1 Square Root Component Reliability Assessment

The information required for the reliability model includes the assembly operational profile and its corresponding component test profile. In this case the assembly has only one component. The component test profile, described in Section 4.4.1, includes 5000 test points with their corresponding input, output, and execution trace. There are four execution nodes defined for this component (i.e., *A*, *B*, *C*, and *D*). A fragment for the square root component test profile is found in Table 10.

A fragment of the component operational profile is shown in Table 20. This table contains the required information to identify which test points are contained in each subdomain. Four subdomains are defined in this operational profile. We assume that the corresponding subdomain usage probabilities are known.

Component Operational Profile			
Subdomain	**Lower**	**Upper**	**Usage Probability**
1	0	1000	0.2
2	1001	5000	0.4
3	5001	20000	0.3
4	20001	65535	0.1

Table 20. Square Root Computation Component Operational Profile.

The model calculates the execution trace distribution to estimate the execution node probabilities of failure. The execution trace distribution is built by classifying all execution traces according to their subdomain membership and by counting the frequencies of every execution node (i.e., Definition 7.3). The resulting distribution is shown as a matrix where the rows denote execution nodes and the columns denote subdomains.

$$V = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 78 & 667 & 78 & 78 \\ 1068 & 11538 & 1068 & 1068 \\ 382 & 4580 & 382 & 382 \\ 3472 & 42771 & 3472 & 3472 \end{pmatrix} \end{array}$$

Next, the execution node probabilities of failure are estimated using the execution trace distribution (i.e., Definition 7.4). Table 21 shows the execution node probabilities of failure.

	Execution Node Probability of Failure			
Subdomain	***A***	***B***	***C***	***D***
1	$1.25E^{-02}$	$1.49E^{-03}$	$1.25E^{-02}$	$1.25E^{-02}$
2	$9.35E^{-04}$	$8.67E^{-05}$	$9.35E^{-04}$	$9.35E^{-04}$
3	$2.60E^{-03}$	$2.18E^{-04}$	$2.60E^{-03}$	$2.60E^{-03}$
4	$2.88E^{-04}$	$2.34E^{-05}$	$2.88E^{-04}$	$2.88E^{-04}$

Table 21. Execution Node Probabilities of Failure.

The test point probability of failure is estimated using the corresponding execution node probabilities of failure (i.e., Definition 7.5). A partial list of the test point probabilities of failure is shown in Table 22.

Input	Output	Execution Trace	Test Point Probability of Failure
1	1	*ABCD*	0.03847262
43	6.557438	*ABBBBBBBCD*	0.04706402
49	7	*ABBBBBBBCD*	0.04706402
55	7.416198	*ABBBBBBBCD*	0.04706402
56	7.483315	*ABBBBBBBCD*	0.04706402
.	.	.	.
.	.	.	.
.	.	.	.

Table 22. Test Point Probabilities of Failure.

The next step is the estimation of the subdomain probabilities of failure. These probabilities represent an average of all test point probabilities in the corresponding subdomain (i.e., Definition 7.6). Table 23 shows the corresponding subdomain probabilities of failure.

Subdomain	Probability of Failure
1	0.049271616
2	0.003734267
3	0.010385330
4	0.001151028

Table 23. Subdomain Probabilities of Failure.

Finally, the point estimation for the domain probability of failure or component probability of failure is (i.e., Definition 7.7):

$$\hat{\theta} = 0.01640990$$

The confidence interval estimation is performed by generating 10,000 Bootstrapping samples. Each bootstrapping sample is generated by selecting 5000 random points with replacement from the component test profile. The domain probability of failure is estimated for each sample. Table 24 shows a partial list of the estimated domain probabilities of failures. Figure 41 shows the empirical CDF determined from all bootstrapping samples.

Bootstrapping Sample	Statistic
1	0.01485100
2	0.01348857
3	0.01379082
4	0.01425098
5	0.01644132
.	.
.	.
.	.

Table 24. Domain Probabilities of Failure Estimated by Bootstrapping Sampling.

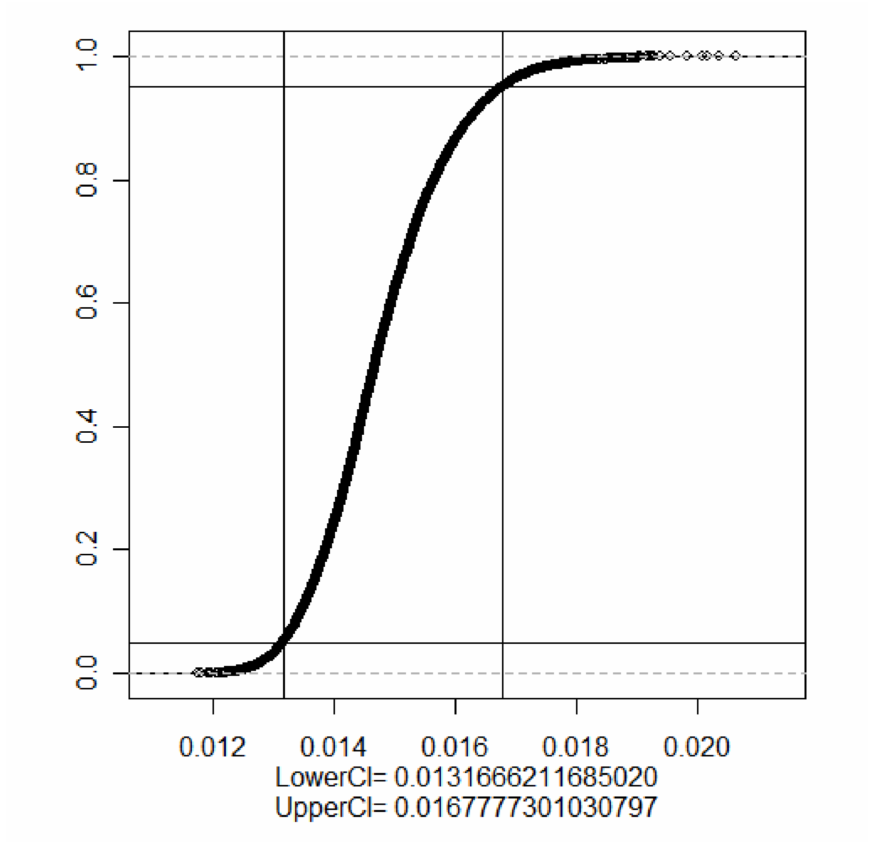


Figure 41. Plot Showing the Bootstrapping Sampling Statistic CDF.

An approximate 90% confidence interval estimation is obtained by selecting the corresponding quantiles for 5% and 95%. The resulting estimation interval is (0.0131666, 0.0167777).

7.6 Tool Support and Implementation Issues

This research includes the implementation of a module that performs the reliability assessment using the component test profile information contained within a MS Access database. This tool is implemented with VBA scripts and R statistical functions. The statistical functions provided by the R environment reduce the computation time required for the bootstrapping sampling.

7.7 Discussion

This chapter describes the solution that addresses the supporting claim: “Execution trace information allows one to define a consistent reliability model.” This chapter describes a reliability model fulfilling the aspects about “consistency.” First, the estimations derived from the reliability model are weighted using the operational profile as point of reference.

Second, the model uses internal information represented by the set of execution node sequences.

As in the case of any reliability model, the proposed model has some limitations. One of such limitations is the fact that the model uses the execution node frequencies rather than the entire execution trace. The incorporation of such ordering and dependency might require the removal of the assumption about the independence of the failures. However, the model requires using more sophisticated statistical mechanisms such as Markov Chains or the use of Bayesian Techniques to address the dependency issue. Additionally, more sophisticated prediction models are required to predict not only the frequencies, but also the sequence itself.

Other limitation of this model is the assumptions about the absence of failure in the test cases. In practice this assumption is difficult to hold. The incorporation of failures requires redefining the reliability model by considering issues such as the location of the failure in terms of execution nodes and the possible dependence of such failure.

However, the model presented in this chapter has major advantages. This model can be extended to use more information in the execution node probability of failure estimation. This extension does not affect the remaining formulations. Additionally, this model provides confidence interval estimations without any assumption about the failure distribution. The calculations are easily implemented.

Finally, the model described in this chapter can be considered as a general case of some input-data models such as the model proposed by (Hamlet 1994). The particular case of this model is when there is a only one component with one execution node.

7.8 Summary

This chapter has addresses the supporting claim: "Execution trace information allows one to define a consistent reliability model." The solution is a hierarchical model that uses the information within component test profiles.

The contributions derived of this chapter include:

- A reliability prediction model based on the information contained in the component test profiles. This model produces a reliability point estimate and the corresponding confidence interval estimation.
- The design and implementation of a software system that partially automates the task defined for the fourth phase in the framework.

The next chapter will describe the validation performed for the major claim stated in this dissertation.

Chapter 8

Evaluation and Validation

This chapter addresses the major claim: "*It is feasible to produce a meaningful (significant, useful) reliability assessment of software component assemblies, before their actual integration, based on (i) component abstract execution information collected in (ii) a limited number of previously executed component test cases and (iii) information about the expected assembly use*". Evidence supporting this claim is provided in this chapter.

8.1 Validation Approach

The strategy proposed to validate the claims stated in this dissertation follows the claim structure shown in Figure 6 (c.f., page 31). The validation of all supporting claims uses experimental results showing that it is feasible to use the framework for reliability assessment. Specifically, experimental evidence is provided to sustain every supporting claim.

The nature and scope of the experiments reported in this chapter provide evidences to guarantee the major claim. However, the estimation of the probability of failure shares the same issue that any estimate has: the accuracy and confidence in the estimation depend on the quality of the information analyzed. In our research, the quality depends on the prediction accuracy. Section 8.4 provides a more comprehensive description of the results obtained through experimentation and the scope of such evidences.

The arguments supporting the claims come from two case studies describing systems that are defined as sequential assemblies of components. The first system is a face-based monitoring security system and the second is a data compression and encryption system. The information described for every case study includes a brief system description, a domain characterization, a summary of the most relevant artifacts, results from the reliability assessment, and a discussion of the results.

8.2 Case Study: Security Monitoring System

This section describes a case study of a Security Monitoring System (SMS). The SMS' goal is to monitor user accesses to different locations based on facial recognition. Figure 42 shows the architecture for this system. The SMS is composed of three software components, namely, facial detection, image converter, and facial recognition component. These three components are connected together in a dataflow architecture which is shown in Figure 42. The facial detection component processes the images from different cameras and inspects these images to detect the presence of people. If a person is found in the captured image, the facial part of the individual is detected and extracted from the image. The image converter component converts the image into a suitable format for the face recognition component. The facial recognition component compares the extracted facial image with images in its own embedded user knowledge database. If it matches any image in the database, nothing happens. If it does not match any image in the database, it is assumed that the person is an intruder and an alarm is activated.

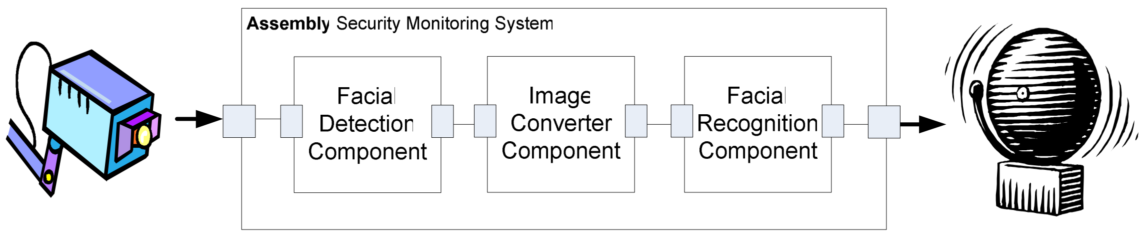

Assembly Security Monitoring System
Facial Detection Component
Image Converter Component
Facial Recognition Component

Figure 42. SMS Architecture.

8.2.1 System and Component Domains

The input domain complexity of the components in SMS is higher than the domain complexity in the mathematical computation example discussed in previous chapters. The input domain for the system is based on image files. These image files have size-related attributes, such as width and height, which may be used as elements to characterize the input domain.

However, there is an important semantic content in the image that can not be characterized considering only the size-related attributes, as mentioned in Section 5.2.2. Examples of semantic content are all facial properties of the people contained within the image file. This issue is addressed by defining a differentiation ontology, as discussed in Chapter 5. Figure 43 shows the set of attributes associated with the input domain for the SMS assembly. This ontology groups all attributes from the respective component ontologies.

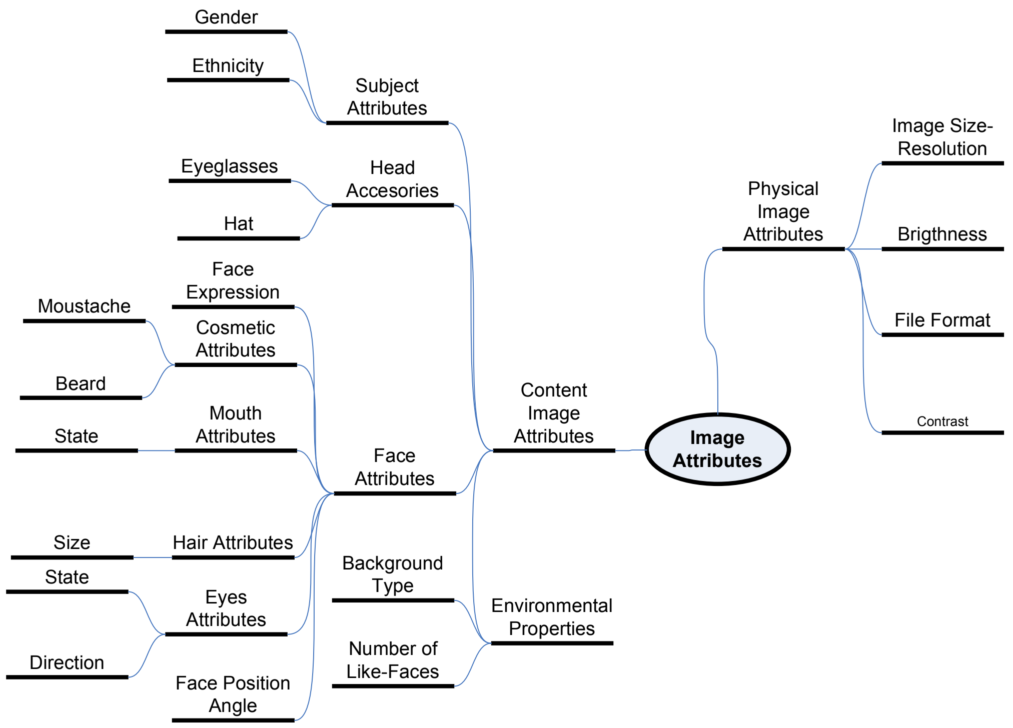


Figure 43. SMS Differentiation Ontology.

Figure 44 shows the corresponding differentiation ontology defined for the Face Detection Component. This is the same ontology described in Section 4.4.2.

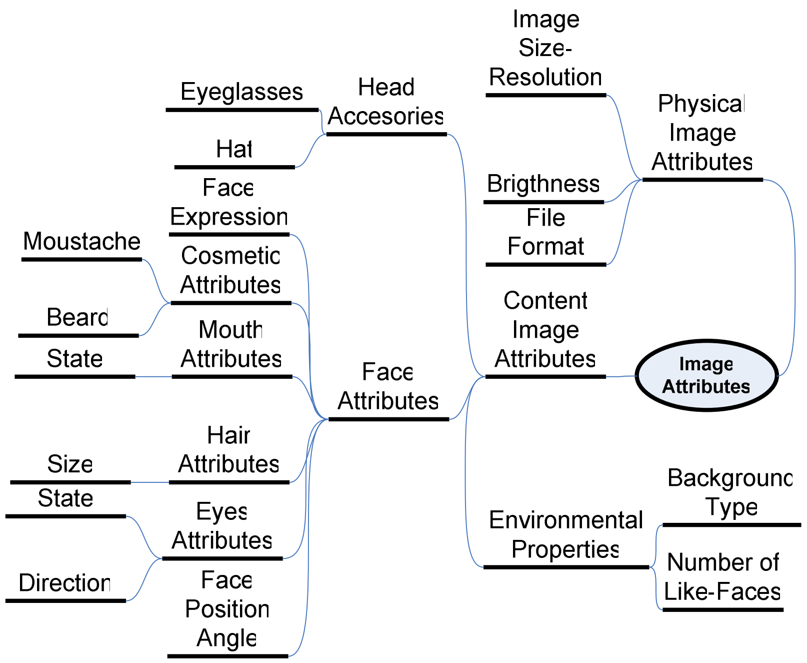


Figure 44. Face Detection Component Differentiation Ontology.

Figure 45 and Figure 46 show the differentiation ontology defined for the Face Recognition Component and the Image Converter Component, respectively.

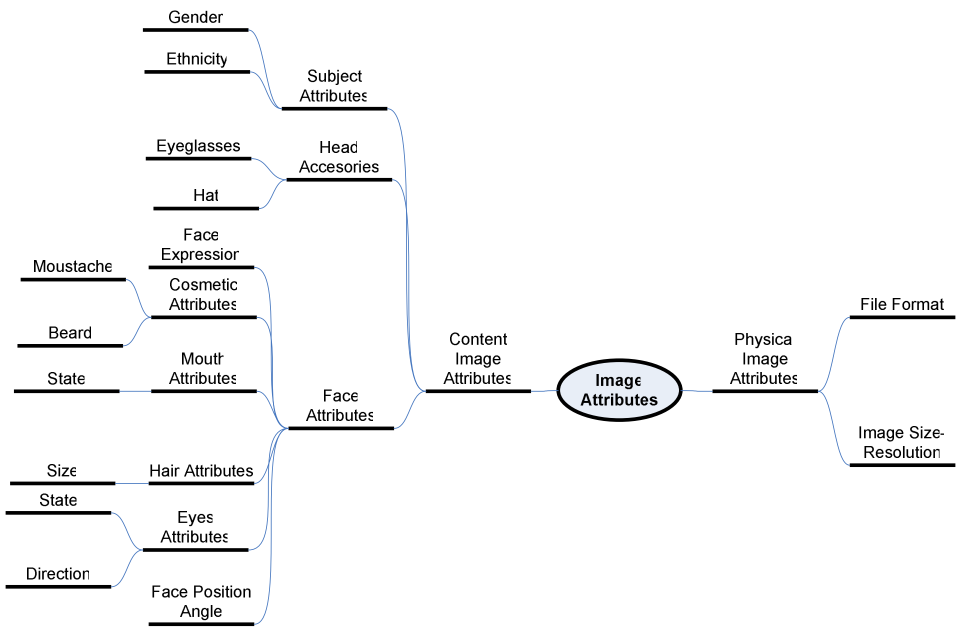


Figure 45. Face Recognition Component Differentiation Ontology.

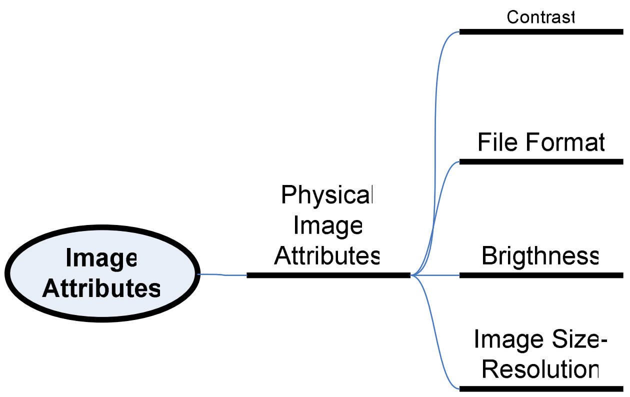


Figure 46. Image Converter Component Differentiation Ontology.

All test cases found in every component test profiles are organized based on such differentiation ontologies.

8.2.2 Artifacts

The most relevant artifacts in this case study are the set of component test profiles, the assembly operational profile, and the reliability assessment artifact.

The format and content of all component test profiles is similar to that shown in Table 13 and Table 14 (c.f., page 55). The number of component test cases is shown in Table 25. The set of test cases was obtained from training datasets developed by the research community working on face recognition (Grgic and Delac 2007). The number of initial test cases was higher, but they were reduced after using the differentiation ontologies defined for each component.

Component	Number of Test Cases Before Classification
Facial Detection	624
Image Converter	200
Facial Recognition	230

Table 25. Number of Test Cases Performed for SMS's Components.

The assembly operational profile defined for this system uses the differentiation ontology to characterize three subdomains. It is assumed that the system is located in a building where three areas are monitored, namely, production, corporate, and storage areas. Every area has its own properties defined by the people working there and its corresponding environmental conditions. Three different operational sub-domains are defined for the system:

- Subdomain 1 (corporate area). This area is considered for male population with additional specific attributes. The probability for this sub-domain is 0.254.
- Subdomain 2 (storage area). This area is considered for female population using protection glasses with additional specific attributes. The probability for this sub-domain is 0.068.
- Subdomain 3 (production area). This area is considered for male population with additional specific attributes. The probability for this sub-domain is 0.678.

The complet assembly operational profile defined for this system is shown in Table 16 (c.f., page 69).

8.2.3 Assembly Composition and Prediction

Statistical prediction models such as artificial neural networks or classification trees could not be used as prediction techniques in this case study. Experiments using these statistical techniques did not show good predictive results. Thus, we use the best binary matching technique during the composition process.

The composition follows the model defined in Section 6.4.2. In this case, the composition uses the test cases defined for the first component in the assembly. Then, the composition model finds the best matching point in the second component connected in the assembly. Similarly, using the best matched test case in the second component, the composition looks for the best matched test case in the third component. Using these test cases, the composite execution trace is produced.

The experimental metric described in Section 6.4.1, named as cumulative error in selection, is shown in Table 26. This metric represents the normalized average number of different bits found in the composite test case. For example, the smallest cumulative error in selection is 0.278. This value means that the 27% of the bits are different.

Subdomain	Cumulative Selection Error	Number of Test Points
1	0.825	310
2	0.278	34
3	0.798	182

Table 26. Subdomain Cumulative Selection Error.

8.2.4 Reliability Assembly Assessment

The reliability model uses execution node probabilities of failure as basic units. A partial list of the execution node probabilities of failure is shown in Table 27. The number within brackets denotes the subdomain associated with the estimation.

Component	Execution Node	Occurrences	Probability of Failure
FACE DETECTION[1]	A1	310	3.205E-03
FACE DETECTION[1]	A2	310	3.205E-03
FACE DETECTION[2]	A1	34	2.778E-02
FACE DETECTION[2]	A2	34	2.778E-02
FACE DETECTION[2]	A5	34	2.778E-02
FACE DETECTION[2]	A7	34	2.778E-02
FACE DETECTION[2]	C7	34	2.778E-02
.	.	.	.
.	.	.	.
.	.	.	.
IMAGE CONVERTER	A1	202	4.902E-03
IMAGE CONVERTER	A2	202	4.902E-03
IMAGE CONVERTER	A3	100	9.804E-03
IMAGE CONVERTER	A5	202	4.902E-03
.	.	.	.
.	.	.	.
.	.	.	.
FACE RECOGNITION	A1	230	4.310E-03
FACE RECOGNITION	A2	230	4.310E-03
FACE RECOGNITION	A3	230	4.310E-03
FACE RECOGNITION	A4	460	2.165E-03
.	.	.	.
.	.	.	.
.	.	.	.

Table 27. Partial List of Execution Node Probabilities of Failure.

Finally, using the reliability model described in Chapter 7, the estimation of the assembly probability of failure (i.e., point and interval estimation) is shown in Table 28. This table also shows the corresponding subdomain probabilities of failure. Similar to other artifacts, a Reliability Assembly artifact template is described in Appendix B.

Subdomain	Subdomain Probability of Failure	Number of Test Points	Subdomain Operational Probability
1	0.1113	310	0.254
2	0.4001	34	0.678
3	0.1445	182	0.068
ASSEMBLY PROBABILITY OF FAILURE (Point Estimation)			**= 0.309352474**
ASSEMBLY PROBABILITY OF FAILURE (Interval Estimation, 90%)			**= (0.26567, 0.35890)**

Table 28. SMS Assembly Reliability Assessment.

8.2.5 Discussion

The probability of failure obtained from the assessment is a measure about the degree of failure-free operation confidence provided by the component test profile information. In this case, the assessment estimates a considerable probability of failure (i.e., 0.30935). Intuitively, the circumstances originating this value are related with the high operational probability specified for the second subdomain (i.e., 0.678) and the small number of test cases associated with such subdomain (i.e., 34 test cases). This number of test cases does not provide enough evidences of the failure-free component operation in such subdomain.

The composition process was performed following the composition model described in Chapter 6. The selection of prediction models is an issue in this case study. The prediction model for these components used the BBMT due to the poor performance of other statistical models. The reasons for such weak performance were the small number of test cases defined for every component and the high number of categorical variables defined for the input domain in every component.

The operational specification of this assembly used differentiation ontologies to characterize the components. However, the creation of the component test profiles required a considerable effort. The time required for the test case classification was particularly high. The instrumentation and component testing were performed with no significant overhead.

The relevance of this case study comes from the information provided to support the claims defined in this dissertation. Table 29 shows the most important claims and the evidence provided by the case study.

Claim	**Comments and Evidences**
It is feasible to collect reliability-related information from the testing process	The execution trace information was collected using source code manual instrumentation and a simple library that implemented the monitoring infrastructure.
Operational information can be specified in a well-suited form that improves the reliability assessment.	The operational information was modeled using the notation proposed by the UML profile. This information was used for the reliability assessment.
It is feasible to compose execution trace information from partial information of different components	The prediction used the BBMT. Other prediction models did not show good results because of the reduced number of test cases and the input domain characteristics. The composition process uses as *covered* criterion the best matched test cases.
It is feasible to produce a meaningful (significant, useful) reliability assessment of software component assemblies, before their actual integration, based on (i) component abstract execution information collected in (ii) a limited number of previously executed component test cases and (iii) information about the expected assembly use.	The results from the case study provide evidence to support this claim. However, the uncertainty of the prediction model seems to play a key role in the assessment.

Table 29. Claim Evidences Comments for SMS Case Study.

This case illustrates the applicability of the framework. However, there are two issues to be considered before using the framework: the uncertainty of the prediction models and the required effort for test point classification. These issues are discussed in Section 8.4.

8.3 Case Study: Compression-Encryption Assembly

This section describes a case study of a Compressing-Encrypting Assembly (C&E). The C&E goal is to compress and encrypt character streams (i.e., strings). This assembly is used as a part of a secure communication system. Figure 47 shows the architecture for this assembly. The C&E assembly has two software components, namely, the compression component and the encryption component. These components are inter-connected defining a dataflow architectural style. The compression component processes an input string with some additional parameters. The output of such component is a smaller string. The encryption component receives this compressed string and it performs an encryption process on the compressed string. The output of such component is the compressed and encrypted string.

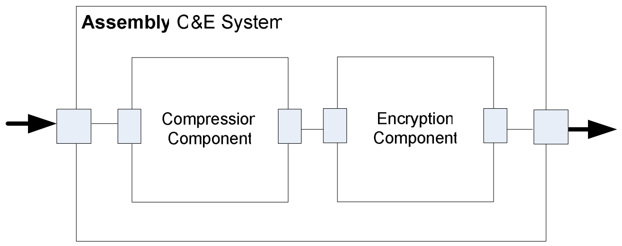


Figure 47. C&E Architecture.

8.3.1 System and Component Domains

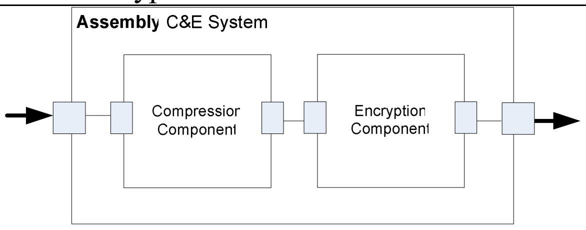
The input domain for the assembly is less complex than the previous case study (i.e., SMS assembly). The input considered for the system is based on strings. The physical information provides enough information to characterize the operational subdomains. In this case, the input domain defined for every component uses the size and string diversity with additional parameters. String diversity represents the heterogeneity of characters found in a string.

8.3.2 Artifacts

This assembly is used to compress and encrypt strings for a security module that stores emails, passwords, and answers to secret questions. The scenarios defined for the assembly are:

- Subdomain 1 (password encryption). This subdomain is defined as the set of strings representing passwords. The probability for this sub-domain is 0.50.
- Subdomain 2 (secret answer). This subdomain is defined as the set of strings representing answers to the secret question that is used to reset account passwords. The probability for this sub-domain is 0.250.
- Subdomain 3 (email address). This subdomain is defined as the set of strings representing user email addresses. The probability for this sub-domain is 0.250.

Table 30 shows the operational profile associated with the C&E assembly. Subdomains use the length of the string (i.e., using the symbol "| |") and the allowed character sets (i.e., using PERL regular expressions) to characterize the input data.

Assembly Name	C&E Assembly
Purpose	Compress and Encrypt a character stream.
Assembly Specification	 Assembly C&E System Compression Component Encryption Component
Quantitative Model	

	Subdomain Description (*InputString*)			**Comments**
	Subdomain	**Characterization**	**Probability**	
Password	1	|*InputString*| > 6 AND |*InputString*| < 12 AND *InputString* in PRegExp	0.50	
Secret Answer	2	|*InputString*| > 1 AND |*InputString*| < 50	0.25	
Email Address	3	|*InputString*| > 1 AND |*InputString*| < 25 AND *InputString* in MRegExp	0.25	
	Regular Expressions			

	PRegExp = [a-zA-Z][\w\.-]*[a-zA-Z0-9]
	MRegExp = [a-zA-Z][\w\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\w\.-]*[a-zA-Z0-9]\.[a-zA-Z][a-zA-Z\.]*[a-zA-Z]			

Table 30. C&E Assembly Operational Profile.

The component test profiles defined for this assembly are both similar in their information structure. Table 31 shows the Compression Component Test Profile. The encryption component test profile required some modifications to achieve the composition process, as discussed later in this section. The numbers of test cases contained in the compression and encryption component were 1000 and 2000, respectively.

8.3.3 Assembly Composition and Prediction

Artificial neural networks were selected as prediction models. The number of test points and their randomness allowed the generation of good prediction models since the mean square error for each component is less than 15%. However, the second component characterization was modified because the matching condition required other additional attributes.

The composition used a matching condition defined over the length of the strings and the string diversity. The string diversity represents the heterogeneity of characters found in the string. For example, the string "AABCBBACB" has lower string diversity than "ABCVFFGHWERTS". String diversity is defined as the number of different characters found in a specific string.

Component Test Profile		
Component Name	Compression Component	
Component Functionality	**Functionality Signature**	**Domain Description**
Compress an String	Compress(*inputString*, *compLevel*, *compStrategy*): *outputString* *inputString*: Parameter that contains the string that is compressed. *outputString*: Return value that represents the compressed string *CompLevel*: Parameter representing the compression level. *CompStrategy*: Parameter representing the compression strategy.	**Implemented** *inputString* < 1.2 GB *outputString* < 1.2 GB *compLevel* $\in$ {1, 2, 3, 4} *compStrategy* $\in$ {1, 2, 3}

Test Data						
Test Case Number	**Input String**[8]	**Compression Strategy**	**Compression Level**	**Expected Output String**	**Execution Trace**	**Test Result**
1	3Hbcfh28Pd70 Op1tNJ83tO10 UIw8Afc9e…	1	1	0wQ6U2OAG or95kgYJjeRr r56t…	A1B1B1C1 D1D1…	Correct
2	7jaH5rB51nrY 327Ckp1xSw8 FF683XckR4s 58lsT6gj…	1	2	7jaH5rB51nr Y327Ckp1xS w8FF683Xck R4…	A1B1B1C1 D1E1D1E1 …	Correct
3	23GF1Uj4C8L 4J8…	1	1	SW77jDWUc Z311…	A1B1B1C1 D1D1…	Correct
4	RGeczVq5flmc 5sTs24v…	1	2	H7Xq54Y3A6 6tZ8ak2zX5yt toU1uG…	A1B1B1C1 D1E1D1E1 …	Correct
5	kYmX095v8JO 16384rw1E219 If…	1	1	90hv7D3qHV HQ0YAm4zo o2Kq6Gjf..	A1B1B1C1 D1D1…	Correct
. . .	. . .	. . .	. . .	. . .	. . .	. . .

Table 31. Compression Component Test Profile.

String diversity and string length were the input parameters characterizing the second component. This extended characterization did not change the uncertainty of the prediction model. Table 32 shows a partial list of the modified test points in the second component test profile.

[8] The symbol "…" denotes that more characters are omitted from the text.

Test Data						
Test Case Number	Input String	String Length	String Diversity	Expected Output String	Execution Trace	Test Result
1	3Hbcfh28Pd70 Op1tNJ83tO10 UIw8Afc9e…	1225	145	0wQ6U2OAG or95kgYJjeRr r56t…	A2B2B2C2 C2C2…	Correct
2	7jaH5rB51nrY 327Ckp1xSw8 FF683XckR4s 58lsT6gj…	45334	117	7jaH5rB51nr Y327Ckp1xS w8FF683Xck R4…	A2B2B2B2 B2B2…	Correct
3	23GF1Uj4C8L 4J8…	88	67	SW77jDWUc Z311…	A2B2B2C2 C2C2…	Correct
4	RGeczVq5flmc 5sTs24v…	978	74	H7Xq54Y3A6 6tZ8ak2zX5yt toU1uG…	A2B2B2C2 C2C2…	Correct
. . .	. . .	. . .	. . .	. . .	. . .	. . .

Note: The symbol "…" denotes that more characters are omitted from the text.

Table 32. Modified Test Data Section from the Encryption Component Test Profile.

8.3.4 Reliability Assembly Assessment

Similarly to the previous case study, a partial list of execution node probabilities of failure is shown in Table 33.

Component	Execution Node	Occurrences	Probability of Failure
COMPRESSION[1]	A1	430	2.31E-03
COMPRESSION[1]	B1	175	5.65E-03
COMPRESSION[1]	C1	255	3.89E-03
COMPRESSION[1]	D1	125000	8.00E-06
. . .	. . .	. . .	. . .
ENCRYPTION	A2	1093	9.13E-04
ENCRYPTION	B2	994	1.00E-03
ENCRYPTION	B3	1874	5.33E-04
ENCRYPTION	B4	22344	4.48E-05
. . .	. . .	. . .	. . .

Table 33. C&E Assembly Execution Node Probabilities of Failure.

Finally, using the reliability model described in Chapter 7, the estimation of the assembly probability of failure (i.e., point and interval estimation) is shown in Table 34. This table also shows the corresponding subdomain probabilities of failure.

Subdomain	Subdomain Probability of Failure	Number of Test Points	Subdomain Operational Probability
1	0.0109	430	0.50
2	0.0340	350	0.25
3	0.0217	220	0.25
ASSEMBLY PROBABILITY OF FAILURE (Point Estimation)			**= 0.019375**
ASSEMBLY PROBABILITY OF FAILURE (Interval Estimation, 90%)			**= (0.01167, 0.02254)**

Table 34. C&E Assembly Reliability Assessment.

8.3.5 Discussion

This case study exposes two important issues impacting the practical utilization of the framework. The first aspect is the work required to adapt the characterization of the second component (i.e., encryption component). This issue may impact the scalability of this approach since it is required to perform a deep analysis on every component connection. This analysis is based on the matching criterion defined for the component assemblies.

The second issue is the input domain re-characterization feasibility. It is possible that a domain re-characterization may require a considerable effort. This issue includes the effort required to implement the matching criterion in the tool performing the composition. This tool may require the implementation of sophisticated mechanisms to verify the matching condition.

The C&E reliability assessment shows a relatively small probability of failure (although the qualification depends on the specific context). The number of test points is not as small as in the previous case study.

The most relevant issue during the composition process was the additional work to re-characterize the input domain for the second component. The prediction model used artificial neural networks with good prediction results.

The assembly operational specification contains subdomains characterized by the structure and content of the input strings. In this case, the characterization of such subdomains was specified using PERL regular expressions.

Table 35 shows the most important claims and the evidence provided by this case study.

Claim	Comments and Evidences
It is feasible to collect reliability-related information from the testing process	The execution trace information was collected using semi-automatic instrumentation and using the monitoring infrastructure mentioned in Chapter 4.
Operational information can be specified in a well-suited form that improves the reliability assessment.	The UML profile was used to model the operational information The information in the model was used for the reliability assessment.
It is feasible to compose execution trace information from partial information of different components	The prediction model used the artificial neural network technique. Even with the modification in the Encryption Component Test Profile, artificial neural network provided good results. The composition was performed using the model described in Chapter 5. However, the composition required some modifications to the component test profile of the second component.
It is feasible to produce a meaningful (significant, useful) reliability assessment of software component assemblies, before their actual integration, based on (i) component abstract execution information collected in (ii) a limited number of previously executed component test cases and (iii) information about the expected assembly use.	The results from this case study provide evidences to support this claim. The reliability assessment was performed on the composite model following the mathematical model defined for the reliability estimation.

Table 35. Claim Evidences Comments for C&E Case Study.

These two case studies have provided evidence and experiences allowing a critical discussion about the framework, models, and artifacts supporting the reliability assessment, as it is discussed in the next section.

8.4 Evaluating the Framework

This section discusses the results obtained after using the framework. Specific issues for some models and artifacts are also discussed.

8.4.1 Composition Model Evaluation

Theoretical vs Practical Approach. The theoretical rationale supporting the composition model is shown in the examples and case studies. From a theoretical perspective, the composition model is feasible and correct. From a practical perspective the composition model can be automated and extended to handle other architectural styles.

However, two issues in the composition process impact its applicability in real scenarios: The definition of the matching criteria and the characterization of input and output domains. These issues are at the heart of the composition mechanism. Case studies show that the determination of the matching criteria is not as trivial as the basic mathematical assembly example. There are cases where the definition of such criteria requires a considerable effort not only in the matching criterion definition, but also in the re-definition of the component domain characterization.

Architectural Style. The composition model only considers sequential component assemblies. This constraint reduces the scope of the framework applicability since other popular architectural styles are not included. However, this architectural style can be extended to handle other architectural styles such as procedure-call. This extension is easy from a theoretical point of view but it requires some additional monitoring infrastructure to be done in practice.

8.4.2 Prediction Model Evaluation

Theoretical vs Practical Approach. The theoretical rationale supporting the prediction is sustained by evidences found in the examples and case studies. The use of prediction models is a correct approach to address the problem of limited information. However, case studies shown the real complexity of such problem. This research used only two techniques to define prediction models. The selection of good prediction models requires a detailed analysis of every situation on a case by case basis and exploring distinct techniques. The practical use of different prediction models might require advanced training.

Limited Number of Techniques Explored. The number of prediction techniques explored in this dissertation includes just two statistical techniques since the goal was to illustrate the applicability of the framework rather than to perform an exhaustive research work on prediction models. Specifically, the goal was to show how to overcome the problem of having a limited amount of information.

BBMT Maturity. The Best Binary Matching Technique is an experimental technique that requires more research and analysis. Particularly, the practical interpretation of the error in prediction requires special attention. This technique, however, seems a promising approach since it can be automated and extended. The extension might include the results from a correlation analysis between attributes, the attribute importance, and the dependency among the attributes.

8.4.3 Reliability Model Evaluation

Assessment Interpretation. The reliability assessment provided by the reliability model is interpreted as a measure about the degree of certainty of a failure-free operation. The early assessment is important since it provides quantitative information during the early stages of the development process. This measure uses the information provided in the test profiles. Intuitively, a large number of test points will produce a better estimation (i.e., a smaller probability of failure).

The reliability value produced by the framework gives a pessimistic estimation of the probability of failure since this model assumes that having more information is better than having less information. For example, consider a component having some paths defined by exception handlers. It is possible that those execution nodes have a high probability of failure since they are rarely visited. This situation introduces test points having a considerable probability of failure and affecting the overall estimation.

Another issue in the reliability estimation is derived from the execution node probability of failure estimation. The information collected from execution traces do not allow to differentiate similar or identical execution node instances. This limitation comes from the assumption that all execution node instances are different since they are produced using distinct input parameters. This assumption might not hold in the general case. However, the reliability model can be extended to handle different assumptions to estimate the execution node probability of failure without modifying the other estimations (i.e., test case probability of failure, subdomain probability of failure, and domain probability of failure).

Comparison with Other Models. The comparison of reliability models is not an easy task. The heterogeneity and diversity of such models makes it difficult to establish a fair comparison. In addition, the real effectiveness of such models is closely connected with the testing strategy used to find failures.

The reliability model described in Chapter 7 can be classified as a combination of an architectural composite model and an input-domain model. The model follows an architectural composite approach since it combines the architectural information found in the execution traces and then it estimates the overall component reliability. In addition, this model follows the input domain approach since the execution node probabilities of failure are estimated considering strictly the number of successful runs. This combined approach makes this model a generalization of the input domain model defined by (Farr 1983). The input domain model represents an assembly that has only one component with one execution node.

Theoretical vs Practical Approach. The theoretical rationale supporting the reliability estimation is sustained by evidences found in the examples and case studies. The applicability of such model in real scenarios depends on the amount of information. Particularly, the point estimation does not require as much computational resources as the interval estimation. Case studies had a relatively large number of test cases and the corresponding estimation did not require high-powered computing resources.

Replication. The point estimation can be easily replicated based on the information from the assembly test profile. This replication is helpful in validating different implementations of the model. The interval estimation, however, is not suitable for being exactly replicated because the randomness in the Bootstrap sampling method. This issue can be addressed by using a larger bootstrap sample.

Relevance of the First Component in the Assembly. The test points from the first component in the assembly guides the test point composition. The use of these test cases, instead of predicting test cases, reduces the uncertainty in the estimation. However, using such points has an impact on the estimation since a small number of test points in the first component might impact the overall estimation even if the subsequent components in the assembly have a large number of test points.

8.4.4 Operational Information Evaluation

Notation. The notation proposed in this dissertation is compatible with industrial modeling techniques based on UML. This compatibility makes the notation easy to learn and to use. The critical part of such notation relies on the model construction and its consistency. The verification of consistency is critical when modeling constructs are extended to collect operational information. The proposed notation does not prescribe any language to model such consistency rules. However, there are tools that verify constraints of models based on OCL predicates (Gogolla et al., 2003).

Gathering Operational Information. This research does not address the issue of collecting operational information. The technology behind the framework can be used to collect such kind of information during field-testing. The monitoring infrastructure and the instrumentation technology can be adapted to collect other information such as the operations performed by the system. Using this approach, operational information can be collected and analyzed using the framework.

Integration to the Development Process. The task of defining operational models is suitable for being incorporated into a development process which uses architectural models. However, deriving assemblies for reliability assessment requires a deep understanding of the system requirements and the existence of a system architecture specification.

8.4.5 Component Test Profile Evaluation

Creation and Classification. Test Profile creation requires a considerable effort for collecting and organizing all the information in settings where the information is not properly organized. Particularly, the classification of test cases may need additional work when the domain is complex. Another issue is the instrumentation procedure that may require sophisticated tools requiring specialized training. However, these issues can be solved by incorporating such classification during the test case generation and developing tools to instrument the source code as a part of the development infrastructure.

Source of Information. The use of Component Test Profiles represents a feasible approach for collecting information into an organized artifact. The advantage of defining a metamodel for such artifact is that it allows the extension and tailoring for different needs. However, the success of using such an artifact requires having tools to manipulate, store, export, and analyze all the information contained in the component test profile.

Domain Characterization and Re-Characterization. Case studies showed some critical issues in the composition process. Particularly, the effort required to re-characterize an input domain to enable the composition process. This issue introduces a practical limitation to the automatic composition since it is required to have a compatibility analysis prior to performing the composition. However, this issue can be managed if the development organization has sufficient knowledge and experience in the application domain.

8.4.6 Framework Evaluation

Theoretical vs Practical Approach. The theoretical rationale supporting the framework feasibility is intuitively described by the examples and case studies. The framework performance is correct since it produces estimates according to the quality of the data contained in the component test profiles. The general application of the framework requires consideration of critical issues such as the composition process (specifically the definition of the matching condition), the selection of prediction models, the domain characterization for composition purposes, and data management. These issues might become more complex for larger assemblies.

Scalability. The framework is theoretically capable of assessing larger assemblies with complex components. However, the scalability of this framework is constrained by the critical issues previously mentioned. The evidences provided in case studies do not empirically support the assessment of larger assemblies.

Applicability and Scope. This framework is designed to reason about evidence contained in the component test profiles. The framework is not designed to assess designs or other forms of abstract specifications. The framework proposed in this dissertation is completely related with the testing results and the corresponding execution information. This framework can be used to assess any kind of software assembly that can be seen as a sequential assembly of components (e.g., embedded software, business software, or scientific software).

Data Quality. The framework does not incorporate mechanisms to analyze the data for properties such as freshness and randomness. Nevertheless, this kind of analyses can be easily incorporated but they are not covered in the framework described in this document.

8.5 Summary

This chapter has addresses the major claim: "*It is feasible to produce a meaningful (significant, useful) reliability assessment of software component assemblies, before their actual integration, based on (i) component abstract execution information collected in (ii) a* *limited number of previously executed component test cases and (iii) information about the expected assembly use.*” Two case studies were described and discussed. Specific issues for all models, some artifacts, and the framework were discussed.

The next chapter will present the conclusions and future research directions derived from the research presented in this dissertation.

Chapter 9

Conclusions and Future Research

This chapter provides a summary of the contributions described in this dissertation. Furthermore, future research directions based on this work are also presented.

9.1 Contributions

The framework addresses the problem of the early assembly reliability assessment by considering internal component execution information (i.e., execution traces) and the expected use of the system. This assessment uses a composite model produced from the component test profile composition process.

Specifically, the main contributions derived from this dissertation include:

1. A *novel approach* for the reliability assessment of software component assemblies based on partial information about the component testing process before the actual integration of the system. This approach is instantiated as a framework composed of artifacts, tasks, and models. This approach is unique since (i) it combines information from the component testing process to predict the composite execution behavior of such components (i.e., assembly), (ii) it assess the reliability of the assembly using execution information and the expected assembly usages, and (iii) it is a practical framework having tasks, artifacts, and models.
2. A *new artifact*, called the Component Test Profile. This artifact organizes and consolidates information obtained from the component software testing process. This artifact extends the "traditional" testing information to include execution-related evidences (i.e., execution traces).
3. A *strategy to monitor execution-related information* which uses a combination of source code instrumentation and a run-time monitoring infrastructure.
4. A *technique to characterize complex domains,* called the Domain Characterization Method. This technique uses tools from ontological engineering. The goal of this technique is to provide a domain characterization when the domain physical stored information does not provide enough information for differentiating data items for testing purposes.

5. A UML-based notation, referred to as the UML Operational Characterization Profile, designed to specify operational information. This notation is compliant with UML 2.0 and it extends the modeling capabilities of classifier constructs in UML.
6. A *formal composition model* based on component test profile information. This formalization defines a matching criterion between the domains from two connected component test profiles to characterize the composition mechanism.
7. A *strategy* to handle the limited amount of information based on *well-known prediction models*. The models used in this research include artificial neural networks and classification trees.
8. An experimental matching technique that is used when statistical prediction models do not provide good results. This technique abstracts test points as binary chains. The selection mechanism finds the most similar binary chain from a set of chains. This technique is called the *Best Binary Matching Technique*.
9. A *reliability prediction model* for software assemblies which combines execution trace information described in the component test profiles and the assembly expected usages described in the assembly operational profile. The model includes point and interval estimations.
10. A *set of artifact* templates used in all phases of the framework.
11. Software tools that partially automate the assessment process are described.

The early reliability assessments produced by the framework are important since they provide quantitative information during the early stages of the development process. This information can be used in several areas including test planning, component evaluation, and architecture evaluation.

The application of the framework opened new research directions that are described in the next section.

9.2 Future Research Directions

The Test Profile Analysis Framework is an extendible assessment framework. The extendibility is achieved by refining artifacts and models. For example, extending the framework to assess performance requires the incorporation of time-stamp monitoring mechanisms and the corresponding theoretical models supporting such analyses.

Future works will address specific research issues in different areas. These areas include development of more sophisticated models to predict software execution, to estimate the reliability, and to compose information derived from other architectural styles. Other research directions will address testing issues having a close relationship with reliability assessment. All these research areas are described in the following sections.

9.2.1 Software Execution Characterization

The approach to characterize the component execution considers only the different paths that an execution may follow. This technique does not consider the amount of code contained within an execution node; the criterion finds execution divergences within the source code. Future research will address this issue by investigating other approaches in static software analysis. The objective is to provide a better software characterization that can be used during reliability assessment. For example, the characterization may include the incorporation of the number of statements found in each execution node. This characterization can be used to define a better estimator for the execution node probability of failure.

These software characterization modifications may require changes in the structure of the models used in the framework, particularly the composition and the reliability models.

9.2.2 Software Execution Prediction

Predicting software execution is a major challenge. Future research will address the problem of finding prediction models that include not only the prediction of the execution node frequencies prediction but also the sequence prediction (i.e., considering the ordering of the nodes).

This research issue direction requires exploring more sophisticated prediction models and defining a more comprehensive methodology for the best prediction model selection. As it was described in Chapter 5, defining a universal prediction model may not be feasible given the heterogeneity of systems and the diversity of their corresponding attributes.

9.2.3 Software Reliability Assessment Considering Faults and Dependency

The reliability model presented in this research uses the frequencies obtained from the test point analysis and assumes the failure independence between execution nodes. Future research will consider failure dependencies among execution nodes. This consideration requires modifying the framework to include the prediction of such sequences as described in Section 9.2.2. This new assumption will also require modifying the reliability model with appropriate statistical models such as Markov Chains.

In addition, the current reliability model assumes that no failure is found in all test cases. In practice, the model will need to include failures. The incorporation of failures has to consider issues such as the failure pattern defined by the failure occurrence. This issue is difficult and it has been initially discussed in works debating the effectiveness of partition testing versus random testing (Chan et al., 1996) (Ntafos 2001) (Chen et al., 2004) (Chen et al., 2006) (Hamlet 1994) (Duran and Ntafos 1984) (Padilla et al., 2006). Some of these works proposes the existence of failure patterns. These kinds of patterns represent recurrent structural subsets defined over the system input domain.

9.2.4 Extend the Framework to Consider Other Architectural Styles

The architectural style considered in the framework is the sequential composition structure (i.e., data-flow architectures). This architectural style is suitable for small assemblies. However, other styles, such as the procedure-call and implicit invocation architectural styles (Shaw and Garlan 1996) are popular architectural styles. Future research will include investigation of methods for extending the framework to handle the procedure-call architect rural style.

The framework can be extended to handle the procedure-call architectural style. Nevertheless, there are some practical issues to consider such as the additional capabilities required by the monitoring infrastructure and the resources required to store the additional information in component test profiles. Changes in the composition model are not as complex as the infrastructure required to perform the composition. The composition model may require an extended trace substitution mechanism to abstract the procedure calls found in the execution traces. The matching criterion is similar to the one used for sequential assemblies.

9.2.5 Using the Framework for Assessing Other Properties

The framework has been used for reliability assessment. The entire infrastructure developed seems suitable for assessing other properties, such as performance, latency, and test coverage metrics. Future research will explore the assessment of other properties such as performance and test coverage metric.

Performance assessment requires the inclusion of performance models (e.g., timing and queue models) as well as the monitoring infrastructure modification to handle and collect performance time stamps. Test coverage assessment seems easy because of the execution-related information currently stored in component test profiles. However, it may be required to adapt the model for execution trace coverage. Examples of these models include execution graph and SDL models (Ellesberger et al., 1997).

9.2.6 Using Evolutionary Techniques to Find the Best Test Point Distribution

The use of evolutionary techniques for optimization problems has been used with considerable success in different domains (Goldberg 1989) (Larranaga and Lozano 2001) (Muñoz-Zavala et al., 2005). The advantage of such technique relies on its flexibility and the minimum assumptions required for the models. Evolutionary strategies are optimization techniques based on ideas of adaptation and evolution.

The framework assumes the execution diversity in execution traces. Future research will explore the use of evolutionary techniques to find the minimum set of test points characterizing the coverage execution. This characterization will be based on the criteria of source code coverage.

9.3 Final Remarks

Software Reliability is a critical attribute in several systems. From a theoretical point of view, the contributions presented in this dissertation augment the current theoretical knowledge about the software reliability assessment. From a pragmatic point of view, the contributions of this work provide solutions to practical problems, thereby helping software architects to asses the reliability of software component assemblies according to the expected usage of the assemblies.

Bibliography

Alhir, S. (1998) *UML in a Nutshell*, O'Reilly.

Allen, F. E. (1970) "Control Flow Analysis", In: *Proceedings of a Symposium on Compiler optimization*, Urbana-Champaign, Illinois 1-19.

Amyot, D., Logrippo, L. and Weiss, M. (2005) "Generation of Test Purposes from Use Case Maps", *Comput. Networks*, **49** (5) 643-660.

Anisimov, N. A., Golenkov, E. A. and Kharitonov, D. I. (2001) "Compositional Petri Net Approach to the Development of Concurrent and Distributed Systems", *Program. Comput. Softw.*, **27** (6) 309-319.

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., W252, st, J., rgen, Zettel, J. and rg (2002) *Component-Based Product Line Engineering with UML*, Addison-Wesley Longman Publishing Co., Inc.

Avizienis, A., Laprie, J.-C. and Randell, B. (2001) *Fundamental Concepts of Dependability*, University Of California - Computer Science Department.

Babar, M. A. and Gorton, I. (2004) "Comparison of Scenario-Based Software Architecture Evaluation Methods", In: *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)* 600-607.

Bengtsson, P. and Bosch, J. (1998) "Scenario-Based Software Architecture Reengineering", In: *5th International Conference on Software Reuse (ICSR 98)* 308-319.

Bengtsson, P. and Bosch, J. (1999) "Architecture Level Prediction of Software Maintenance", In: *Proceedings of the Third European Conference on Software Maintenance and Reengineering* 139-147.

Bishop, P. G. (2002) "Rescaling Reliability Bounds for a New Operational Profile", In: *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* 180-190.

Bond, W. P., Al-Otaiby, T. N. and Alsharif, M. N. (2004) "Software Architecture Assessment Using Quality Function Deployment", In: *Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, Austria 65-70.

Bondarev, E., Chaudron, M. and de With, P. (2006) "A Process for Resolving Performance Trade-Offs in Component-Based Architectures", In: *Proceedings of the 9th International Symposium on Component-Based Software Engineering (CBSE)* 254-269.

Bousquet, L. D., Ouabdesselam, F. and Richier, J. (1998) "Expressing and Implementing Operational Profiles for Reactive Software Validation", In: *Proceedings of the 9th International Symposium on Software Reliability Engineering* 222-230.

Brown, J. R. and Lipow, M. (1975) "Testing for Software Reliability", In: *Proceedings of the International Conference on Reliable Software*, Los Angeles, California 518-527.

Cai, K. Y., Li, Y. C. and Liu, K. (2004) "Optimal and Adaptive Testing for Software Reliability Assessment", *Information and Software Technology*, **46** (15) 989-1000.

Cai, K. Y., Li, Y. C. and Ning, W. Y. (2005) "Optimal Software Testing in the Setting of Controlled Markov Chains", *European Journal of Operational Research*, **162** (2) 552-579.

Chaki, S. (2006) *SAT-Based Software Certification*, Software Engineering Institute - Carnegie Mellon University.

Chaki, S. and Nishant, S. (2006) *Assume-Guarantee Reasoning for Deadlock*, Software Engineering Institute - Carnegie Mellon University.

Chan, F., Chen, T., Mak, I. and Yu, Y. T. (1996) "Proportional Sampling Strategy: Guidelines for Software Testing Practitioners", *Information and Software Technology*, **38** (12) 775-782.

Chen, T. Y., Kuo, F. C. and Merkel, R. (2006) "On The Statistical Properties of Testing Effectiveness Measures", *Journal of Systems and Software*, **79** (5) 591-601.

Chen, T. Y., Kuo,Fei-Ching and Merkel, R. (2004) "On The Statistical Properties of the F-Measure", In: *Fourth International Conference on Quality Software* 146-153

Chen, T. Y. and Yu, Y. T. (1996) "On the Expected Number of Failures Detected by Subdomain Testing and Random Testing", *IEEE Transactions on Software Engineering*, **22** (2) 109- 119.

Chen, T. Y. and Yu, Y. T. (2000) "The Universal Safeness of Test Allocation Strategies for Partition Testing", *Information Sciences*, **129** (1-4) 105- 118.

Cheng, R. C. H. (1995) "Bootstrap Methods in Computer Simulation Experiments", In: *Proceedings of the 27th Conference on Winter Simulation* (*WSC '95)* 171-177.

Cheung, R. C. (1980) "A User-Oriented Software Reliability Model", *IEEE Transactions on Software Engineering*, **6** (2) 118-125.

Clark, L. and Pregibon, D. (1991) "Tree Based Models", In: Chambers, J.M. and Hastie, T.J. (ed.). *Statistical Models in S* CRC Press, Inc. 377-419.

Clements, P., Kazman, R. and Klein, M. (2002) *Evaluating Software Architectures: Methods and Case Studies*, Pearson Education Limited, Indianapolis, IN.

Corcoran, W. J., Weingarten, H. and Zehna, P. (1964) "Estimating Reliability after Corrective Action", *Management Science*, **10** (4) 786-795.

Cortellessa, V., Singh, H. and Cukic, B. (2002) "Early Reliability Assessment of UML based Software Models", In: *3rd international Workshop on Software and Performance (WOSP02)*, Rome, Italy 302-309.

CRAN (2006) *The Comprehensive R Archive Network* http://cran.r-project.org/

Crnkovic, I., Larsson, S. and Stafford, Judith (2002) "Component-Based Software Engineering: Building Systems from Components", *Software Engineering Notes*, **(27)3**.

Davila-Nicanor, L. and Mejia-Alvarez, P. (2004) "Reliability Improvement of Web-Based Software Applications", In: *QSIC '04: Proceedings of the Quality Software, Fourth International Conference on (QSIC'04)* 180-188.

Denise, A., Gaudel and Gouraud, S. (2004) "A Generic Method for Statistical Testing", In: *15th. IEEE International Symposium on Software Reliability Engineering*, Saint-Malo, Bretagne, France 25-34.

Dobrica, L. and Niemela, E. (2002) "A Survey on Software Architecture Analysis Methods", *IEEE Transactions on Software Engineering*, **28** (7) 638-653.

Dolbec, J. and Shepard, T. (1995) "A Component Based Software Reliability Model", In: *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada 19-28.

Duda, R. O., Hart, P. E. and Stork, D. G. (2000) *Pattern Classification*, Wiley-Interscience.

Dulz, W. and Zhen, F. (2004) "MaTeLo - Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains and TTCN-3", In: *Third International Conference on Quality Software*, Dallas, Texas 336 - 346.

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

Ellesberger, J., Hogrefe, D. and Sarma, A. (1997) *SDL: Formal Object-oriented Language for Communicating Systems*, Prentice Hall.

Elliot, R., Marchbank, M. P., McWilliams, M., Ringer, L. and Simmons, D. (1978) "Measuring Computer Software Reliability", *Computers and Industrial Engineering*, **2** (3) 141-151.

Everett, W. W. (1992) "An Extended Execution Time Software Reliability Model", In: *Proceedings of Third International Symposium on the Software Reliability Engineering* 4-13.

Everett, W. W. (1999) "Software Component Reliability Analysis", In: *IEEE Symposium on Application - Specific Systems and Software Engineering and Technology* 204-211.

Farr, W. H. (1983) *A Survey of Software Reliability Modeling and Estimation*, Naval Surface Warfare Center.

Feiler, P. H., Gluch, D. P., Hudak, J. J. and Lewis, B. A. (2003) *Embedded Systems Architecture Analysis Using SAE AADL*, Software Engineering Institute - Carnegie Mellon University.

Feiler, P. H., Gluch, D. P. and Hudak, J. J. (2006) *The Architecture Analysis & Design Language (AADL): An Introduction*, Software Engineering Institute - Carnegie Mellon University.

Fenton, N. E. and Pfleeger, S. (1998) *Software Metrics: A Rigorous and Practical Approach*, Course Technology.

Fernandez-Lopez, M., Gomez-Perez, A. and Juristo, N. (1997) "METHONTOLOGY: From Ontological Art Towards Ontological Engineering", In: *Spring Symposium on Ontological Engineering of AAAI*, Stanford University 33-40.

Folmer, E., van Gurp, J. and Bosh, J. (2003) "Scenario-based Assessment of Software Architecture Usability", In: *Workshop on Bridging the Gaps between Software Engineering and Human-Computer Interaction, ICSE*, Portland, USA.

Forgacs, I. (1996) "An Exact Array Reference Analysis for Data Flow Testing", In: *Proceedings of the 18th International Conference on Software Engineering* (*ICSE '96*) 565-574

Foundation, T. E. (2007) *Eclipse Java Development Tools (JDT) Subproject* http://www.eclipse.org/jdt/

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999) *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional.

Friedrichsen, L. (2003) *Microsoft Access 2003: Illustrated Brief*, Course Technology Press.

Froberg, Wallin, P. and Axelsson, J. (2006) "Towards Quality Assessment in Integration of Automotive Software and Electronics: An ATAM approach", In: *Proceedings of the 6th Conference on Software Engineering and Practice in Sweden*, Umea, Sweden.

Gaffney, J. and Davis, C. (1988) "An Approach to Estimating Software Errors and Availability", In: *11th Minnowbrook Workshop on Software Reliability.*

Gaffney, J. and Pietrolewicz, J. (1990) "An Automated Model for Software Early Prediction (SWEEP)", In: *13th Minnowbrook Workshop on Software Reliability.*

Gao, C., Liu, R., Song, Y. and Chen, H. (2006) "A Model Checking Tool Embedded into Services Composition Environment", In: *GCC '06: Proceedings of the Fifth International Conference on Grid and Cooperative Computing (GCC'06)* 355-362.

Geiger, K. (1995) *Inside ODBC*, Microsoft Press.

Gittens, M., Lutfiyya, H. and Bauer, M. (2004) "An Extended Operational Profile Model", In: *Eighth International Symposium on Software Reliability Engineering (ISSRE)*, Mont Saint-Michel, Bretagne, France 314-325.

Goel, A. L. and Okumoto, K. (1979) "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures", *IEEE Transactions on Reliability*, **28** 206-211.

Gogolla, M., Richters, M. and Bohling, J. (2003) "Tool Support for Validating UML and OCL Models through Automatic Snapshot Generation", In: *SAICSIT '03: Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology* 248-257.

Gokhale, S., Wong, W., Trivedi, K. and Horgan, J. (1998) "An Analytical Approach to Architecture-Based Software Reliability Prediction", In: *International Symposium on Computer Performance and Dependability (IPDS)*, Durham, North Carolina, USA 13-22.

Gokhale, S. S. and Trivedi, K. S. (2002) "Reliability Prediction and Sensitivity Analysis Based on Software Architecture", In: *13th International Symposium on Software Reliability Engineering (ISSRE'02)* 64-75.

Gokhale, S. and Trivedi, K. (1997) "Structure-Based Software Reliability Prediction", In: *Fifth Intl. Conference on Advanced Computing*, Chenai, India 447-452.

Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc.

Gomez-Perez, A., Corcho-Garcia, O. and Fernandez-Lopez, M. (2003) *Ontological Engineering*, Springer-Verlag New York, Inc.

Goseva-Popstojanova, K., Hamill, M. and Perugupalli, R. (2005) "Large Empirical Case Study of Architecture-based Software Reliability", In: *16th International Symposium on Software Reliability Engineering (ISSRE 05)*, Chicago, IL, USA 43-52.

Goseva-Popstojanova, K. and Kamavaram, S. (2003) "Assessing Uncertainty in Reliability of Component-Based Software Systems", In: *14th International Symposium on Software Reliability Engineering* 307-320.

Goseva-Popstojanova, K. and Trivedi, K. (2003) "Architecture-Based Approaches to Software Reliability Prediction", *IEEE Transactions on Software Engineering*, **46** (7) 1023-1036.

Grassi, V., Mirandola, R. and Sabetta, A. (2006) "A Model Transformation Approach for the Early Performance and Reliability Analysis of Component-Based Systems", In: *9th International Symposium on Component-Based Software Engineering (CBSE)* 270-284.

Grgic, M. and Delac, K. (2007) *Face Recognition DataBases* http://www.face-rec.org/databases/

Gruninger, M. and Fox, M. (1995) "Methodology for the Design and Evaluation of Ontologies", In: *Workshop on Basic Ontological Issues in Knowledge Sharing* 6.1-6.10.

Grunske, L., Kaiser, B. and Papadopolous, Y. (2005) "Model-Driven Safety Evaluation with State-Event-Based Component Failure Annotations", In: *8th International Symposium on Component-Based Software Engineering (CBSE 05)*, St. Louis, MO, USA 33-49.

Hairston, M., Ruszkiewicz, J. and Friend, C. (1999) *The Scott, Foresman Handbook for Writers*, Longman.

Hamlet, D., Mason, D. and Woit, D. (2001) "Theory of Software Reliability Based on Components", In: *23rd International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada 361-370.

Hamlet, D., Mason, D. and Woit, D. (2004) "Properties of Software Systems Synthetized from Components", In: Lau, K.-K. (ed.). *Component-Based Software Development: Case Studies* World Scientific Publishing Company 129-58.

Hamlet, D. and Taylor, R. (1990) "Partition Testing does not Inspire Confidence", *IEEE Transactions on Software Engineering*, **16** (12) 1402-1411.

Hamlet, R. (1994) "Random Testing", In: J.Marciniak (ed.). *Encyclopedia of Software Engineering* Wiley 970-978.

Hecht, H. (1977) *Measurement, Estimation, and Prediction of Software Reliability*, National Aeronautics and Space Administration.

Hibbeler, R. C. (2001) *Structural Analysis*, Prentice-Hall.

Hissam, S., Hudak, J., Ivers, J., Klein, M., Larsson, M., Moreno, G., Northrop, L., Plakosh, D., Stafford, J., Wallnau, K. and Wood, W. (2002a) *Predictable Assembly of Substation Automation Systems: An Experiment Report*, Software Engineering Institute.

Hissam, S., Klein, M., Lehoczky, J., Merson, P., Moreno, G. and Wallnau, K. (2004) *Performance Property Theories for Predictable Assembly from Certifiable Components (PACC)*, Software Engineering Institute - Carnegie Mellon University.

Hissam, S., Moreno, G., Stafford, J. and Wallnau,Kurt (2002b) "Packaging Predictable Assembly", In: *First International IFIP/ACM Working Conference on Component Deployment*, Berlin, Germany.

Hissam, S., Moreno, G. and Wallnau, K. (2005) *Using Containers to Enforce Smart Constraints for Performance in Industrial Systems*, Software Engineering Institute - Carnegie Mellon University.

Hissam, S. and Ivers, J. (2002) *PECT Infraestructure: A Rough Sketch*, Software Egineering Institute.

Hoffman, D. (1998) *A Taxonomy for Test Oracles* http://www.softwarequalitymethods.com/Papers/OracleTax.pdf.

Hoyland, A. and Rausand, M. (2003) *System Reliability Theory: Models and Statistical Methods*, John Wiley & Sons, 2nd.

Hughes, R. (2002) *Practical Software Measurement*, Addison-Wesley Longman Publishing Co., Inc.

Hui-Qun, Z., Jing, S. and Yuan, G. (2001) "A New Method for Estimating the Reliability of Software System Based on Components", In: *International Symposium on Software Reliability Engineering*, Hong Kong

Humphrey, W. S. (1994) *A Discipline for Software Engineering: The Complete PSP Book*, Addison-Wesley.

Humphrey, W. S. (2002) *Winning with Software: An Executive Strategy*, Addison-Wesley, Boston, MA.

IEEE (1998) "IEEE Std. 610.12-1990. Standard Glossary of Software Engineering Terminology", *IEEE*.

Ivers, J. and Sharygina, N. (2004) *Overview of ComFoRT: A Model Checking Reasoning Framework*, Software Engineering Institute - Carnegie Mellon University.

Jacobson, I., Booch, G. and Rumbaugh, J. (1998) *The Unified Modeling Language*, Addison-Wesley.

Jelinski, Z. and Moranda, P. (1972) "Software Reliability Research", In: *Statistical Computer Performance Evaluation*, New York: Academic Press 465-84.

Jorgensen, P. C. (1995) *Software Testing: A Craftsman's Approach*, CRC Press.

Kazman, R., Abowd, G., Bass, L. and Clements, P. (1996) “Scenario-Based Analysis of Software Architecture” In: *IEEE Software*, Vol. **13**, pp. 47-55.

Kazman, R., Bass, L., Abowd, G. and Webb, M. (1994) "SAAM: A Method for Analyzing the Properties of Software Architectures", *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, 81-90.

Kazman, R., Klein, M. and Clements, P. (2004) *ATAM: Method for Architecture Evaluation*, Software Engineering Institute - Carnegie Mellon University.

Klein, M. and Kazman, R. (1999) *Attribute-Based Architectural Styles*, Software Engineering Institute - Carnegie Mellon University.

Komi-Sirvio, S., Rus, I. and Costa, P. (2003) *Models for Software Dependability Properties Assessment and Estimation*, Fraunhofer Center for Experimental Software Engineering.

Krishnamurthy, S. and Mathur, A. P. (1997) "On the Estimation of Reliability of a Software System Using Reliability of its Components", In: *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE '97)* 146-155.

Kubat, P. (1989) "Assessing Reliability of Modular Software", *Operations Research Letters*, **8** 35-41.

Kuhn, T. and Thomann, O. (2007) *Abstract Syntax Tree* http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html

Langberg, N. and Singpurwalla, N. D. (1985) "A Unification of Some Reliability Models", *SIAM Journal on Scientific Computing*, **6** (3) 781-790.

LaPadula, L. (1975) *Engineering of Quality Software Systems, Volume VIII - Software Reliability Mdeling and Measurement Techniques*, Rome Rome Air Development Center.

Laprie, J.-C. and Kanoun, K. (1996) "Software Reliability and System Reliability", In: . *Handbook of Software Reliability and System Reliability*, Hightstown, NJ: McGraw-Hill 27-69.

Larranaga, P. and Lozano, J. A. (2001) *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*, Springer.

Larsson, M. (2004) *Predicting Quality Attributes in Component-based Software Systems. Department of Computer Science and Engineering*-162. PhD Thesis. Cranfield University, School of Management.

Lassing, N., Rijsenbrij, D. and Vliet, H. v. (2001) "Using UML in Architecture-Level Modifiability Analysis", In: *1st ICSE Workshop on Describing Software Architecture with UML*, Toronto, Canada.

Le Guen, H., Marie, R. and Thelin,Thomas (2004) "Reliability Estimation for Statistical Usage Testing using Markov Chains", In: *15th International Symposium on Software Reliability Engineering (ISSRE'04)*, Saint-Malo, Bretagne, France 54-65.

Lenat, D. and Guha, R. (1990) *Building Large Knowledge-based Systems: Representation and Inference in the Cyc Project*, Addison-Wesley, Boston.

Littlewood, B. (1975) "A Reliability Model for Systems with Markov Structure", *Applied Statistics*, **24** (2) 172-177.

Littlewood, B. (1985) "Software Reliability Model for Modular Program Structure", *IEEE Transactions on Reliability*, **28** (3) 241-246.

Littlewood, B. and Strigini, L. (2000) "Software Reliability and Dependability: A Roadmap", In: *International Conference on Software Engineering*, Limerick, Ireland 175-188.

Littlewood, B. and Verral, J. L. (1973) "A Bayesian Reliability Growth Model for Computer Software", *Applied Statistics*, **22** (3) 332-346.

Liu,Yan and Gorton, I. (2005) "Performance Prediction of J2EE Applications using Messaging Protocols", In: *Proceedings of the 8th International Symposium on Component-Based Software Engineering (CBSE)* 1-16.

Lüttgen, G., Beeck, M. v. d. and Cleaveland, R. (2000) "A Compositional Approach to Statecharts Semantics", In: *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, San Diego, California, United States 120-129.

Lyu, M. R. (1996) *Handbook of Software Reliability Engineering*, McGraw-Hill.

Maindonald, J. and Braun, J. (2006) *Data Analysis and Graphics Using R: An Example-based Approach (Cambridge Series in Statistical and Probabilistic Mathematics)*, Cambridge University Press.

Mason, D. (2002) *Probabilistic Program Analysis for Software Component Reliability. Computer Science*. PhD Thesis. Cranfield University, School of Management.

Mason, D. (2002) *Probabilistic Program Analysis for Software Component Reliability*. PhD Thesis. Cranfield University, School of Management.

May, J. H. R. (2002) "Testing the Reliability of Component-Based Safety Critical Software", In: *20th International System Safety Conference* 214-224.

McGregor, J., Stafford, J. A. and Cho, I.-H. (2003) "Measuring Component Reliability", In: *6th ICSE Workshop on Component-Based Software Engineering*, Portland, Oregon, USA.

Miller, K. W., Morell, L. J., Noonan, R. E., Park, S. K., Nicol, D. M., Murrill, B. W. and Voas, M. (1992) "Estimating the Probability of Failure when Testing Reveals no Failures", *Software Engineering, IEEE Transactions on*, **18** (1) 33-43.

Muñoz-Zavala, A. E., Villa-Diharce, E. R. and Hernandez-Aguirre, A. (2005) "Particle Evolutionary Swarm for Design Reliability Optimization", In: *Third International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, Guanajuato, Guanajuato, Mexico 856-869.

Musa, J. (1971) "A Theory of Software Reliability and Its Applications", *IEEE Trans. Softw. Eng.*, **1** (3) 312-327.

Musa, J. D. (1993) "Operational Profiles in Software-Reliability Engineering", *IEEE Software*, **10** (2) 14-32.

Musa, J. D. (2004a) "Chapter 1. Problem, Process and Product", In: *Software Reliability Engineering: More Reliable Software Faster And Cheaper* Authorhouse.

Musa, J. D. (2004b) *Software Reliability Engineering: More Reliable Software Faster And Cheaper*, Authorhouse.

Narayanan, S. and McIlraith, S. A. (2002) "Simulation, Verification and Automated Composition of Web Services", In: *WWW '02: Proceedings of the 11th international conference on World Wide Web* 77-88.

Nelson, E. (1978) "Estimating Software Reliability from Test Data", In: *Microelectronics and Reliability*, Pergamon, NY 67-74.

Nicol, N. and Albrecht, R. (2002) *Programming Microsoft SQL Server 2000 for Microsoft Access Developers*, Microsoft Press.

Nielson, F., Nielson, H. R. and Hankin, C. (2006) *Principles of Program Analysis*, Kluwer Academic Publishers.

Nilsson, J. W. and Riedel, S. (2004) *Electric Circuits*, Prentice Hall.

Nord, R. L., Barbacci, M., Clements, P., Kazman, R., Klein, M., O'Brien, L. and Tomayko, J. E. (2003) *Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM)*, Software Engineering Institute.

Ntafos, S. C. (2001) "On Comparisons of Random, Partition, And Proportional Partition Testing", *Software Engineering, IEEE Transactions on*, **27** (10) 949-960.

OMG (2002) *OMG Unified Modeling Language Specification Version 1.5*, Object Management Group, Inc.

OMG (2005a) *UML Profile for Schedulability, Performance and Time. Version 1.1.*

OMG (2005b) *UML Testing Profile*.

OMG (2006a) *Object Constraint Language (OCL). Version 2.0*.

OMG (2006b) *UML Profile for Modeling QoS and Fault Tolerance Characteristics and Mechanisms Version 1.0*.

OMG (2006c) *UML Profile for System on a Chip (SoC). Version 1.0.1*.

OMG (2007) *UML 2.1.1 Superstructure Specification*.

Ofelt, D. and Hennessy, J. L. (2000) "Efficient Performance Prediction for Modern Microprocessors", In: *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* 229-239.

Ohba, M. (1984) "Software Reliability Analysis Models", *IBM Journa of Research and Development*, **21** (4) 428-443.

Padilla, G., Montes de Oca, C., -. Y. and Bastani, F. (2006) "Weighted Proportional Sampling : A Generalization for Sampling Strategies in Software Testing ", *Electrical and Electronics Engineering, 2006 3rd International Conference on*, 1 - 4.

Palsberg, J. (2001) "Type-based Analysis and Applications", In: *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* 20-27.

Popic, P., Desovski, D., Abdelmoez, W. and Cukic, B. (2005) "Error Propagation in the Reliability Analysis of Component based Systems", In: *16th International Symposium on Software Reliability Engineering (ISSRE 05)*, Chicago, IL, USA 53-62.

Pressman, R. S. (2005) *Software Engineering: A Practitioner's Approach*, McGraw-Hill.

Reps, T., Ball, T., Das, M. and Larus, J. (1997) "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem", *SIGSOFT Softw. Eng. Notes*, **22** (6) 432-449.

Reussner, R. H., Schmidt, H. W. and Poernomo, I. H. (2003) "Reliability Prediction For Component-Based Software Architectures", *The Journal of Systems and Software*, **66** (3) 241-252.

Rodrigues, G., Rosenblum, D. and Uchitel, S. (2005) "Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems", In: *8th International Conference Fundamental Approaches to Software Engineering (FASE 05)* 111-126.

Rome-Laboratory (1992) *Methodology for Software Reliability Prediction and Assessment Volumes 1 and 2*, Rome Laboratory.

Roshandel, R. and Medvidovic, N. (2004) "Toward Architecture-Based Reliability Estimation", In: *Twin Workshops on Architecting Dependable Systems (WADS 2004)*, Edinburgh, UK.

Runeson, P. and Regnell, B. (1998) "Derivation of an Integrated Operational Profile and Use Case Model", In: *9th International Symposium on Software Reliability Engineering (ISSRE'98)*, Paderborn, Germany.

Sandberg, C., Ermedahl, A., Gustafsson, J. and Lisper, B. (2006) "Faster WCET flow analysis by program slicing", In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers and Tool Support for Embedded Systems*, Ottawa, Ontario, Canada 103-112.

Sayre, K. and Poore, J. . H. (2000) "Stopping Criteria for Statistical Testing", *Information and Software Technology*, **42** (12) 851-857.

Schick, G. J. and Wolverton, R. (1978) "Assessment of Software Reliability", In: *Operations Research*, Vienna 395-422.

Schneidewind, N. F. (1975) "Analysis of Error Processes in Computer Software", *Sigplan Note*, **10** (6) 337-346.

SEI (2007a) *Performance-Critical Systems (PCS) Initiative* http://www.sei.cmu.edu/pcs/

SEI (2007b) *Predictable Assembly from Certifiable Components (PACC) Initiative* http://www.sei.cmu.edu/pacc/

SEI (2007c) *Software Architecture for Software-Intensive Systems (SAT) Initiative* http://www.sei.cmu.edu/architecture/index.html

Shaw, M. and Garlan, D. (1996) *Software Architectures: Perspectives on an Emerging Discipline*, Prentice Hall.

Shelemyahu, Z. and Kenett, o. S. (1998) *Estadística Industrial Moderna. Diseño y Control de la Calidad y Confiabilidad*, International Thomson Editores.

Shooman, M. L. (1976) "Structural Models for Software Reliability Prediction", In: *2nd International Conference on Software Engineering*, San Francisco, California, United States 268-280.

Shukla, R., Carrington, D. and Strooper, P. (2004a) "Systematic Operational Profile Development for Software Components", *Software Engineering Conference, 2004. 11th Asia-Pacific*, 528-537.

Shukla, Strooper, P. and Carrington, D. (2004b) "A Framework for Reliability Assessment of Software Components", In: *7th International Symposium on Component-Based Software Engineering*, Edinburgh, UK 272-279.

Singh, H., Cortellessa, V., Cukic, B., Gunel, B. and Bharadwaj, V. (ed.) (2001) *A Bayesian Approach to Reliability Prediction and Assessment of Component Based Systems*, . 12th International Symposium on Software Reliability Engineering (ISSRE) 12-21Honk Kong: IEEE.

Singpurwalla, N. D. and Wilson, S. P. (1999) *Statistical Methods in Software Engineering: Reliability and Risk*, Springer.

Smidts, C., Sova, D. and Mandela, G. K. (1997) "An Architectural Model for Software Reliability Quantification", In: *8th International Symposium On Software Reliability Engineering* 324-335.

Smidts, C. and Sova, D. (1999) "An Architectural Model for Software Reliability Quantification: Sources of Data", *Reliability Engineering & System Safety*, **64** (2) 279-290.

Smidts, C., Stoddard,R. and Stutzke,M. (1996) "Software Reliability Models: An Approach to Early Reliability Prediction", In: *Seventh International Symposium on Software Reliability Engineering (ISSRE '96)* 132-141.

Srikant, Y. and Shankar, P. (2002) *The Compiler Design Handbook: Optimizations & Machine Code Generation*, CRC.

Stafford,Judith, Wallnau, K., Moreno, G. A. and Hissam, S. A. (2001) *Packaging Predictable Assembly with Prediction-Enabled Component Technology*, Software Engineering Institute.

Stutzke, R. D. (2005) *Estimating Software-Intensive Systems: Projects, Products, and Processes*, Addison-Wesley Professional.

Sugiura, N., Yamamoto, M. and Shiino, T. (1974) "On the Software Reliability", *Microelectronics and Reliability*, **13** 529-533.

Sukert, A. (1976) *A Software Reliability Modeling Study*, Rome Air Development Center.

Svahnberg, M. and Wohlin, C. (2005) "An Investigation of a Method for Identifying a Software Architecture Candidate with Respect to Quality Attributes", *Empirical Software. Engineering*, **10** (2) 149-181.

Team, C. P. (2006) *CMMI for Development Version 1.2*, Software Engineering Institute - Carnegie Mellon University.

Thayer, A. , Craig, R. , Frey, E. , Hetrick, L. and Lipow, (1976) *Software Reliability Study*, Rome Air Development Center.

Thesing, S. (2006) "Modeling a System Controller for Timing Analysis", In: *Proceedings of the 6th ACM IEEE International conference on Embedded software*, Seoul, Korea 292-300.

Thevenod-Fosse, P. and Waeselynck, H. (1991) "An Investigation of Software Statistical Testing", *Journal of Software Testing, Verification and Reliability*, **1** (2) 5-26.

Tian, Jeff (1995) "Integrating Time Domain and Input Domain Analyses of Software Reliability Using Tree-Based Models", *IEEE Transactions on Software Engineering*, **21**.

Uschold, M. and Grüninger, M. (1996) "Ontologies: Principles, Methods, and Applications", *Knowledge Engineering Review*, **11** (2) 93-155.

Utting, M., Pretschner, A. and Legeard, B. (2006) *A Taxonomy of Model-Based Testing*, The University of Waikato. Report.

Voas, J. (2000) "Will the Real Operational Profile Please Stand Up?" In *IEEE Software*, Vol. **17**, pp. 87-89.

Voas, J. (2003) "Predicting System Trustworthiness", In: Crnkovic, I. and Larsson, M. (ed.). *Building Reliable Component-Based Software Systems* Artech House 193-203.

Wallnau, K. (2003) *Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC)*, Software Engineering Institute.

Wallnau, K., Hissam, S. A. and Seacord, R. C. (2002a) *Building Systems from Commerical Components*, Addison-Wesley Longman Publishing Co., Inc.

Wallnau, K., Ivers, J. and Nishant, S. (2002b) *A Basis for Composition Language CL*, Software Egineering Institute.

Wallnau, K. C. and Ivers, J. (2003) *Snapshot of CCL: A Language for Predictable Assembly*, Software Engineering Institute.

Wang, W.-L., Wu, Y. and Chen, M.-H. (1999) "An Architecture-Based Software Reliability Model", In: *1999 Pacific Rim International Symposium on Dependable Computing* 143-150

Weinstock, C. B., Goodenough, J. B. and Hudak, J. J. (2004) *Dependability Cases*, Software Engineering Institute - Carnegie Mellon University.

Weinstock, C. B. and Goodenough, J. B. (2006) *On System Scalability*, Software Engineering Institute - Carnegie Mellon University.

Whittaker, J. A. and Thomason, M. G. (1994) "A Markov chain model for statistical software testing", *Software Engineering, IEEE Transactions on*, **20** (10) 812-824.

Whittaker, J. A. and Voas, J. (2000) "Toward a More Reliable Theory of Software Reliability" In *IEEE Computer*, Vol. **33**, pp. 36-46.

Williams, L. G. and Smith, C. U. (2002) "PASASM: A Method for the Performance. Assessment of Software Architectures", In: *3rd Workshop on Software Performance*, Rome, Italy.

Woit, D. (1993) "Specifying Operational Profiles for Modules", In: *ACM International Symposium on Software Testing and Analysis - ISSTA* 2-10.

Xiaoguang, M. and Yongjin, D. (2003) "A General Model for Component-Based Software Reliability", In: *29th Euromicro Conference (EUROMICRO'03)* 395-398

Xie, M. and Wohlin, C. (1995) "An Additive Reliability Model for the Analysis of Modular Software Failure Data", In: *Proceedings of the 6th International Symposium on Software Reliability Engineering* **188** - 194.

XLMiner (2007) *Classification Tree. Help Reference.* http://www.resample.com/xlminer/help/

Yacoub, S. M., Cukic, B. and Ammar, H. H. (2004) "A Scenario-Based Reliability Analysis Approach for Component-Based Software", *IEEE Transactions on Reliability*, **53** (4) 465-480.

Yujian, F. u., Dong, Z. and Xudong, H. e. (2006) "Modeling, Validating and Automating Composition of Web Services", In: *Proceedings of the 6th International Conference on Web Engineering*, Palo Alto, California, USA 217-224.

Zhu, Y. and Gao, J. (2003) "A Method to Calculate the Reliability of Component-Based Software", In: *12th Asian Test Symposium (ATS'03)* 488-491.

Appendixes

This section describes additional information about the research presented in this dissertation. Appendix A provides a glossary with the most important concepts used in this document. Appendix B describes a set of templates for the most important artifacts defined in the Test Profile Analysis Framework. Finally, appendix C provides the first version of the formal UML profile designed to characterize operational information.

A. Glossary

The symbol * denote definitions taken from the IEEE Standard Glossary of Software Engineering (IEEE 1998).

Term	**Definition**
Abstract Execution Graph	Graph representing the program source code control-flow structure represented by execution nodes.
Artifact	An artifact is one of many kinds of tangible byproduct produced during the development of software. Examples of artifacts include class diagram, architecture descriptions, test plans, software requirements specification, etc.
Assembly	An assembly is an arrangement of two or more components providing a complex functionality. This functionality is produced by the composition of all component functionalities.
Component	A component is a reusable piece of functionality characterized by its input domain and its output domain. In this context, a function or procedure is a component.
Data Constraint	A specification about the allowable ranges defined for data attribute and its quantitative use distribution. This specification is composed of two main elements: the data description and the data distribution. The former specifies the nature and the allowable values defined for the corresponding data attribute, and the latter specifies the probabilistic distribution of such allowable values (ranges, or partitions).
Data-flow Architectural style	Configuration of elements (processes, modules, components, etc.), arranged so that the output of each element is the input of the next.
Data-flow*	The sequence in which data transfer, use, and transformation are performed during the execution of a computer program.
Domain	A domain is a set of all possible values that a set may have. When this concept is related to a software function, the input domain is the set of all possible values that the function may use as input. The output domain is the set of all possible values that such function may generate as output.
Environmental Constraint	A constraint about the environment where the system is under operation. These constraints include hardware, software, run-time environments, user-defined characteristics, etc.
Execution Node	An execution node represents a grouping of source code statements based on performing a control flow analysis.
Execution Time*	The amount of elapsed time or processor time used in executing a computer program.

Execution Trace	(1)* A record of the sequence of instructions executed during the execution of a computer program. Often takes the form of a list of code labels encountered as the program executes. (2) A sequence of execution nodes.
Failure Rate*	The ratio of the number of failures of a given category to a given unit of measure; for example, failures per unit of time, failures per number of transactions.
Failure*	The inability of a system or component to perform its required functions within specified performance requirements.
Fault*	An incorrect step, process, or data definition in a computer program. Usually the terms “bug” and “error” are used indistinctly.
Formal Specification*	(1) A specification written and approved in accordance with established standards. (2) A specification written in a formal notation, often used in proof of correctness.
Function	Atomic functionality performed by a system. The combination of one or more functions defines an operation.
Instrumentation*	Devices or instructions installed or inserted into hardware or software to monitor the operation of a system component.
Latency*	The time interval between the instant at which an instruction control unit issues a call for data and the instant at which the transfer of data is started.
Meta-modeling	Meta-modeling is the construction of a collection of "concepts" (things, terms, etc.) within a certain domain. A model is an abstraction of phenomena in the real world, and a metamodel is yet another abstraction, highlighting properties of the model itself. This model is said to conform to its metamodel like a program conforms to the grammar of the programming language in which it is written
Metric*	A quantitative measure of the degree to which a system, component, or process possesses a given attribute.
Model	A model is a theoretical construct or conceptual constructs that represents something, with a set of variables and a set of logical and quantitative relationships between them.
Monitor *	A software tool or hardware device that operates concurrently with a system or component and supervises, records, analyzes, or verifies the operation of the system or component.
Operation	Major system logical task performed for initiator, which returns control to system when complete.
Operation Group	Set of associated operations based on predefined criteria.

Operational Mode	A major logical system operation mode composed of groups of operations under specific conditions. Usually this is a small number (from 2 to 8) of operational modes defined for a system.
Operational Profile	An operational profile is a quantitative characterization of the expected, recorded, or predicted use of a particular component, assembly, or system.
Performance*	The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.
Petri-Net	An abstract, formal model of information flow, showing static and dynamic properties of the system. A Petri net is usually represented as a graph having two types of nodes (called places and transitions) connected by arcs, and markings (called tokens) indicating dynamic properties.
Protocol*	A set of conventions that govern the interaction of processes, devices and other components within a system.
Reliability Assessment	Reliability assessment is a quantitative assessment of the reliability of a product, system or portion thereof. Such assessments usually employ mathematical modeling, directly applicable results of tests on the product, failure data, and non-statistical engineering estimates.
Reliability*	Reliability is defined as the ability of a device or system to perform a required function under stated conditions for a specified period of time; usually the ability is defined as a probability. Software reliability is the case when the device or system is a software-based system.
Scenario	Detailed interaction model defined to specify such operational scenario.
Sequential*	Pertaining to the occurrence of two or more events or activities in such a manner that one must finish before the next begins.
Software Architecture	The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them. The term also refers to documentation of a system's software architecture.
Software Development Process*	The process by which user needs are translated into a software product. the process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes installing and checking out the software for operational activities. Note: these activities may overlap or be performed iteratively. See: incremental development, rapid prototyping, spiral model, waterfall model.

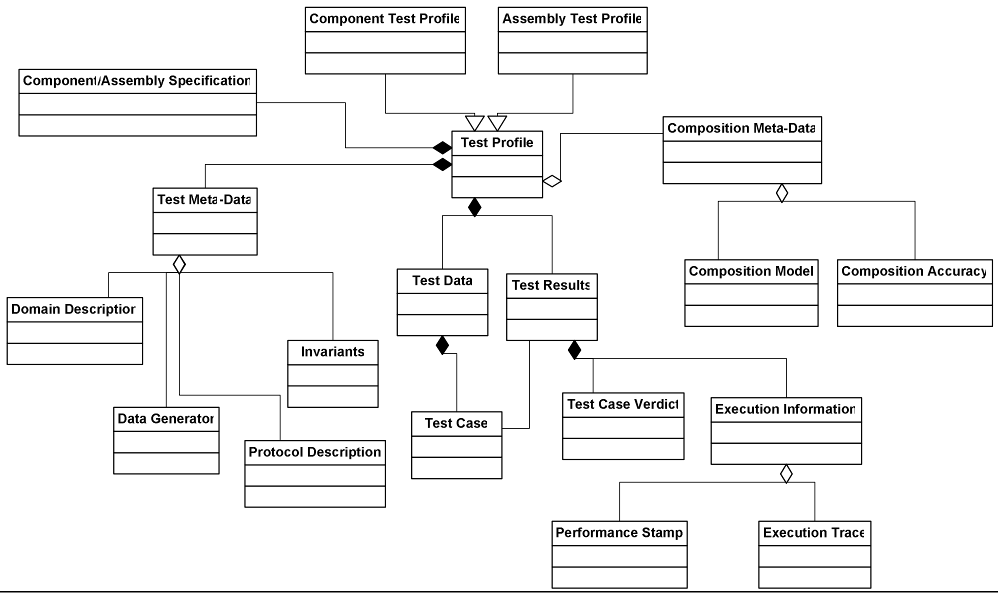
Source Code*	Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler or other translator.
Specification*	A document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or component, and often, the procedures for determining whether these provisions have been satisfied.
Static Analysis	(1) Analysis of a program that is performed without executing the program. (2)* The process of evaluating a system or component based on its form, structure, content, documentation. Contrast with dynamic analysis.
Symbolic Execution*	A static analysis technique in which program execution is simulated using symbols, such as variable names, rather than actual values for input data, and program outputs are expressed as logical or mathematical expressions involving these symbols.
System	Abstraction of any run-time program such as a software component, a system, or a system of systems.
Test Oracle	A test oracle is a source of expected results for a test case.
Test point/ Test Case	A test point or test case is set inputs, execution conditions, and expected results (i.e., set of test outputs) developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
Testing*	(1) The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. (2) The process of analyzing a software item to detect the differences between existing and required conditions, i.e. bugs, and to evaluate the features of the software items. See: dynamic analysis, static analysis, software engineering.
UML	The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.
UML Profile	UML Profiles are extension of the UML that tailor the language to specific areas - some for business modeling; others for particular technologies.

B. Artifact Templates

This section describes the templates proposed for the framework artifacts. These templates are specified using UML class diagrams representing conceptual models. These models specify the information that is suggested for the artifacts. Each template artifact includes the UML conceptual model and a suggested artifact document structure.

B.1 Component/Assembly Test Profile Template

Component/Assembly Test Profile Meta-Model

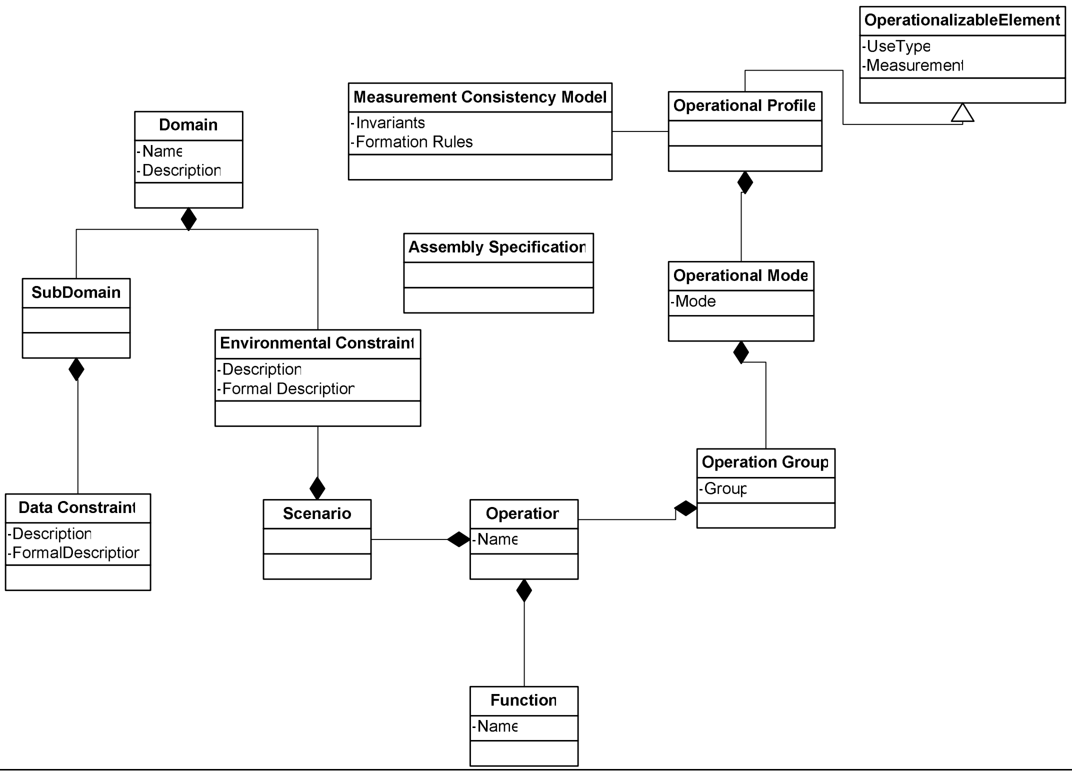


Component/Assembly Test Profile Suggested Structure

1. Component/Assembly Description
2. Component/Assembly Specification
3. Composition Meta-Data
    a. Composition Model
    b. Composition Accuracy
4. Test Meta-Data
    a. Domain Description
    b. Invariants
    c. Data Generator
    d. Protocol Description
5. Test-Data
    a. Test Cases
    b. Test Results
6. Execution-Information

B.2 Assembly Operational Profile Template

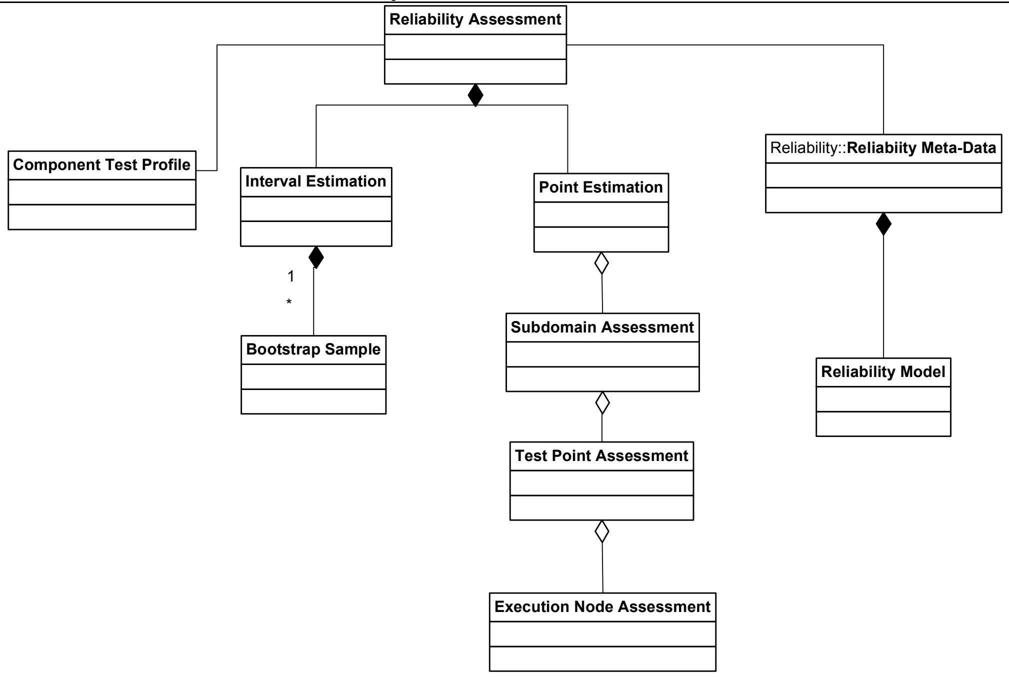
Assembly Operational Profile Meta-Model



Assembly Operational Profile Suggested Structure

1. Component/Assembly Description
2. Assembly Specification
3. Operational Modes
    a. Operational Group 1
        i. Operation 1
        ii. Operation 2
        .
    b. …
4. Basic Functionalities
    a. Function 1
    b. .
    c. .
5. Domain Descriptions
    a. Subdomain 1
    b. Subdomain 2
    c. ….
6. Consistency Model
7. Operational Probability Distribution

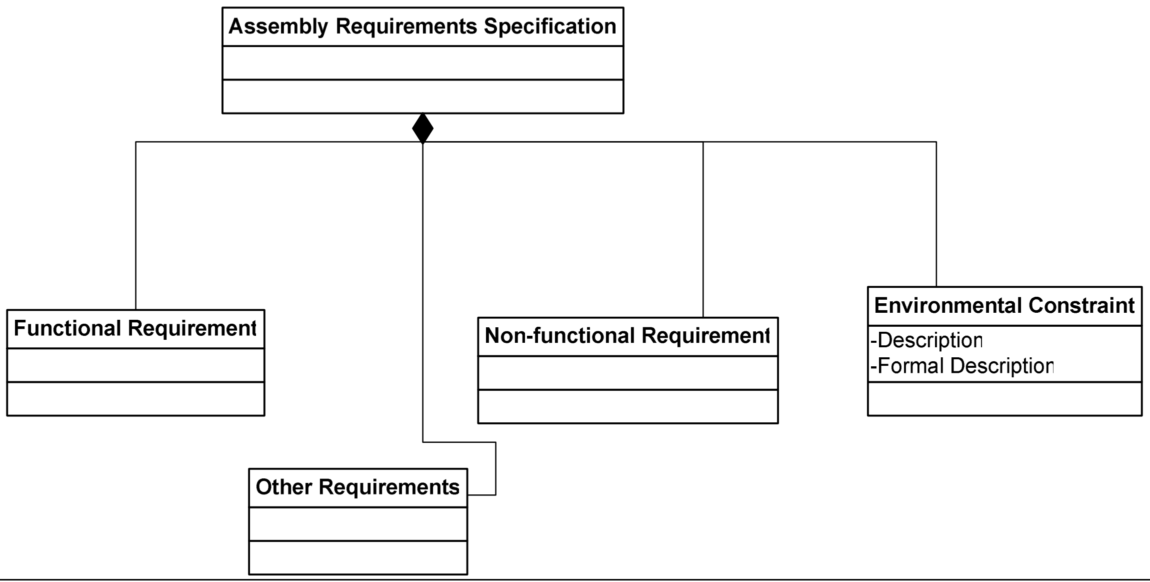
B.3 Reliability Assessment Template

Reliability Assessment Meta-Model
 Reliability Assessment Component Test Profile Interval Estimation Point Estimation Reliability::Reliabiity Meta-Data 1 * Bootstrap Sample Subdomain Assessment Reliability Model Test Point Assessment Execution Node Assessment
Reliability Assessment Suggested Structure

1. Component/Assembly Description
2. Reliability Meta-Data
    a. Reliability Model
3. Reliability Assessment
        i. Point Estimation
        ii. Execution Node Assessment
        iii. Test Point Assessment
        iv. Subdomain Assessment
    b. Interval Estimation
        i. Bootstrap Samples

B.4 Assembly Requirement Specification Template

Assembly Requirement Specification Meta-Model



Assembly Requirement Specification Suggested Structure

1. Assembly Description
2. Environmental Constraints
3. Assembly Functional Requirements
4. Assembly Non-Functional Requirements
5. Other Assembly requirements

C. UML Operational Characterization Profile

Computer Science Department
Software Engineering Group

	The UML Operational Characterization Profile
Doc. Ref.	**2007-01-01**
Version	**DRAFT 0.1**
Date	**01-01-2007**

Author Name	**Gerardo Padilla and Carlos Montes de Oca**	**Version 0.1**
Reviewed by	**TBD**	

ABSTRACT

The size and complexity of current software system require techniques to optimize the testing resources. One recognized approach to handle the limitation of software testing is by using operational information.

Operational information is the specification of the expected use of the system. This specification uses probabilities that are assigned to specific system operations. However, there are more elements that require being incorporated in such descriptions, such as the hierarchy of the system operations, the scenario organization, the constraints, and the corresponding links to the system requirement specification.

This report describes a UML extension allowing the specification of operational information. These modeling constructs are defined using the Unified Modeling Language (version 2.0). Specifically, these elements are organized as an UML profile. The construction of UML profiles is a standard approach to define domain-specific modeling constructs.

The elements defined within the proposed UML profile include: scenario, operation, function, domain, and data constraint. The UML Profile described in this report is named as *UML Operational Characterization Profile.* It follows a metamodeling approach and its organization si similar to other UML profiles.

TABLE OF CONTENTS

INTRODUCTION

The software is playing a major role in our society. Almost every electronic device contains a significant part of software. On the other side, complex and critical machinery such as airplanes, automobiles, and medical devices are based on complex and large portions on software. The development of such complex software requires robust and reliable methodologies, techniques, and technologies [1].

In non-software domains, testing is based on statistical models. However, it is until now that software industry uses statistical techniques to manage the limited testing resources. This new perspective has been adopted following an paradigm that combines the quality control of the process with the quality control of the product [2].

The quality of the product, defined by attributes such as reliability, has been addressed and analyzed for more that 30 years [3, 4, 5]. This long walk of research does not mean that the problem is unsolvable. The problem is that the problem conditions are changing constantly. Sources of this change are the fast changing technological platforms, paradigms, and the new user requirements. Even this dramatic scenario, some generic techniques has been proposed to address the estimation of the software attributes such as reliability. One example of such techniques is the Software Reliability Engineering [6]. This technique proposes a methodological framework where testing efforts are managed using operational information.

Operational information is a term describing information of the intended or expected use of a specific system. This kind of information is usually found in an artifact called *Operational Profile* (the term *profile* in this artifact does not refer to UML profile). Figure 1 shows an example of the information contained in an Operational Profile of a Web Library System.

Operation	Probability
Web Browsing	0.12
Web Search	0.30
Web Transactions	0.40
….	….
Total	*1.00*

Figure 1 Example of Operational Information.

Figure 1 shows a subset of the most important operations defined for a system. In this context, an operation means a major system operation. Additional to these operations, a quantitatively description of the expected use is included (i.e., probabilities).

The use of operational information provides useful data to manage the testing process. However, there is no standard about how to model, specify or analyze operational information.

An UML profile is an UML extension that focuses on a specific domain. This kind of extensions is allowed in the UML language. Examples of such extensions are the UML Testing Profile [7], UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms [8], UML Profile for Schedulability, Performance and Time [9], UML Profile for System on a Chip (SoC) [10], etc. However, there is no UML profile addressing the issue of modeling operational information.

This report proposes a notation to specify such kind of information. This report is organized as an UML Profile. The profile is defined by using a set of metaclasses, as it is proposed in UML 2.0 [11].

THE UML OPERATIONAL CHARACTERIZATION PROFILE

1. Scope

This specification defines a UML extension to represent Operational Information for a particular System. This extension reduces the problems of UML 2.0 for the description of Operational Information.

The UML Operational Characterization Profile defines a language for specifying, analyzing, and documenting operational information of systems. This profile defines a modeling language that can be used with any system whose operational information can be described using such modeling elements. The UML Operational Characterization Profile can be used stand alone or in an integrated manner with UML.

The UML Operational Characterization Profile extends UML with usage-oriented concepts like operations, functions, scenarios, etc. Being a profile, the UML testing profile seamlessly integrates into UML: it is based on the UML metamodel and reuses UML syntax.

The UML Operational Characterization Profile is based on the UML 2.0 specification. This profile is defined by using the metamodeling approach of UML and it has been conceptualized using the next design principles:

- UML integration: as a real UML profile, the UML Operational Characterization Profile is defined on the basis of the metamodel provided in the UML superstructure specification and follows the principles of UML profiles as defined in the UML infrastructure specification of UML 2.0.
- Reuse: wherever possible, the UML Operational Characterization Profile makes direct use of the UML concepts and extends them. Otherwise, it adds new concepts only where it is needed.

2. Conformance

The compliance points are as follows:

1. UML Operational Characterization Profile: A compliant implementation supports the UML profiling mechanism, the UML entities extended by the Operational Characterization Profile, and the stereotyped entities of the UML Operational Characterization Profile.

2. MOF-based Metamodel for Operational Characterization: The compliant implementation supports all of the entities in the MOF-based metamodel.

3. Notation: If graphical notation is used, the compliant implementation recognizably supports the notation defined by the Operational Characterization Profile specification.

4. Static Requirements: The compliant implementation checks the specified constraints automatically.

Compliance requires meeting point 1 and/or 2. Points 3-4 are optional and can be claimed in any combination. A compliance statement should address every point.

3. Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references subsequent amendments to, or revisions of, any of these publications do not apply.

- UML 2.0 Infrastructure Specification [12].
- UML 2.0 Superstructure Specification [11].
- UML 2.0 OCL Specification [13].
- MOF 2.0 Specification [14].

Note – You will find these documents in the OMG Specification Catalog at this URL: http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

4. Terms and Definitions

This section describes the terms and concepts included in the UML Operational Characterization Profile. These concepts are defined as meta-classes within the MOF-based metamodel.

- **Environmental Constraint**: An Environmental Constraint is a condition on the parameters defining the system environment. These constraints include hardware, software, run-time environments, and user characteristics.

- **Data Constraint**: A specification of the allowable ranges defined for a set of data attributes. This specification is composed of two main elements: the data description and the data distribution. The former specifies the allowable values for the corresponding data attribute, and the latter specifies the probabilistic distribution of such allowable values (ranges, or partitions).

- **Domain**: A set of all possible values that any particular program, functionality might require or generate.

- **Subdomain**: is a subset of values from any specific domain. Usually, a set of subdomains forms a partition over a specific domain.

- **Input Domain/ Output Domain**: A classification of two different domains where the Input Domain refers to the set of all possible values that a specific program requires as input and the Output Domain refers to the set of all possible values that a specific program generate as output. A program may be characterized by its corresponding Input and Output domain.

- **Function**: Atomic functionality performed by a system. The combination of one or more functions defines an operation.
- **Operation**: Major system logical task performed for initiator, which returns control to system when complete.
- **Operation Group**: Set of associated operations based on predefined criteria.
- **Operational Profile:** Quantitative characterization of the expected or recorded use of a system**.**
- **Scenario:** a typical interaction between the user and the system or between two software components.
- **Operational Mode:** A set of major logical system operations. A system usually has from 2 to 8 operational modes.
- **System:** Abstraction of any run-time program such as a software component, a system, or a system of systems.

5. Additional Information

5.1 How to read this specification

The rest of this document contains the technical content of this specification. Section 6 describes the technical elements defining the UML profile.

Section 6.1 provides an overview and the approach followed to design this profile. In addition, the motivation behind of this work is described. Section 6.2 describes the elements, structure of the organization of all modeling constructs. Section 6.3 describes how UML profiles are constructed. Section 6.4 describes the metamodel specified for this profile in a MOF style. This section describes specific information items for all metaclasses defined in the metamodel such as semantics, attributes, operations, and descriptions.

6. The UML Operational Characterization Profile

6.1 Overview

This section provides an overview and introduction to the UML Operational Characterization Profile.

The term *Operational Information* describes quantitatively the intended or expected use of any system. This information has two key attributes: (i) the information is based on the system user-perspective, and (ii) the information is designed to be quantitatively specified in terms of intended, expected, or recorded use.

Operational information can be found at three different levels: first, the use can be specified using data domains. Second, the use can be specified at the level of functionalities. Finally, the third level is a combination of both previous levels. The use of any level of quantification depends on the specific system.

The main motivation for this profile is the need of a specialized modeling constructs for such kind of information. Existing UML semantics is capable of modeling and specifying operational information. However, there is a need of standardized constructs and notations that facilitates the analysis of operational information.

Similarly to other UML profiles, a set of stereotypes for each UML metaclass are defined for specific operational concepts. Additionally, the profile has been designed to define a set of new stereotyped modeling elements instead of defining new kind of diagrams. This approach allows the use of almost all other UML modeling capabilities.

The extension includes a new generalization element used to extend the semantic capabilities of any classifier (defined in the Kernel package). The generalization element is named as

OperationalizableElement. This metaclass allows the incorporation of attributes representing operational information.

The consequence of this extension is that almost any UML model may be “operationalizable”. This means that any UML diagram can be “enriched” with measurements of the expected or intended use. The consistency and coherence of any UML operationalized model depends on the specific system.

6.2 Structure of the UML Profile

The UML Operational Characterization Profile is organized in two logical groups of concepts that are stored in three different packages: Package for data-related information, package for functional information, and a package providing generic support.

The first package contains constructs to model operational information at data level; the second contains constructs to model operational information at function level, and the third package contains generic constructs providing support and consistency to all packages.

The UML Operational Characterization Profile is specified by:

- Giving the terminology for a basic understanding of the UML Operational Characterization Profile concepts.
- Defining the UML 2.0 based metamodel of the UML Operational Characterization Profile.
- Defining a MOF model for the pure UML Operational Characterization Profile enabling the use of the Operational Characterization Profile independent of UML.
- Giving examples to demonstrate the use of the UML Operational Characterization Profile for operational system specification using both architectures (data and functional).

In case of ambiguities, the UML 2.0 based metamodel takes precedence.

6.3 The Profile

This profile defines limited extensions to the reference UML 2.0 meta-model with the purpose of adapting the meta-model to a specific platform or domain. The extension of this profile does not change the reference metamodel, and keeps its semantics.

In UML 2.0, profiles are packages that structure UML extensions. The principal extension mechanism in UML 2.0 is the concept of stereotype. Stereotypes are specific metaclasses, having restrictions and the specific extension mechanism. Additional semantics can be specified using Stereotype features ("attributes" in UML2.0) and new well-formedness rules in the context of a profile.

A UML profile extends parts of the UML metamodel in a constrained way. All new modeling concepts must be supported by UML modeling elements. The new attributes must respect the semantic of UML modeling elements. All associations are binary associations. The general structure of the profile model is the same as the general structure of the metamodel.

6.4 MOF-based Metamodel

This section provides a standalone metamodel for the UML Operational Characterization Profile. Figure 2 shows the package structure for this profile. This profile is composed of two packages named as *Data-Oriented Modeling Constructs* and *Operation-Oriented Modeling Constructs*.

The first package, *Data-Oriented Modeling Constructs*, defines the basic constructs to model data-related operational information such as domains, sub-domains, and data constraints. The second package, *Operation-Oriented Modeling Constructs*, defines the basic constructs to model operation-related information, such as function, operations, and operational modes. Finally, the third package, Generic Support, defines generic constructs such as environmental constraints and measurement consistency model.

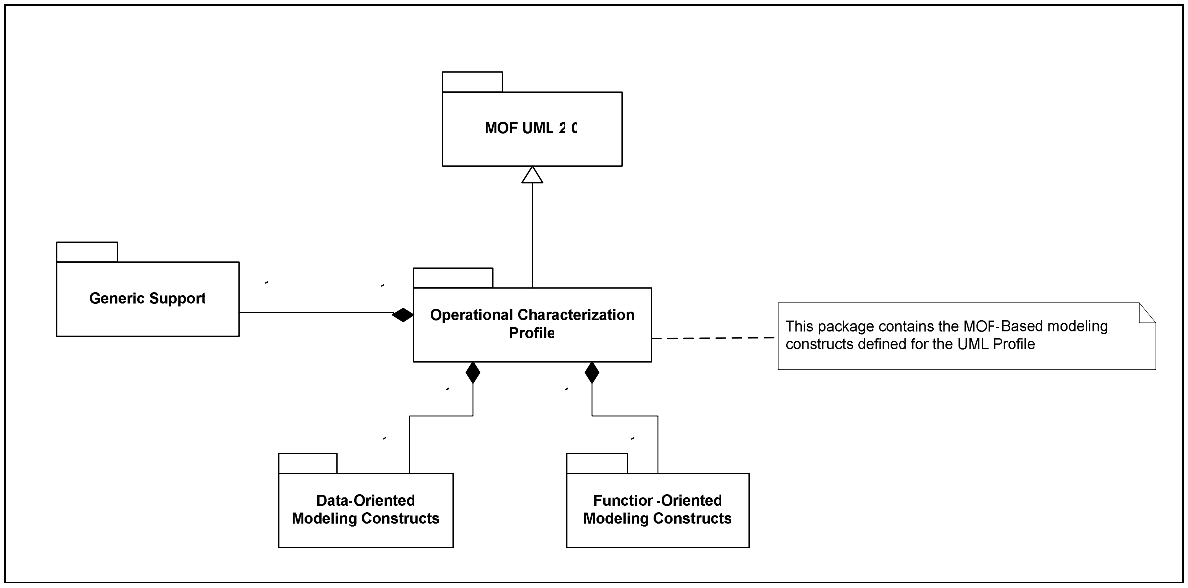

MOF UML 2 0
Generic Support
Operational Characterization Profile
This package contains the MOF-Based modeling constructs defined for the UML Profile
Data-Oriented Modeling Constructs
Function-Oriented Modeling Constructs

Figure 2. Package structure within the UML Operational Characterization Profile.

The compliance provided by the MOF-based metamodel is limited to the modeling elements of the Operational Characterization Profile. This profile allows the use of any other UML construct suitable for operational characterization.

6.4.1 Generic Support Package

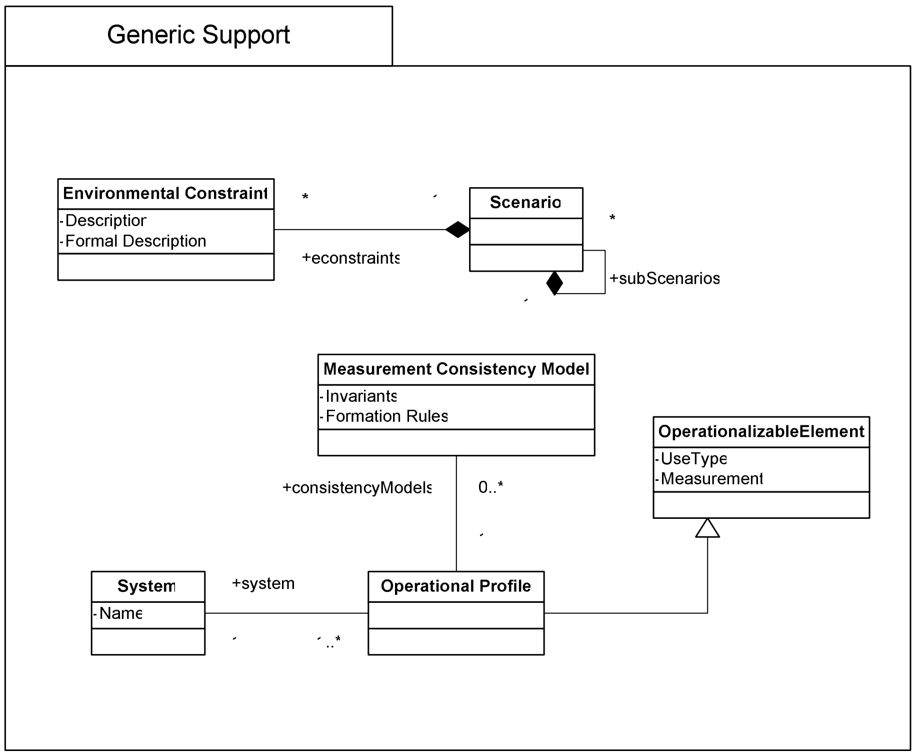


Figure 3. Generic Support Package.

Environmental Constraint

Description and Semantics

An Environmental Constraint is a condition on the parameters defining the system environment (i.e., the conditions of the system execution).

Associations

• econstraints:Scenario[1]	The scenario that is specified with the environmental constraints.
• domain:Domain[1]	The domain that is specified or constrained with the corresponding environmental constraint.

Attributes

• Description: String[1]	The description of the specific environmental constraint.
• Formal Description: String[1]	The formal specification of such constraint.

Measurement Consistency Model

Description and Semantics

The Measurement Consistency Model represents a set of rules defined over measurement attributes in a model for some specific modeling constructs. The rules are defined by *Formation Rules* over the modeling elements and *Invariants* over such elements.

Associations

• operationalProfile: Operational Profile[1]	The operational profile associated to the specific Measurement Consistency Model.

Attributes

• Invariants: String[0..*]	Set of conditions that must be met in the model assuring the model consistency.
• Formation Rules[0..*]	Set of predicates on the use of measurements over the models defined.

Operational Profile

Description and Semantics

An Operational Profile is a quantitative specification of the expected use of a system. The specification can be defined at different levels: operation level, data level, or a combination of both levels.

Associations

• consistencyModels: Measurement Consistency Model[0..*]	The consistency models defined for an specific operational profile.
• system:System[1..*]	The system that the Operational Profile specifies.
• modes:OperationalMode[1..*]	The operational profile is composed of one or more operational modes covering the entire major functionalities of the system.

OperationalizableElement

Description and Semantics

The *OperationalizableElement* is used to extend the modeling capabilities for measurement specification. The attributes defined in this class, which are extended to all classifier modeling constructs, allow the specification of constraints and invariants over any model that uses this feature.

Attributes

• UseType: String[1]	This attribute specifies the use of the measurement. Using this parameter allows the inclusion of such measurement in

	the invariants defined in the Measurement Consistency Models.
• Measurement:Value[1]	Te measurement or value defined for the modeling construct.

Scenario

Description and Semantics

A Scenario is a typical interaction between the user and the system or between two software components. A scenario can be described as a set of scenarios following a structured organization such as a state chart.

Associations

- subScenarios: Scenario[0..*] An scenario may represent a structured set of scenarios following a hierarchy.
- environmentalContraint: Environmental Constraint[0..*] The environmental constraints defined for an specific scenario.

Attributes

• Name: String[1]	This attribute specifies the name of the specific scenario.
• Context:String[1]	This attribute is a textual description of the context where the scenario may occur.

System

Description and Semantics

A system is an abstraction of any run-time program such as a software component, a system, or a system of systems

Associations

- operationProfile:Operational Profile[1..*] The operational profile defined for the system.

Attributes

- Name: String[1] This attribute specifies the name of the system.

6.4.2 Data-oriented Modeling Constructs Package

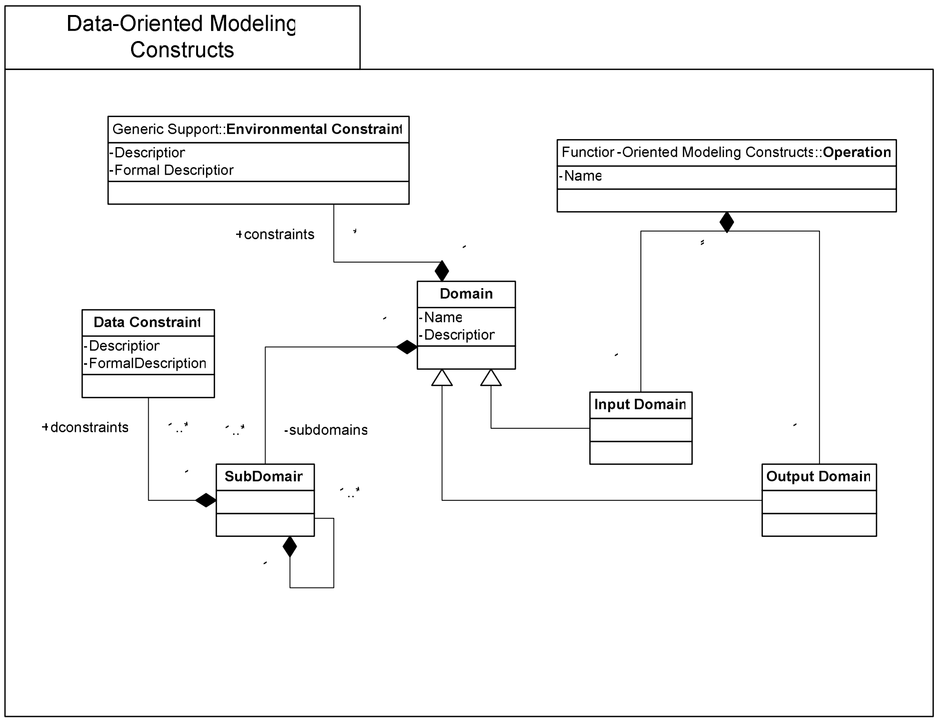


Figure 4. Data-oriented Modeling Constructs Package.

Data Constraint

Description and Semantics

A specification of the allowable ranges defined for a set of data attributes. This specification is composed of two main elements: the data description and the data distribution. The former specifies the allowable values for the corresponding data attribute, and the latter specifies the probabilistic distribution of such allowable values (ranges, or partitions).

Associations

- subDomain:SubDomain[1] The subdomain which above the constraint is defined.

Attributes

- Description: String[1] The textual description of the constraint.
- FormalDescription: String[1] The formal description used in the constraint.

Domain

Description and Semantics

A Domain is a set of all possible values that any particular program, functionality might require or generate.

Associations

- subDomains:SubDomain[1..*] The set of subdomains that represents the domain.
- constraints:EnvironmentalContraint[0..*] The set of environmental constraints defined for the subdomain.

Attributes

- Name: String[1] The domain name.

- Description: String[1] The description of the domain.

SubDomain

Description and Semantics

A Subdomain is a subset of values from any specific domain. Usually, a set of subdomains forms a partition over a specific domain.

Associations

- subDomains:SubDomain[1..*] The set of subdomains that represents the domain.
- dconstraints:DataConstraint[1..*] The set of data constraints that defines the subdomain.

InputDomain

Description and Semantics

An Input Domain is the domain referred to all input values required by a specific function, operation, or program.

Associations

- operation:Operation[1] The set of allowed input values defined for the operation.

OutputDomain

Description and Semantics

An Output Domain is the domain referred to all output values produced by a specific function, operation, or program.

Associations

- operation:Operation[1] The set of allowed output values defined for the operation.

6.4.3 Function-oriented Modeling Constructs Package

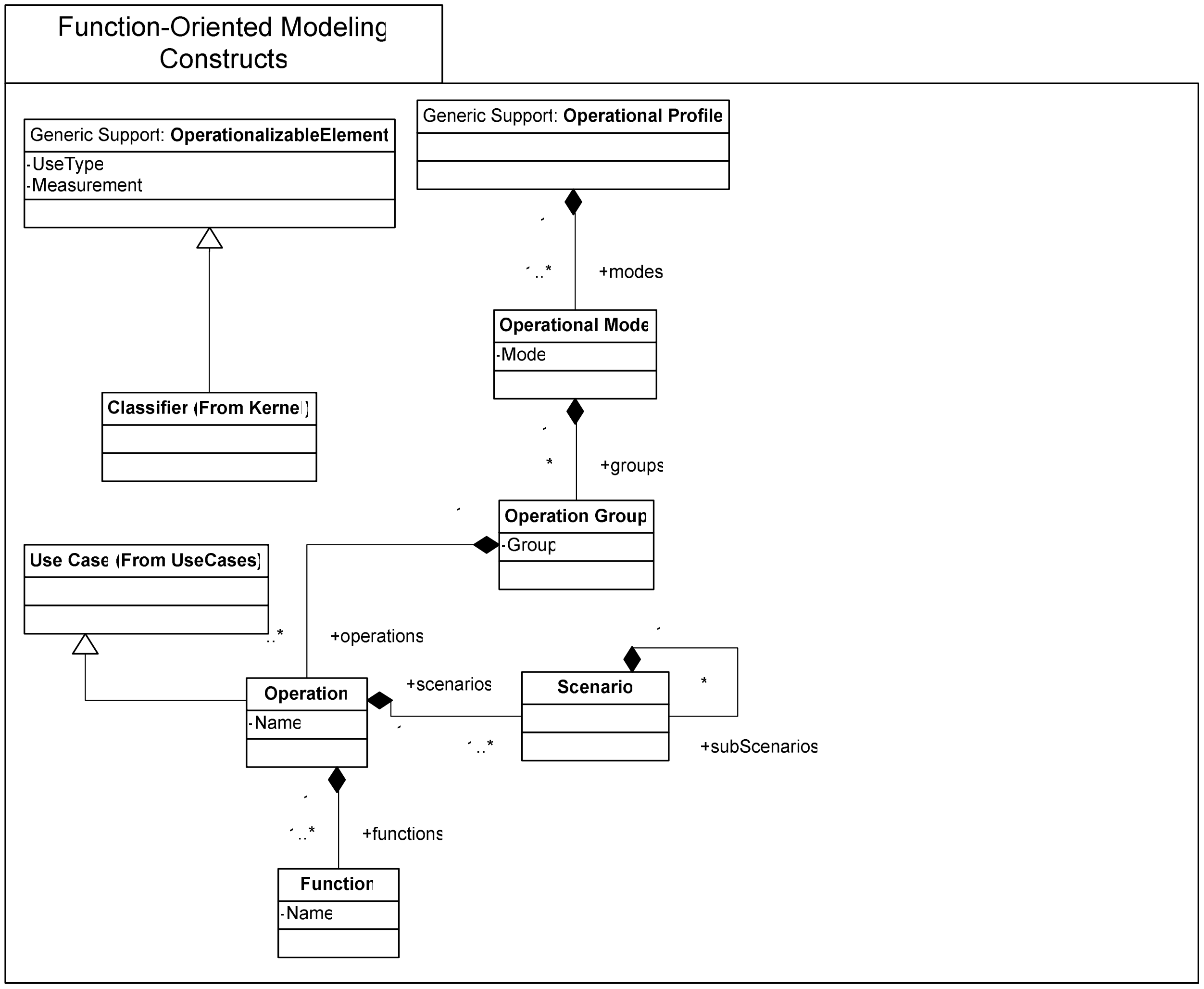


Figure 5. Function-oriented Modeling Constructs Package.

Operational Mode

Description and Semantics

An Operational Mode is a set of major logical system operations. A system usually has from 2 to 8 operational modes.

Associations

- operationalProfile:Operational Profile [1] The operational profile that contains the operational modes.
- groups:OperationGroup[0..*] Operations groups are associated to the Operational Model.

Attributes

- Mode: String[1] Operational mode of the system.

Operation

Description and Semantics

An Operation is a major system logical task performed for an initiator, which returns control to system when complete.

Associations

- functions:Function [1..*] The operation may be composed or defined by at least one concrete function.
- operationGroup:OperationGroup[1] Operations are arranged in operation groups.
- scenarios:Scenario[1..*] An operation may be described using one or more scenario.

Attributes

- Name: String[1] Operation name.

Operation Group

Description and Semantics

An Operation Group is a set of associated operations based on predefined criteria.

Associations

- operationalMode:Operational Mode [1] An operational mode may be defined by one or more operation groups.
- operations:Operation[1..*] An operation group is a set of operations.

Attributes

- Group: String[1] Operation group name.

Function

Description and Semantics

A Function is an atomic functionality performed by a system. The combination of one or more functions defines an operation.

Associations

- operation:Operation [1] A function is associated to one operation.

Attributes

- Name: String[1] Function name.

6.4.4 Merging All Packages

Figure 6 shows the resulting package (i.e., the Operational Characterization Profile).

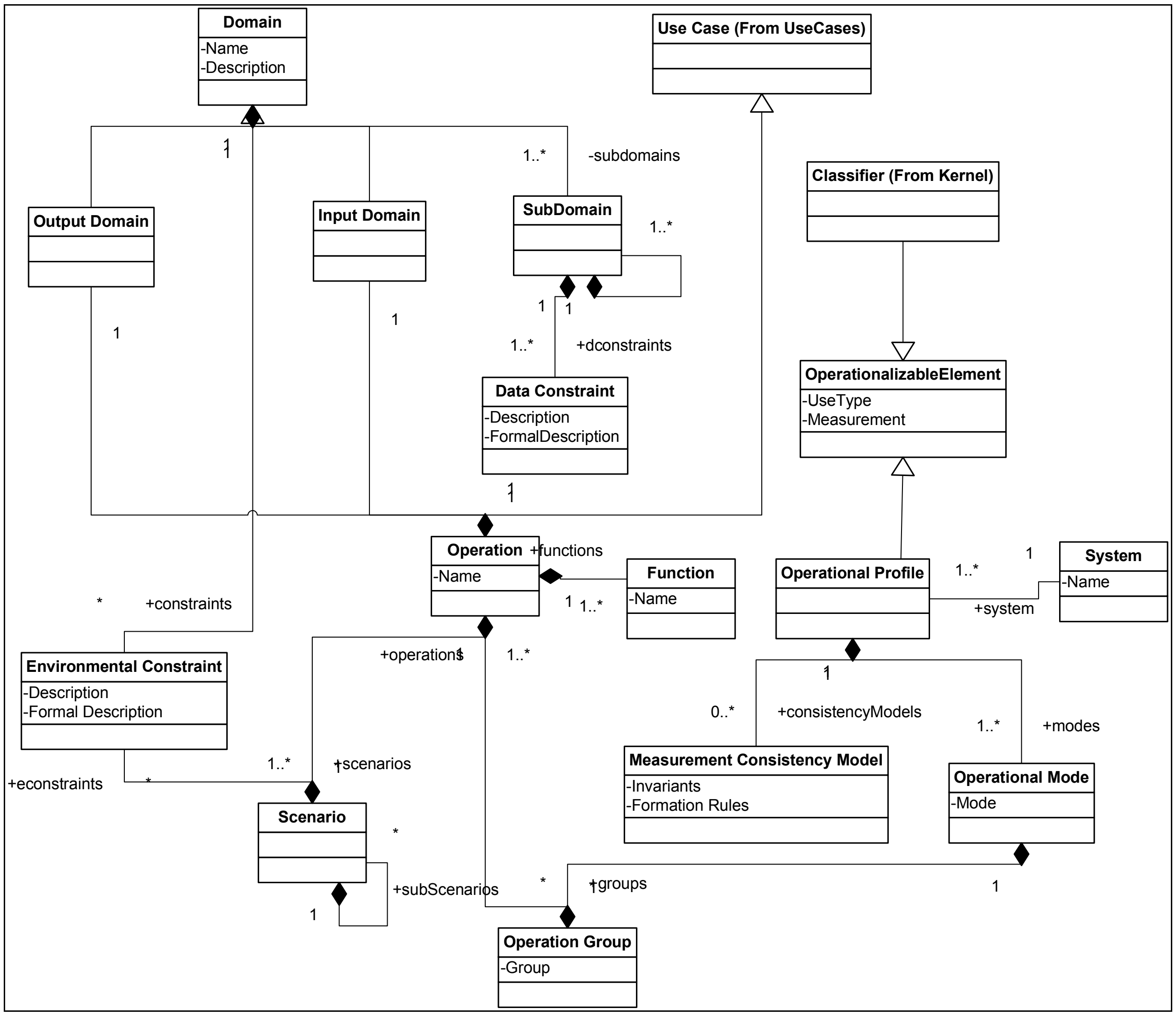


Figure 6. All Modeling Constructs defined for the UML Profile.

7. Examples

This section presents two examples to illustrate the use of this profile. The first example uses the functional-related modeling constructs to specify an operational profile for a *WEB Library System*. The second example uses the data-oriented modeling constructs to define an operational profile for a *Security Monitoring System*.

7.1 Example 1: WEB Library System

Figure 7 shows a model representing the operational profile for a WEB Library System. The operational profile is specified at the operation level. Four major operations are defined for this profile: *Review Account, Search Catalogue, Renew Books,* and *Request Books*. Every operation has the *Measurement* attribute denoting the expected use probability. The *Measurement Consistency Model* is defined as an invariant over the measurement attributes in the model.

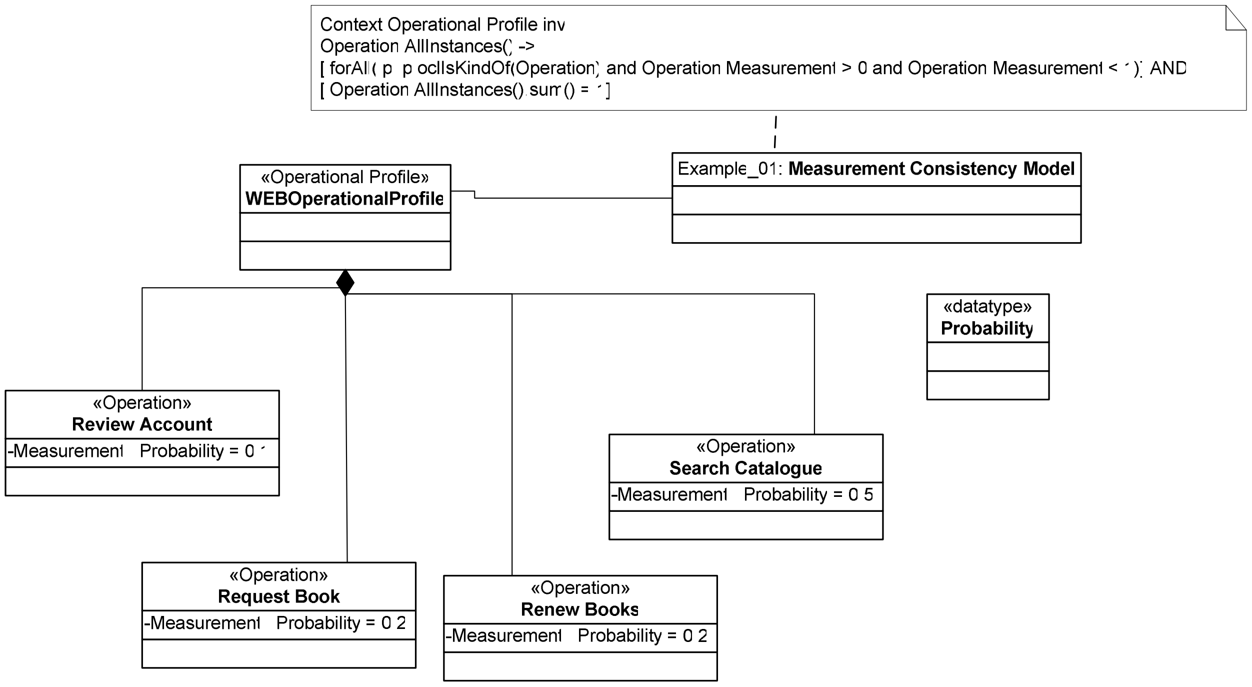

Context Operational Profile inv
Operation AllInstances() ->
[forAll(p p oclIsKindOf(Operation) and Operation Measurement > 0 and Operation Measurement < 1)] AND
[Operation AllInstances() sum() = 1]
«Operational Profile»
WEBOperationalProfile
Example_01: Measurement Consistency Model
«datatype»
Probability
«Operation»
Review Account
-Measurement Probability = 0 1
«Operation»
Search Catalogue
-Measurement Probability = 0 5
«Operation»
Request Book
-Measurement Probability = 0 2
«Operation»
Renew Books
-Measurement Probability = 0 2

Figure 7. Web Library System Example.

7.2 Example 2: Security Monitoring System

Figure 8 shows a model representing the operational profile for a Security Monitoring System. The operational profile is defined at the data level. The input domain defined for this model is a set of attributes over an image file. Three sub-domains are defined for this profile: *Production Storage, and Corporate Section*. Every sub-domain has the *Measurement* attribute denoting the expected use probability. Every sub-domain has a data constraint characterizing the specific subdomain. The *Measurement Consistency Model* is defined as an invariant over the measurement attributes in the model.

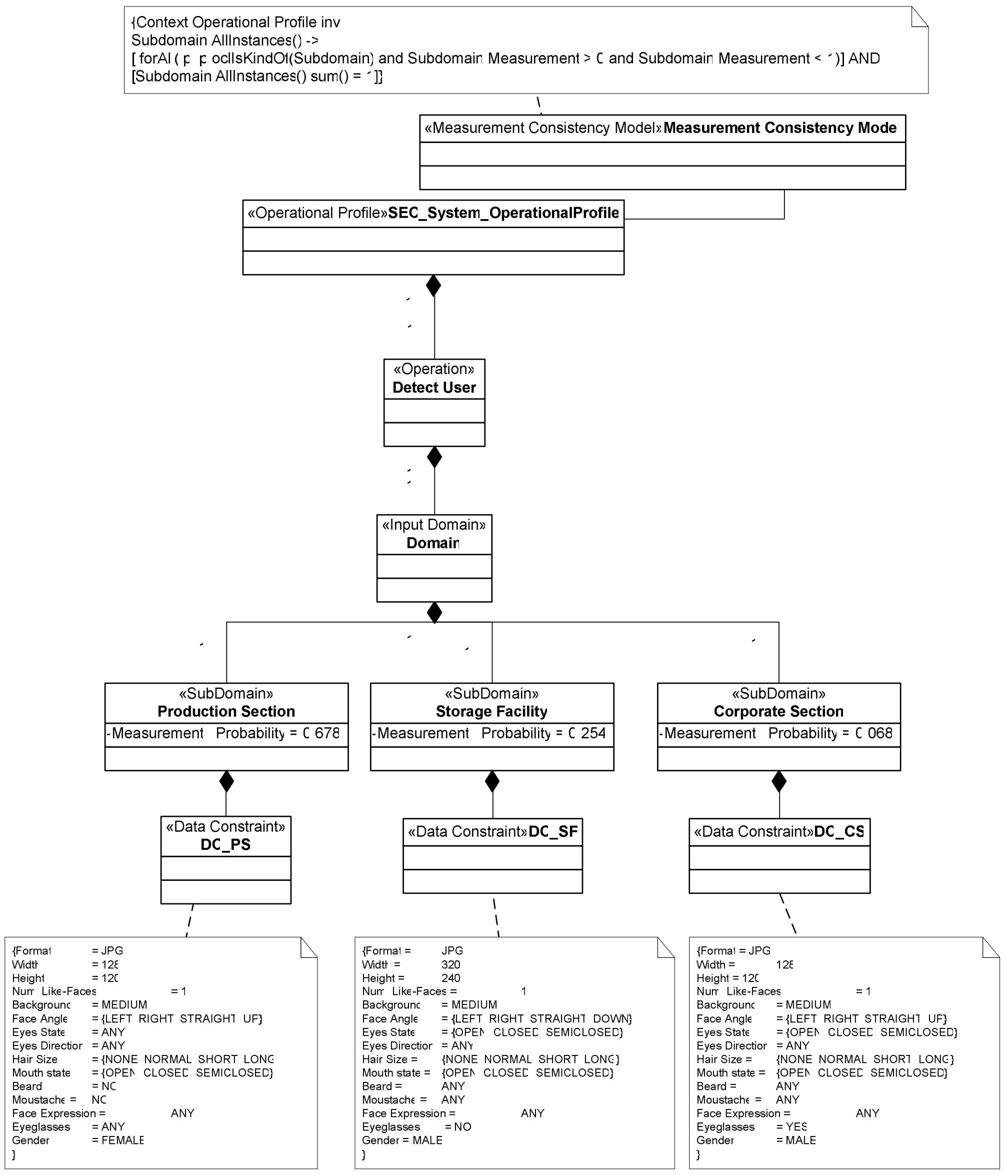

{Context Operational Profile inv
Subdomain AllInstances() ->
[forAl (p p oclIsKindOf(Subdomain) and Subdomain Measurement > C and Subdomain Measurement < ´)] AND
[Subdomain AllInstances() sum() = ´]}
«Measurement Consistency Model» Measurement Consistency Mode
«Operational Profile» SEC_System_OperationalProfile
«Operation» Detect User
«Input Domain» Domain
«SubDomain» Production Section
-Measurement Probability = C 678
«SubDomain» Storage Facility
-Measurement Probability = C 254
«SubDomain» Corporate Section
-Measurement Probability = C 068
«Data Constraint» DC_PS
«Data Constraint» DC_SF
«Data Constraint» DC_CS
{Format = JPG
Width = 128
Height = 120
Num Like-Faces = 1
Background = MEDIUM
Face Angle = {LEFT RIGHT STRAIGHT UP}
Eyes State = ANY
Eyes Direction = ANY
Hair Size = {NONE NORMAL SHORT LONG
Mouth state = {OPEN CLOSED SEMICLOSED}
Beard = NC
Moustache = NC
Face Expression = ANY
Eyeglasses = ANY
Gender = FEMALE
}
{Format = JPG
Width = 320
Height = 240
Num Like-Faces = 1
Background = MEDIUM
Face Angle = {LEFT RIGHT STRAIGHT DOWN}
Eyes State = {OPEN CLOSED SEMICLOSED}
Eyes Direction = ANY
Hair Size = {NONE NORMAL SHORT LONG}
Mouth state = {OPEN CLOSED SEMICLOSED}
Beard = ANY
Moustache = ANY
Face Expression = ANY
Eyeglasses = NO
Gender = MALE
}
{Format = JPG
Width = 128
Height = 120
Num Like-Faces = 1
Background = MEDIUM
Face Angle = {LEFT RIGHT STRAIGHT UP}
Eyes State = {OPEN CLOSED SEMICLOSED}
Eyes Direction = ANY
Hair Size = {NONE NORMAL SHORT LONG}
Mouth state = {OPEN CLOSED SEMICLOSED}
Beard = ANY
Moustache = ANY
Face Expression = ANY
Eyeglasses = YES
Gender = MALE
}

Figure 8. Face-Based Recognition Security System Example.

8. Final Remarks and Future Work

The profile described in this report has been proposed to address the current need of standard notations to specify operational information. The UML Operational Characterization Profile has been designed to be use with other modeling constructs allowing the use of other UML capabilities.

Future work will focus on:

- Analyze the semantic compatibility with other UML profiles such as the Test Profile [7] and the QoS profile [8].
- Include a set of standard measurement consistency models to guide the user to build operational models.

9. References

[1] W.S. Humphrey "Winning with Software: An Executive Strategy," *The SEI Series in Software Engineering* -256, 2002.

[2] Denise,A., Gaudel,M. and Gouraud,S. "A Generic Method for Statistical Testing," *International Symposium on Software Reliability Engineering* 2004.

[3] J. Musa "Tools for measuring software reliability," *IEEE Spectrum* vol. 26, no. 2, pp. 39-42, 1989.

[4] E. Nelson "Estimating Software Reliabilty from Test Data," *Microelectronics and Reliability* vol. 17, pp. 67-74, 1978.

[5] S. Gokhale, W. Wong, K. Trivedi and J. Horgan "An Analytical Approach To Architecture-Based Software Reliability Prediction," *International Symposium on Computer Performance and Dependability (IPDS)* pp. 13-22, 1998.

[6] J.D. Musa "Software Reliability Engineering: More Reliable Software Faster And Cheaper," p. 608, 2004.

[7] Object Management Group, "UML Testing Profile", Version 1.0, Formal/05-07-07, Available at: http://www.uml.org/#UML2.0

[8] Object Management Group, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms", Version 1.0, Formal/06-05-02, Available at: http://www.uml.org/#UML2.0

[9] Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification", Version 1.1, Formal/05-01-02, Available at: http://www.uml.org/#UML2.0

[10] Object Management Group, "UML Profile for System on a Chip (SoC)", Version 1.0.1, Formal/06-08-01, Available at: http://www.uml.org/#UML2.0

[11] Object Management Group, "Unified Modeling Language: Superstructure", Version 2.1, PTC/2006-04-02, Available at: http://www.uml.org/#UML2.0

[12] Object Management Group, "Unified Modeling Language: Infrastructure", Version 2.1, PTC/06-04-03, Available at: http://www.uml.org/#UML2.0

[13] Object Management Group, "Object Constraint Language", Version 2.0, Formal/06-05-01, Available at: http://www.uml.org/#UML2.0

[14] Object Management Group, "Meta Object Facility (MOF) Core Specification", Version 2.0, Formal/06-01-01, Available at: http://www.uml.org/#UML2.0