VERIFICATION AND VALIDATION IN SOFTWARE PRODUCT LINE ENGINEERING

Edward A. Addy

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

Doctor of Philosophy
in
Computer Science

John R. Callahan, Ph.D., Chair
John M. Atkins, Ph.D.
Stephen Edwards, Ph.D.
Ali Mili, Ph.D.
Murali Sitaraman, Ph.D.

Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
1999

Keywords: Verification, Validation, Software Product Line, Software Reuse, Assurance, Formal Reasoning, Criticality Analysis, Risk Assessment

Copyright 1999 EDWARD A. ADDY

ABSTRACT

Verification and Validation in Software Product Line Engineering

By Edward A. Addy

Verification and Validation (V&V) is currently performed during application development for many systems, especially safety-critical and mission-critical systems. However, the V&V process has been limited to single system development. This dissertation describes the extension of V&V from an individual application system to a product line of systems that are developed within an architecture-based software engineering environment.

In traditional V&V, the system provides the context under which the software will be evaluated, and V&V activities occur during all phases of the system development lifecycle. The transition to a product line approach to development removes the individual system as the context for evaluation, and introduces activities that are not directly related to a specific system. This dissertation presents an approach to V&V of software product lines that uses the domain model and the domain architecture as the context for evaluation, and enables V&V to be performed throughout the modified lifecycle introduced by domain engineering.

This dissertation presents three advances that assist in the adaptation of V&V from single application systems to a product line of systems. The first is a framework for performing V&V that includes the activities of traditional application-level V&V, and extends these activities into domain engineering and into the transition between domain engineering and application engineering. The second is a detailed method to extend the crucial V&V activity of criticality analysis from single system development to a product line of systems. The third advance is an approach to enable formal reasoning, which is needed for high assurance systems, on systems that are based on commercial-off-the-shelf (COTS) products.

DEDICATION

I am honored to dedicate this publication to the members of my family, who have encouraged me, supported me, and sacrificed their time so that I could concentrate on this work. I want to express my love and gratefulness to my parents, Alton J. and Margaret G. Addy, who have given me encouragement and guidance throughout my life, and who serve as shining examples of parental love. I thank my daughters, Rachel Ann Addy and Rebekah Lynn Addy, for allowing me to be a student alongside them, and for their willingness to allow me the time and energy required for this effort. Most of all, I express my appreciation and admiration to my wife, Eleanor Ann Spriggs Addy. She is my inspiration and my encouragement, the one who shares my burdens and my dreams, my partner in the duties and joys of this life. I love and appreciate each one of you.

ACKNOWLEDGEMENTS

I would like to thank the current and former members of the NASA/WVU Software Research Laboratory for their continuous support. I am also grateful to the members of the Reusable Software Research Group at West Virginia University and to members of the Institute for Software Research for numerous discussions relating to topics within this dissertation. I am very appreciative for the assistance provided to me by the members of my graduate committee, including Dr. John R. Callahan, who served as my academic advisor and principle investigator as well as the chair of the graduate committee.

I gratefully acknowledge the financial support for my research provided by the NASA/Ames Software Technology Division at the NASA Software IV&V Facility in Fairmont, WV, under NASA Cooperative Agreements NCC-0040 and NCC 2-979. Portions of this work also received support under DARPA grant DAAH04-96-1-0419, monitored by the U. S. Army Research Office.

TABLE OF CONTENTS

Abstract ii
Dedication iii
Acknowledgements iv
Table of Contents v
List of Tables vi
List of Figures vi

Chapter 1 - Introduction 1
1.1 Verification and Validation in Traditional System Application Engineering 1
1.2 Justification for Performing V&V within Domain Engineering 2
1.3 Organization of the Dissertation 4

Chapter 2 - Related Work 5
2.1 Verification and Validation Process 5
2.2 Software Product Lines 5
2.3 Component Certification 6
2.4 Criticality Analysis 7
2.5 COTS Integration 8

Chapter 3 - Framework for Performing V&V within Product line Software Engineering 9
3.1 Domain-Level Tasks 11
3.2 Correspondence Tasks 12
3.3 Communicating Results 13

Chapter 4 – Determining Levels of Assurance 14
4.1 Criticality Considerations 14
4.2 Criticality 15
4.3 Marketability 17
4.4 Assurance 17

Chapter 5 – Formal Reasoning about COTS-Based Systems 21
5.1 Surface Movement Advisor 21
5.2 Formal Specification of the Flight Takeoff Prediction Subsystem 22
5.3 Mathematical Modeling of Aspects of A COTS Database 25
5.4 Modular Reasoning about a COTS-Based Subsystem 27
5.5 Status 30
5.6 Lessons Learned 30

Chapter 6 – Conclusions and Future Work 32

References 35
Appendix A. Assurance Scores for the Components of the Waiting Queue Simulation Product Line 40
Appendix B. RESOLVE Specification of the SMA Flight Takeoff Time Prediction Subsystem 64

LIST OF TABLES

Table 1-1. Minimum V&V Tasks for Critical Software in Application Engineering 3
Table 3-1. V&V Tasks for Life-Cycle Phases at the Domain and Transition Levels 11
Table 4-1. Component Risk Likelihood Drivers 16
Table 4-2. Component Severity Categories and Ratings 16
Table 4-3. Determining Relative Levels of Needed Assurance for Components in a Software Product Line 18
Table 4-4. Assurance Scores for Components of the Waiting Queue Simulation Product Line 19

LIST OF FIGURES

Figure 3-1. STARS Two Life-Cycle Model 10
Figure 3-2. Framework for V&V within Reuse-Based Software Engineering 10
Figure 4-1. Architecture for the Waiting Queue Simulation Product Line 19
Figure 5-1. Departure Gate Areas and Several Predefined Splits for Atlanta Hartsfield International Airport 23
Figure 5-2. Flight Departure Prediction Concept 24
Figure 5-3. Mathematical Modeling of Database Information 26
Figure 5-4. Illustration of Modular Reasoning 28
Figure 5-5. Modular Reasoning for the SMA Flight Departure Prediction Algorithm 28
Figure 5-6. Database Interface Contract (part 1) 29
Figure 5-7. Database Interface Contract (part 2) 29

CHAPTER 1 - INTRODUCTION

V&V methods are used to increase the level of assurance of critical software, particularly that of safety-critical and mission-critical software. Software V&V is a systems engineering discipline that evaluates software in a systems context [Wallace and Fujii 1989a]. The Verification and Validation (V&V) methodology has been used in concert with various software development paradigms, but always in the context of developing a specific application system. However, the product line software development process separates domain engineering from application engineering in order to develop generic reusable software components that are appropriate for use in multiple applications.

The earlier a problem is discovered in the development process, the less costly it is to correct the problem. To take advantage of this, V&V activities begin within system application development at the concept or high-level requirements phase. However, a product line software development process has tasks that are performed earlier, and possibly much earlier, than high-level requirements for a particular application system.

In order to bring the effectiveness of V&V to bear within a product line software development process, V&V must be incorporated within the domain engineering process. Failure to incorporate V&V within domain engineering will result in higher development and maintenance costs due to losing the opportunity to discover problems in early stages of development and having to correct problems in multiple systems already in operation. Also, the same V&V activities will have to be performed for each application system having mission or safety-critical functions.

On the other hand, it is not possible for all V&V activities to be transferred into domain engineering, since V&V within domain engineering does not have identical goals to V&V within application engineering. Domain engineering V&V focuses on considerations across multiple systems, including the extensibility and flexibility of the product line architecture, generic functionality of components, and conformance to interface definitions. Application engineering V&V focuses on considerations related to a specific application system, such as timing and capacity constraints and involvement of a component in a specific critical process. This leads to the question of which existing (and/or new) V&V activities would be more effectively performed in domain engineering rather than in (or in addition to) application engineering. Related questions include how to identify the reusable components for which V&V at the domain level would be cost-effective, and how to determine the level to which V&V should be performed on the reusable components.

Software product lines routinely rely on the use of Commercial-off-the-shelf (COTS) software for which source code and design information is not available. Methods to reason about systems that include COTS components, and to guide the testing of COTS components, are needed in order to provide the high level of assurance often required in critical systems.

1.1 Verification and Validation in Traditional System Application Engineering

V&V has been performed during application system development, within the context of many different development methodologies, including waterfall, spiral, and evolutionary development. V&V is a set of activities performed in parallel with system development and is designed to provide assurance that a software system meets the needs of the user. It ensures that the requirements for the system are correct, complete, and consistent, and that the life-cycle products correctly implement system requirements.

The term *verification* refers to the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. The term *validation* means the process of evaluating software at the end of the software development process to ensure compliance with software requirements [IEEE 610.12 1990]. Verification is intended to ensure that the product is built correctly, while validation assures that the correct product is built (in the sense that the product meets the needs of the user).

While verification and validation have separate definitions, in practice the activities are merged into a single process. This process evaluates software in a systems context, using a structured approach to analyze and test the software against system functions and against hardware, user and other software interfaces [Wallace and Fujii 1989a]. V&V is also described as a series of technical and management activities performed to improve the quality and reliability of that system and to assure that the delivered product satisfies the user's operational needs [Lewis 1992].

V&V activities are designed to be independent of but complementary to the activities of the development and test teams. The development team is usually focused on nominal performance and the testing is usually based on requirements and operational profiles. V&V includes both analysis and tests, and considers critical and off-nominal behavior throughout all phases of the development lifecycle. V&V activities also complement the activities of the configuration management and quality assurance groups rather than being a duplicate or replacement of these activities [Wallace and Fujii 1989b].

A set of minimal and optional V&V activities is defined in the IEEE Standard for Software Verification and Validation Plans [IEEE 1012 1992]. These activities are divided into the life-cycle phases listed below. The minimum V&V tasks for critical software are shown in Table 1-1.

- Management of V&V
- Concept Phase V&V
- Requirements Phase V&V
- Design Phase V&V
- Implementation Phase V&V
- Test Phase V&V
- Installation and Checkout Phase V&V
- Operations and Maintenance Phase V&V

V&V is performed as a part of a risk mitigation strategy for application systems. The risks can be in areas such as safety, security, mission, finance, or reputation. The scope and level of V&V can vary with each project, based on the criticality of the system and on the role of software in accomplishing critical functions of the system [Makowsky 1992]. V&V is used to determine which software functions are involved in high-risk areas, and V&V activities are focused on this critical software. Criticality analysis is used to determine not only the critical software, but also the level of intensity to which each V&V task should be performed on various components of the critical software [IEEE 1059 1993].

1.2 Justification for Performing V&V within Domain Engineering

Studies have shown that the cost and difficulty of correcting an error increases dramatically as the error is discovered in later life-cycle phases [Boehm 1976, Makowsky 1992]. V&V addresses that issue in traditional system development through activities that begin in the

PHASE	TASKS
Management	Software Verification and Validation Plan Generation Baseline Change Assessment Management Review Review Support
Concept	Concept Documentation Review
Requirements	Software Requirements Traceability Analysis Software Requirements Evaluation Software Requirements Interface Analysis System Test Plan Generation Acceptance Test Plan Generation
Design	Design Traceability Analysis Design Evaluation Design Interface Analysis Component Test Plan Generation Integration Test Plan Generation Test Design Generation • component testing • integration testing • system testing • acceptance testing
Implementation	Source Code Traceability Analysis Source Code Evaluation Source Code Interface Analysis Source Code Documentation Evaluation Test Case Generation • component testing • integration testing • system testing • acceptance testing Test Procedure Generation • component testing • integration testing • system testing Component Test Execution
Test	Test Procedure Generation • acceptance testing Integration Test Execution System Test Execution Acceptance Test Execution
Installation and Checkout	Installation Configuration Audit V&V Final Report Generation
Operations and Maintenance	Software V&V Plan Revision Anomaly Evaluation Proposed Change Assessment Phase Task Iteration

Table 1-1. Minimum V&V Tasks for Critical Software in Application Engineering

concept or high-level requirements phase and continue throughout all life-cycle phases. The V&V activities are focused on high-risk areas, so that errors in the high-risk areas can be discovered in time to evolve a complete and cost effective solution rather than forcing a makeshift solution due to schedule constraints.

Within product line software engineering, software engineering activities may be performed prior to the concept phase of a particular application system. In order to extend the benefit of early error detection to software product line engineering, V&V must be incorporated within the domain engineering process. Performing V&V at the domain level may also reduce the level of effort required to perform V&V in the individual application systems.

Although software is the target of V&V activities, software does not execute in isolation, but rather as an integral part of a system [Duke 1989]. In order to provide assurance that critical functions will be performed correctly, software must be evaluated within the context in which it will execute. In product line software engineering, the context for V&V must be provided by the domain model and domain architecture [Ran 1999].

1.3 Organization of the Dissertation

This dissertation describes the extension of the activities of V&V from the development environment of single systems to the development environment of a product line of systems. The current chapter presents an overview of V&V and of software product lines, and motivates the need for extending V&V into software product lines. Chapter 2 discusses other work that is related to the topic of V&V and software product lines. Chapter 3 presents a framework for performing V&V within a software product line environment. Chapter 4 describes the extension of criticality analysis from single system development to software product line development. Chapter 5 presents a method that enables formal reasoning about software systems that are based on Commercial-off-the-shelf (COTS) products, which is a common occurrence within software product lines. Chapter 6 presents a summary of this work and discusses future areas of research

CHAPTER 2 - RELATED WORK

We are not aware of any other research that is focused specifically on the area of performing verification and validation within software product lines. However, there is much work addressing areas directly related to this research, including the V&V process, software product lines, reusable component certification, criticality analysis, and COTS integration. These areas are discussed below.

2.1 Verification and Validation Process

IEEE standards provide industry guidance for conducting V&V. The IEEE Standard for Software Verification and Validation Plans [IEEE 1993] designates a set of minimal and optional V&V activities. These activities are divided into the life-cycle phases listed in Chapter 1. In addition to the standard, the IEEE has approved the Guide for Software Verification and Validation Plans [IEEE 1993] to provide information on the tailoring and application of the standard.

The chair and co-chair of the working group for the original 1986 IEEE Standard for Software Verification and Validation Plans were, respectively, Roger Fujii and Dolores Wallace. Shortly after the publication of the standard, Fujii and Wallace also authored several publications that served to define the V&V process. These publications include a NIST Special Publication entitled *Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards* [Wallace 1989a] and an article entitled “Verification and Validation: An Overview” [Wallace 1989b] in an issue of IEEE Software that focused on V&V. This issue of IEEE Software contained several other papers describing aspects of the V&V process [Musa 1989, Duke 1989, Dunham 1989]. A general text book on V&V was published in 1992, authored by V&V practitioner Robert Lewis and entitled *Independent Verification and Validation, A Life Cycle Engineering Process for Quality Software* [Lewis 1992]. Callahan and Sabolish discuss a process improvement model for software verification and validation [Callahan 1996]. Kitchenham and Linkman argue for the strategy of selecting a diverse set of V&V techniques based on the functionality and quality required by the product [Kitchenham 1998].

Many organizations have developed guidelines for V&V that are tailored to their particular development or acquisition process and to the domain of their systems. Examples of these documents include A Guide to Independent Verification and Validation of Computer Software, written for the Army Mission-Critical Computer Resources programs under the Communications-Electronics Command [Makowsky 1992], the NASA Software Assurance Guidebook [NASA 1989], and NIST guidelines for performing V&V for health care systems [Wallace 1996].

All of this work in V&V has focused on the development of individual software systems. The processes and activities are set entirely within the environment of application engineering, and do not consider the existence of any software development activities outside of the individual system being developed. No V&V activities are considered for software artifacts such as domain models, generic architectures, or reusable components.

2.2 Software Product Lines

The concept of software product lines grew out of work in the related areas of software reuse and domain-specific software architectures. Software product lines are a specialized form

of software reuse, and the product line architecture plays a central role in the development and evolution of a software product line.

The topic of software product lines has recently emerged as a distinct area of study, with national and international workshops held beginning in 1996 [ARES 1997, Bass 1997, Addy 1998a, Bass 1998a, Clements 1998]. The Software Engineering Institute (SEI) at Carnegie Mellon University has established a Product Line Practice Initiative, and has published a case study on product line engineering [Brownsword 1996]. Robyn Lutz has performed an analysis of requirements that support a product line, and in particular analyzed the product line components for hazards using efforts similar to Software Failure Modes and Effects Analysis and Software Fault Tree Analysis [Lutz 1999].

[Garlan 1996] offers an overview of the state of the art in software architectures, although this overview is directed more toward system architectures than toward product line architectures. The analysis of software architectures is the focus of attention and discussion [Tracz 1996, Garlan 1995], but there is not as yet consensus on methods and approaches. One of the approaches being researched that may be applicable to product line architectures is a scenario-based analysis approach, Software Architecture Analysis Method [Kazman 1995]. In the area of tracking dependencies between domain and application components, the Centre for Requirements and Foundations at Oxford is developing a tool (TOOR) to support tracing dependencies among evolving objects [Goguen 1996].

Two of the leading texts that deal with the combined issues of software reuse and software architecture are [Jacobson 1997] and [Bass 1998b]. The SEI has also produced two reports on software architecture that address the issue of evaluating a software architecture [Abowd 1997] [Barbacci 1997]. [DeBaud 1997] describes a methodology with tool support to enable the development and deployment of software product lines.

2.3 Component Certification

Much work has been done in the area of component certification, which is also called evaluation, assessment, or qualification. These terms can have slightly different meanings, but refer in general to rating a reusable component against a specified set of criteria.

Reuse libraries often use a discrete scale to indicate the degree of effort and the methods by which the library has evaluated a component. The Asset Source for Software Engineering Technology (ASSET) library and the Army Reuse Center library both have four levels of certification, although the use of the term "levels" is operationally different in the two libraries [Poore 1992]. The libraries evaluate reusable components against criteria such as reusability, evolvability, maintainability, and portability, as well as expending various amounts of effort to ensure the component meets its specification.

The Certification of Reusable Software Components Program at Rome Laboratory has proposed a certification framework based on removing defects from candidate reusable components [SPS 1996]. This certification process consists of four levels of analysis and testing, each designed to remove certain categories of defects from the reusable component. The levels of analysis and testing correspond to more stringent levels of certification, which are composed of the factors of scope and confidence.

The Comprehensive Approach to Reusable Defense Software (CARDS) library is a model-based library based on a generic architecture. Reusable components are evaluated not only on the same general criteria as that of component-based libraries, but also on the "form, fit, and function" relative to the generic architecture [Unisys 1994]. The CARDS library uses this difference to draw a distinction between "certification" and "qualification." The Component

Providers and Tool Developers Handbook defines component certification as "The process of determining if a component being considered for inclusion in a library meets the *requirements of the library* and passes all testing procedures. Evaluation takes place against a common set of criteria (reusability, portability, etc.)." Component qualification is defined as "The process of determining if a potential component is *appropriate to the library* and meets all quality requirements. Evaluation takes place against domain criteria." (Italics added for emphasis.) Hence, qualified components not only meet quality requirements of the library related to testing and standards, but are also appropriate for the domain area and architectural style of the library.

Knight and Dunn [Knight 1998] have proposed an evaluation framework that extends beyond functional correctness to consider other aspects of reusable components, such as performance, maintainability, portability, readability, and testability. They propose that the aspects of interest in the reusable components be determined relative to their ability to support desired properties in the application products. Their scheme first determines the qualities desired in the application products, and then uses this information to determine the properties that the components should have and the techniques by which the properties will be demonstrated.

As in other industries, manufacturers of software systems acquire components and turn these into consumer products. The common thread through all of these certification processes is the focus on the component rather than the systems in which the component will eventually be (re)used. Dunn and Knight note that with the exception of the software industry itself, customers purchase systems and not components [Dunn 1993]. Ensuring that components are well designed and reliable with respect to their specifications is necessary but not sufficient to show that the final system meets the needs of the user. Component evaluation is but one part of an overall V&V effort, analogous to code evaluation in V&V of an application system.

Another distinction between V&V and component certification is the scope of the artifacts that are considered. While component certification is primarily focused on the evaluation of reusable components (usually code-level components), V&V also considers the domain model and the generic architecture, along with the connections between domain artifacts and application system artifacts. Some level of component certification should be performed for all reusable components, but V&V is not always appropriate. V&V should be conducted at the level determined by an overall risk mitigation strategy.

2.4 Criticality Analysis

V&V is performed as a part of a risk mitigation strategy for application systems. The risks can be in areas such as safety, security, mission, finance, schedule, or reputation. The scope and level of V&V can vary with each project, based on the criticality of the system and on the role of software in accomplishing critical functions of the system [Makowsky 1992]. V&V determines the software involved in high-risk areas, and V&V activities are focused on this critical software.

The existing risk methods that are concerned with operation risk include Criticality Analysis and Risk Assessment (CARA) [McCaugherty 1998], and criticality analysis in Independent Software Nuclear Safety Analysis (ISNSA) [NASC 1989] and in the Independent V&V (IV&V) methods of the USAF Space Division [Shere 1998]. These methods do not identify the areas of operation risk, but rather are used to prioritize subsections of the software based on their association with operation risk. The subsections being prioritized are usually either sections of the requirements document or design or code modules. These methods require that studies (e.g., safety analysis, mission analysis) be performed prior to the criticality/risk assessment, and that the software artifact be decomposed into the subsections to be prioritized.

CARA was developed by Averstar, Inc. (formerly Intermetrics, Inc.), primarily in the context of IV&V of the Space Station and Shuttle programs. CARA is used to prioritize code modules in order to determine the level of IV&V to be applied to each module. CARA assigns an ordinal value of "criticality" to the relationship of each code module with mission, safety, and development, and an ordinal value of "risk" based on the Complexity, Maturity of Technology, Requirements Stability, Testability, and Developer Experience for the module. The CARA score of a module is the product of the average criticality score and the average risk score, and this score is used to prioritize the modules. The restriction of criticality considerations to mission, safety, and development rises from the context in which CARA was developed (e.g., security was not a major issue).

ISNSA was developed to provide assurance for nuclear weapons systems deployed by the U.S. Navy; hence the only area of interest was (nuclear) safety. The United Stated government requires the design of nuclear weapons to include a sequence of stages that must be achieved prior to detonation, and the activities that cause a weapon to reach particular stages are termed critical factors. Criticality analysis within ISNSA uses ordinal values to indicate the involvement (direct, indirect, none) of requirements subsections or code modules with each of the critical factors, and the maximum of these values across all critical factors is used to determine the priority of the component.

Criticality analysis as performed by the USAF Space Division is similar to CARA in that the criticality of a component is determined by the average product of the severity level and likelihood level across all critical factors. However, the critical factors are determined uniquely for each system rather than using a predetermined set, which allows more flexibility in dealing with software from different domains. The critical factors are divided into the two categories of performance factors and delivery factors, where performance factors are concerned with operation issues and delivery factors are concerned with development issues.

2.5 COTS Integration

COTS components are routinely used as significant portions of systems developed using a software product line approach. Systems that require higher levels of assurance often need to use some amount of formal methods as a part of the assurance process. While there is on-going work on developing COTS-based software and on the use of formal methods in the reuse community, there are few concrete efforts to integrate both of these issues.

A significant part of COTS work is on comparative evaluation [Oberndorf 1997], although there is no accepted standard of evaluation [Carney 1997, Challenge 2000]. The National Product Line Asset Center (NPLACE) has established a method of evaluating COTS products against a set of predefined testable criteria. Voas has developed a method for COTS certification that involves testing the product based on the operational profile of the system, system-level fault injection, and operational system testing [Voas 1998]. He also advocates taking defensive measures by putting a wrapper around the COTS software to limit the output of the COTS software to acceptable values. However, [Weyuker 1998] discusses many difficulties in effectively testing components that are being used to construct a system, either as individual components or within the developed system. [Leach 1997] suggests that the best approach may be to locate a COTS vendor that can be trusted, and attempt to match the interface specifications.

Formal methods literature addresses issues in supporting component certification [Leavens 1998], component selection [Chen 1997], and component modification [Jeng 1994]. However, none of these considers the formal specification of COTS software.

CHAPTER 3 - FRAMEWORK FOR PERFORMING V&V WITHIN PRODUCT LINE SOFTWARE ENGINEERING[1]

One model for reuse-based software engineering is the Two Life-Cycle Model shown in Figure 3-1, developed by the U.S. Department of Defense Software Technology for Adaptable, Reliable Systems (STARS) program. This model assumes a domain-specific, architecture-centered approach to software reuse. The domain model describes the problem space of the domain, and expresses requirements. The domain architecture describes the solution space of the domain, while the domain components are intended to be used within application systems to meet the functions described in the domain architecture. The arrows depict the dependencies between the activities. Multiple boxes within Application Engineering indicate that multiple systems can be developed from the products of Domain Engineering.

A draft framework for performing V&V within product line software engineering is formed by adding V&V activities to the STARS Two Life-Cycle Model. The application-level IV&V tasks described in [IEEE 1012 1992] serve as a starting point. Domain-level tasks are added to link life-cycle phases in the domain level, and correspondence tasks in the transition level are added to link application phases with domain phases. This draft framework was refined by a working group at Reuse '96 [Addy 1996], and further described in [Addy 1998b]. A diagram of the resultant framework is shown in Figure 3-2. The specific tasks of each phase at the domain and transition levels are listed in Table 3-1.

Domain-level V&V tasks are performed to ensure that domain products fulfill the requirements established during earlier phases of domain engineering. Correspondence tasks in the transition between the domain level and the application level provide assurance that an application artifact correctly implements the corresponding domain artifact. Traditional application-level V&V tasks ensure the application products fulfill the requirements established during previous application life-cycle phases.

Performing V&V tasks at the domain and transition levels will not automatically eliminate any V&V tasks at the application level. However, it might be possible to reduce the level of effort for some application-level tasks. For example, if domain level V&V has demonstrated that a component correctly implements a specified functionality, application level V&V can focus on the use of the functionality within the context of the system.

Domain maintenance and evolution are handled in a manner similar to that described in the operations and maintenance phase of application-level V&V. V&V evaluates the changes proposed to domain artifacts in order to determine the impact of the proposed correction or enhancement. If the evaluation determines that the change will impact a critical area or function within the domain, appropriate V&V activities are repeated to assure the correct implementation of the change.

Although not shown as a specific V&V task for any particular phase of the lifecycle, criticality analysis is an integral part of V&V planning. Criticality analysis is performed in V&V of application development in order to allocate V&V resources to the most important (i.e., critical) areas of the software [IEEE 1059 1993]. This assessment of criticality and the ensuing determination of the level of effort for V&V tasks are crucial also within product line software

---

[1] Portions of this chapter appeared in the Annals of Software Engineering, Vol 5, 1998, and are used here by permission of the publisher.

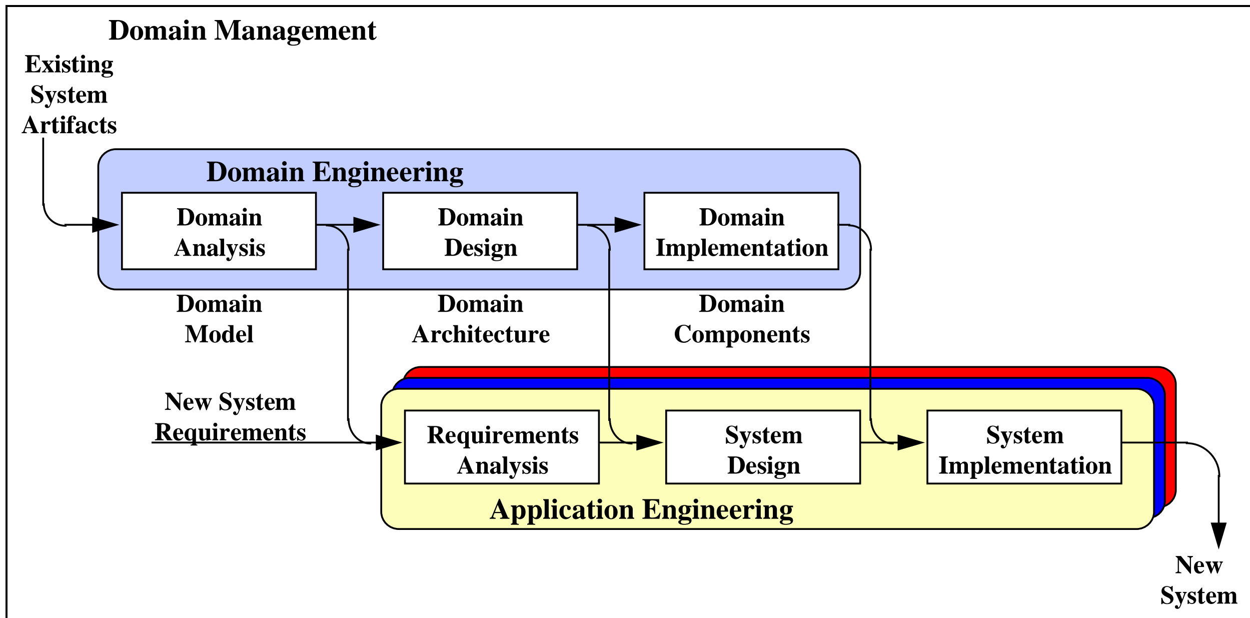


Figure 3-1. STARS Two Life-Cycle Model

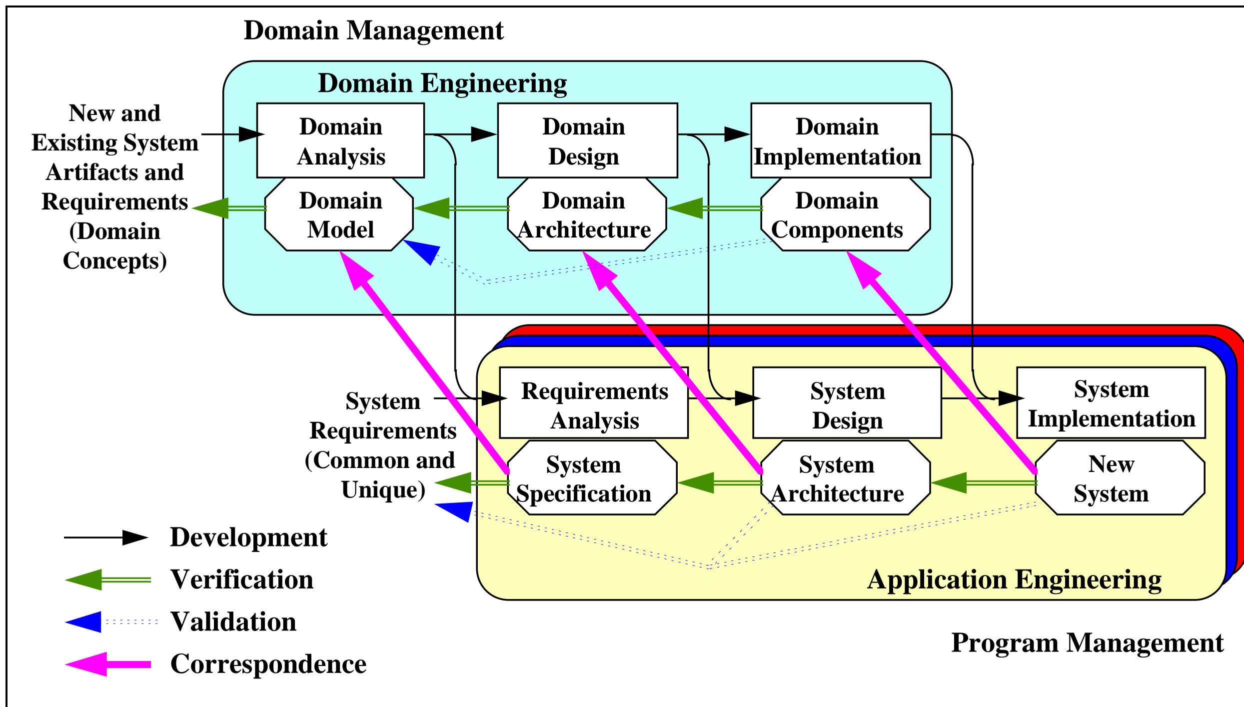


Figure 3-2. Framework for V&V within Reuse-Based Software Engineering

LEVEL	PHASE	TASKS
Domain Engineering	Domain Analysis	Validate Domain Model Model Evaluation Requirements Traceability Analysis (especially forward traceability for completeness)
	Domain Design	Verify Domain Architecture Design Traceability Analysis Design Evaluation Design Interface Analysis Component Test Plan Generation Component Test Design Generation
	Domain Implementation	Verify and Validate Domain Components Component Traceability Analysis Component Evaluation Component Interface Analysis Component Documentation Evaluation Component Test Case Generation Component Test Procedure Generation Component Test Execution
Transition	Requirements	Correspondence Analysis between System Specification and Domain Model
	Design	Correspondence Analysis between System Architecture and Domain Architecture
	Implementation	Correspondence Analysis between System Implementation and Domain Components

Table 3-1. V&V Tasks for Life-Cycle Phases at the Domain and Transition Levels

software engineering. The adaptation of criticality analysis from application development to product line software engineering is discussed in Chapter 4 of this dissertation.

3.1 Domain-Level Tasks

The domain-level tasks are analogous to the application-level tasks, in that the products of each phase are evaluated against the requirements specified in the previous stage and against the original user requirements. The domain-level tasks can be divided into the three phases of domain analysis, domain design, and domain implementation, which correspond to the application phases of requirements, design, and implementation.

During domain analysis V&V, the V&V team should ensure that the domain model is an appropriate representation of the user requirements. (The singular term "model" is not intended to imply that only one model will be constructed; this term is used to mean the one or more models that express the domain requirements.) Note that ensuring that user requirements are satisfied implies that the requirements of the domain must be explicitly stated. Criticality analysis is performed to ensure that high risk requirements are appropriately addressed, either mission-critical requirements or those related to properties such as safety and security. The criticality analysis should also determine critical functions that will be performed by software. The domain model is evaluated to ensure that the requirements are consistent, complete, and realistic, especially in the high risk areas. The model is evaluated to determine responses to error and fault conditions and to boundary and out-of-bounds conditions. As the domain engineering progresses into later phases, the requirements are traced forward. This will allow evaluation of the impact of changes to the domain artifacts.

Domain design V&V tasks focus on ensuring that the domain architecture satisfies the requirements expressed in the domain model. Each requirement in the domain model should trace to one or more items in the domain architecture (forward traceability), and each item in the domain architecture should trace back to one or more requirements in the domain model (reverse traceability). The domain architecture is evaluated to ensure that it is consistent, complete, and realistic. Interfaces between components are evaluated to ensure that the architecture supports the necessary communication between components in the architecture, users, and external systems. Planning and design of component testing are performed during this phase. The component testing should include error and fault scenarios, functional testing of critical activities, and response to boundary and out-of-bounds conditions.

Domain Implementation V&V tasks ensure that the domain components satisfy the requirements of the domain architecture and will satisfy the original user requirements. The components should have a forward and reverse tracing with the domain architecture. Components that are involved with performing critical actions should receive careful consideration. The interface implementation, both within components of the architecture and with systems outside the architecture, is evaluated to ensure that it meets the requirements of the domain architecture. Component test cases and test procedures are generated, and component testing is performed.

Integration test activities are explicitly omitted from the domain-level tasking, since integration testing is oriented toward application-specific testing. Some form of integration testing might be appropriate within domain-level V&V in the case where the architecture calls for specific domain components to be integrated in multiple systems. This limited form of integration testing could be done along with the component testing activities.

3.2 Correspondence Tasks

Correspondence analysis is a term not found in [IEEE 1012 1992]. The term is used within this dissertation to describe the activities that are performed to provide assurance that an application artifact corresponds to a domain artifact; i.e., the application artifact is a correct implementation of the domain artifact. Four activities are to be performed during correspondence analysis:

- Map the application artifact to the corresponding domain artifact.
- Ensure that the application artifact has not been modified from the domain artifact without proper documentation.
- Ensure that the application artifact is a correct instantiation of the domain artifact.
- Obtain information on testing and analysis on a domain artifact to aid in V&V planning for the application artifact.

Correspondence analysis is performed between the corresponding phases of the domain engineering and application engineering life-cycles. The system specification for any system within the domain should correspond to the domain model. The system specification could involve instantiating, parameterizing, or simply satisfying the requirements expressed in the domain model. Any system-unique requirements should be explicit, and the rationale for not addressing these system-unique requirements within the domain model should be stated. Although some degree of correspondence analysis should be at least implicitly performed for all systems developed in accordance with the domain architecture, more care should be taken for systems with critical functions and for their critical areas of software.

The system architecture is analyzed to ensure that it satisfies the requirements specified in the domain architecture. Any variations should be documented along with the reason for the variation. The rationale for parameters chosen or options selected in constructing the system architecture from the domain architecture should be recorded.

The system components are analyzed to ensure correspondence to domain components. Again, variations, parameters, and options should be recorded along with their rationale. Baseline testing might be appropriate in order to compare variants of a domain component.

3.3 Communicating Results

Communicating V&V work products and results is vital to avoiding the repetition of V&V tasks and to ensuring that potential reusers can properly assess the status of reusable components. V&V work products and results should be associated with the component and made available to domain and application engineers. In some cases, V&V efforts might be directed at a grouping of components rather than at an individual component, and this information should also be available. Groupings might include components that are expected to occur together in several applications, or might include variants of one domain artifact.

The information on similar types of components within the domain should be consistent in content and format, in order to allow the information to be easily used by both domain engineers and application engineers. The information that should be communicated includes the following:

- V&V Planning Decisions and Rationale
- V&V Analysis Activities
- V&V Test Cases and Procedures
- V&V Results and Findings

The framework presented in this chapter provides an overall structure for performing V&V in an architecture-centered approach to reuse-based software engineering. Since V&V is one portion of an overall risk mitigation strategy, criticality analysis plays a crucial role in determining the level of V&V that will be performed and the portions of the software that will receive the most attention. The next chapter discusses the process of criticality analysis within a reuse-based environment.

CHAPTER 4 – DETERMINING LEVELS OF ASSURANCE

In accord with the state of the practice in V&V, current methods of criticality analysis that are used to determine the level of assurance required by components are directed at individual application systems. Criticality analysis needs to be extended to include consideration of multiple application systems developed from reusable components. The result of this extension to the current criticality analysis methods is a method to evaluate the criticality of reusable components within a product line that can be used to guide the level of effort for V&V within the product line.

4.1 Criticality Considerations

The Component Verification, Validation and Certification Working Group at WISR 8 considered the issue of V&V within reuse-based software engineering. This working group found four considerations that should be used in determining the level of V&V for reusable components: [Edwards 1997]

- Span of application – the number of components or systems that depend on the component
- Criticality – potential impact due to a fault in the component
- Marketability – degree to which a component would be more likely to be reused by a third party
- Lifetime – length of time that a component will be used

The concept of criticality is related to the concept of "software risk," which is an overloaded term within software engineering literature. The more general term of "risk" usually receives a more formal definition, and has a more uniform meaning across the literature. The typical definition of risk is the expected loss from an unwanted event, and risk is calculated as the product of the loss from the unwanted event and the probability of the unwanted event occurring. In evaluating software risk, the concept of loss is usually termed as "severity," and values are assigned from an ordinal set whose range is fixed by the method as a means of normalizing severities of different types (e.g., injury to a human, cost overrun, schedule slippage). Many of the software risk evaluation methods also use a value from an ordinal set rather than a true probability, so the term "likelihood" is more appropriate than "probability" in these methods.

While the meaning of risk is generally accepted, differences arise among the existing software risk evaluation methods in determining the set of unwanted events, and in the manner in which the risk of the unwanted events are evaluated. The set of unwanted events of a particular method fall into one of two general categories: operation risk and development risk. The set of unwanted events for operation risk focuses on outcomes from the actual use of the system, where operating the delivered system causes an undesired outcome of some type. In development risk, the set of unwanted events centers on the development of the system, and is concerned with the system being developed on schedule, in budget, and with full specified functionality. Operational risk is the category of software risk needed to evaluate the criticality of artifacts in a software product line.

The activities performed by the current application-oriented, operation risk evaluation methods can be used to evaluate the criticality consideration for reusable components listed by the WISR working group, but extensions must be made to include the other considerations of

span of application, marketability, and lifetime. The extended method must be able to accept the evaluation of the criticality and lifetime of a component in multiple systems (i.e., the span of application), as well as the consideration of being used in other systems (e.g., marketability). The following definitions will be useful in discussing the evaluation of the level of assurance required by a component within the context of a product line of systems.

For a component within a particular system

severity	-	level of worst-case impact due to an error in the component
longevity	-	level of time during which the component is anticipated to be used in the system
risk	-	indication of the likelihood and maximum severity of an undesired event due to an error in the component
criticality	-	indication of the risk in the lifetime usage of the component in the system

For a component independent of any particular system

likelihood	-	level of probability that the component has an error
marketability	-	anticipated level of use of the component beyond the identified systems
assurance	-	relative level of effort needed for assurance of this component

The method of determining the assurance level of reusable components described below is based primarily on the CARA method used during IV&V of NASA software, as described in Chapter 2. The method includes some modifications to the basic CARA method in order to make the terminology more consistent with the general literature and to include elementary risk factors from other methods. The intent of this description is not to propose CARA as the ideal method for determining assurance levels, but rather to demonstrate how any such method can be extended from considering a single application system to considering a software product line.

4.2 Criticality

To determine the likelihood that a component has errors, the component is evaluated against a set of Likelihood Drivers. The drivers were determined by examining similar drivers in CARA, the SEI Risk Evaluation Method [Sisti 1994], and a risk assessment method based on the COCOMO scale drivers [Madachy 1997]. The Likelihood Drivers are divided into the categories of Product, Process, and Personnel, and are listed in Table 4-1. Each component is rated against each driver on a scale of 1 to 3, where 1 indicates a low level of risk of an error due to the driver and 3 indicates a high level of risk. The overall Likelihood Score for the component is the average of the likelihood scores over the nine drivers.

The severity level of each component must be determined relative to each system in which the component will be used. The component is rated on a scale of 0 to 4 against each of three severity categories, according to the outcome of maximum severity to which an error in the component could contribute within the system. The categories and rating scales are shown in Table 4-2, with same descriptions of the ratings. The rating descriptions should be tailored to the conditions and situation of the actual product line. The overall Severity Score for the component relative to a system is the average of the three severity levels for the categories.

The product of the Likelihood Score for a component and the Severity Score for a component/system pair is called the Risk Score (which is consistent with the usual definition of

PRODUCT
    Complexity (multiple cases, timing constraints, high reliability, fault tolerance, availability)
    Precedentedness
    Testability
    Platform Difficulty
    Stability (requirements, architecture, platform)
 	
PROCESS
    Maturity
    Complexity (multiple organizations, multiple sites)
 	
PERSONNEL
    Experience (technology, process, platform)
    Stability

Table 4-1. Component Risk Likelihood Drivers

SEVERITY CATEGORIES	MISSION Objectives Reliability Fault Tolerance Availability	SAFETY Death or Injury Damage to the Environment Damage to the System	SECURITY Corruption of System Unauthorized Access Denial of Service
RATINGS			
4 Catastrophic	mission failure	loss of life, loss of system, extreme damage to the environment	complete loss of security
3 Critical	degradation of mission	major injury, irreparable damage to the system or to the environment	some loss of security
2 Marginal	loss of secondary functions	minor injury, repairable damage to the system or to the environment	loss of some security protection
1 Negligible	inconvenience	technical violation of safety rules	technical violation of security rules
0 No involvement			

Table 4-2. Component Severity Categories and Ratings

risk as the product of the severity of an unwanted event and the likelihood of the unwanted event occurring). The Risk Score is determined for each system in which the component is, or is anticipated to be, used. This accounts for the concept of span of application as described by the WISR working group. The Risk Score is multiplied by a Longevity Score, which indicates the timeframe (short, medium, or long) in which the component is anticipated to be used by the system. The Risk Score accounts for the working group's concept of lifetime, but is determined at the system level because the timeframe of use of the component may vary by the system. The product of the Risk Score and the Longevity Score is called the Criticality Score for a component

and system. The sum of the Criticality Scores for a component across all systems in the product line is called the Criticality Score for the component, and is an indicator of the risk of using the component over its lifetime in all systems in the product line.

4.3 Marketability

The remaining concept listed by the WISR working group is marketability, which was defined as the degree to which a component would be more likely to be reused by a third party. This concept is extended to include the use of the concept in any system not included in the product line. This use includes future systems in the product line that are anticipated, but were not sufficiently defined for the component to be evaluated for severity, likelihood, and longevity. The component's Criticality Score must be adjusted not only for the number of additional systems in which its use is anticipated, but also for the variance in the risk and longevity profiles. Thus the marketability factor includes three terms:

1. the ratio of the total number of systems in which the component is anticipated for use with the number of systems in the product line,
2. the ratio of the average risk of the component across all systems to the average risk of the component in the systems in the product line, and
3. the average longevity of the component across all systems with the average longevity of the component across the systems in the product line.

If the risk and longevity profiles of the additional systems cannot be estimated, the average of the systems within the product line can be used as an estimate. Assuming that their averages are equal to the averages of the systems in the product line will eliminate the second and third terms, leaving only the ratio involving the number of systems.

4.4 Assurance

The product of the component's Criticality Score and the Marketability Score is called the Assurance Score, and indicates the level of assurance needed by this component relative to the other components in the product line. Components with a higher Assurance Score generally require a higher level of assurance than do components with a lower Assurance Score. Table 4-3 summarizes all values used to determine a component's Assurance Score, and indicates either the meaning of an input score or the method to calculate a derived score.

The values of the component Assurance Scores should be used as a guide to compare the relative level of assurance needed for components within the product line, with higher Assurance Scores indicating the need for more intensive levels of assurance. However, the association of absolute scores to distinct levels of assurance will be dependent on the domain of the product line, and on the agreement of the scoring group on the definitions of the severity ratings and the likelihood scores. Comparison of scores across different product lines is not generally meaningful, because of different assumptions and agreements by different scoring groups. In addition, factors other than the absolute score should be considered in determining the level of assurance required by a component. For example, a component that scores critical or higher in the safety severity category for any system, and has a Likelihood Score of 2 or 3, might need a relatively high level of assurance regardless of its overall Assurance Score.

4.5 Case Study Illustration

The product line assurance scoring method is illustrated using a product line case study proposed for use in academic settings [Addy 1999]. The product line consists of a number of

INPUT VALUES		
CiSjAk Severity Score	:	level of worst-case impact on the severity area in the system due to an error in the component
CiLm Likelihood Score	:	level of probability that the component has an error due to the likelihood driver
CiSj Longevity Score	:	level of time during which the component is anticipated to be used in the system
Ci Marketability Score	:	level of anticipated use of the component in systems beyond the identified systems
CALCULATED VALUES		
Ci Likelihood Score	=	AVERAGE over all Ci Likelihood Drivers
CiSj Severity Score	=	AVERAGE over all CiSj Severity Drivers
CiSj Risk Score	=	Ci Severity Score * CiSj Likelihood Score
CiSj Criticality Score	=	CiSj Risk Score * CiSj Longevity Score
Ci Criticality Score	=	SUM over all j CiSj Criticality Score
Ci Marketability Score	=	(s + additional systems using Ci) / s * AVERAGE over additional systems Severity Score / AVERAGE over j CiSj Severity Score * AVERAGE over additional systems Longevity Score / AVERAGE over j CiSj Longevity Score
Ci Assurance Score	=	Ci Marketability Score * (Ci Operation Risk Score)
	=	Ci Marketability Score * $\left(\sum_{j=1}^{S} \text{CiSj Criticality Score} * \text{CiSj Longevity Score} \right)$
	=	Ci Marketability Score * $\sum_{j=1}^{S} (\text{CiSj Severity Score} * \text{Ci Likelihood Score} * \text{CiSj Longevity Score})$

Table 4-3. Determining Relative Levels of Needed Assurance for Components in a Software Product Line

simulations for waiting queues, including queues such as a CPU dispatcher, a self-service car wash, grocery check-out counters, immigration stations, and airline check-in counters. As part of a class project, a two-person team developed a product line architecture and reusable components for the product line. The architecture, depicted in Figure 4-1, is centered on a dynamic event list as the communication vehicle between cooperating layers of objects. In addition to the event list, the primary components in the architecture are a customer generator, a customer repository, a queue facility, a server facility, a measurement recorder, and a driver to initiate the other objects and to control the simulation. The queue facility consists of a set of queue categories, where each category consists of a set of customer queues. This hierarchy enables the simulation to consider different groups of queues, such as express and normal grocery check-out counters and first-class and coach-class airline check-in counters. The server facility similarly consists of a set of server categories, each of which is a set of servers. A customer queue contains information that is needed by all waiting queues, but must be fully implemented using either a (normal) queue or a priority queue, depending on the system.

The product line assurance scoring method was applied by the author (who was one of the two developers in the class project) to determine the relative assurance levels of the components within this product line architecture. Since none of the systems has any relation to safety or security, the mission category was the only category to receive a non-zero severity rating. The scoring assumed that no component would be used in any systems other than those defined in this product line, and that all components would be used for a medium period of time. The results of the scoring are shown in the first set of scores in Table 4-4. Part 1 of Appendix A contains the exact scoring of each of the components against each system. Since the Schedule Manager and the Customer Generator are specific to the application, they do not appear in the list of product line components. The most critical components are the Customer and the Event List components, which are used in every system and are crucial to the mission in each system. The least critical component is the Priority Queue, which is used in only one system.

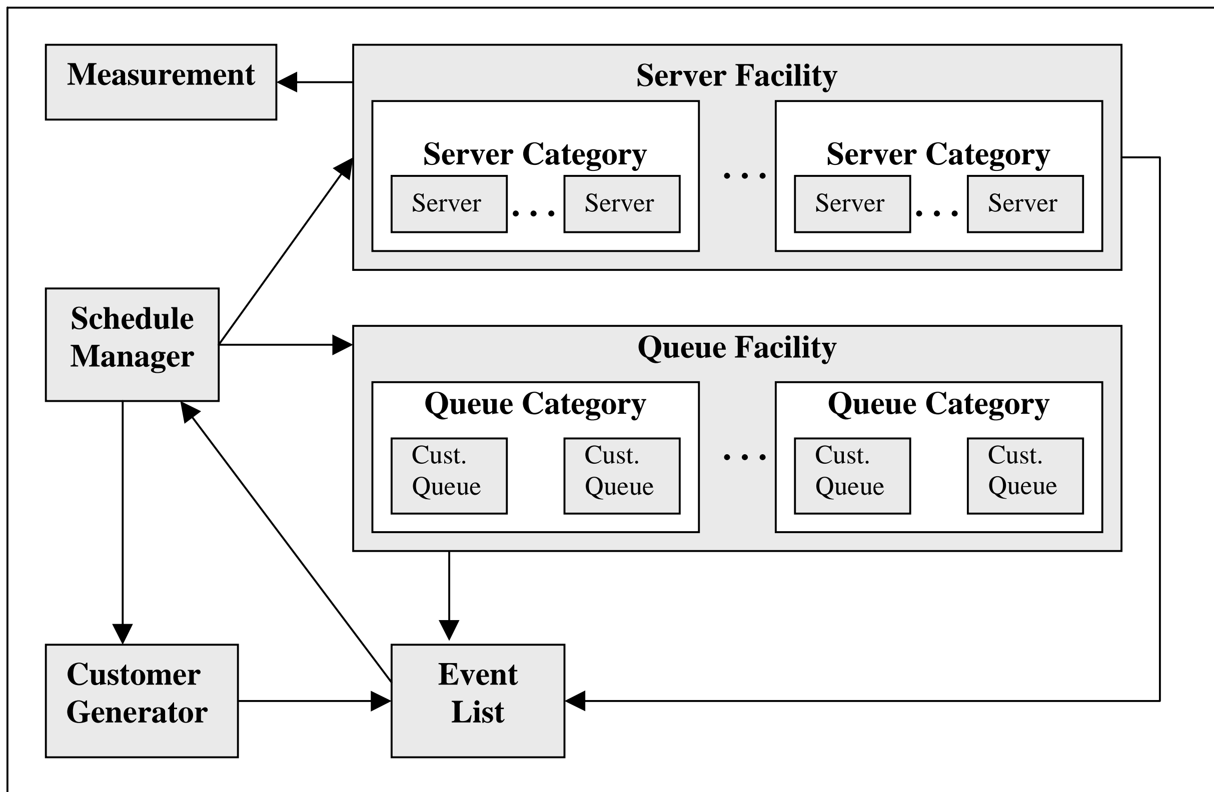


Figure 4-1. Architecture for the Waiting Queue Simulation Product Line

	Scores with no longevity or marketability assumptions			Scores with longevity and marketability assumptions		
	Criticality	Marketability	Assurance	Criticality	Marketability	Assurance
Customer	32.59	1.00	32.59	16.30	1.00	16.30
Customer Queue	19.56	1.00	19.56	22.00	1.14	25.00
Event	29.63	1.00	29.63	32.59	1.36	44.44
Event List	32.59	1.00	32.59	35.85	1.36	48.89
Measurement	29.63	1.00	29.63	32.59	1.36	44.44
Priority Queue	5.33	1.00	5.33	5.33	6.00	32.00
Queue	23.63	1.00	23.63	26.89	1.00	26.89
Queue Category	25.78	1.00	25.78	27.70	1.00	27.70
Queue Facility	25.26	1.00	25.26	28.44	1.00	28.44
Server	22.00	1.00	22.00	24.44	1.01	24.69
Server Category	27.70	1.00	27.70	30.96	1.00	30.96
Server Facility	25.26	1.00	25.26	28.52	1.00	28.52

Table 4-4. Assurance Scores for Components of the Waiting Queue Simulation Product Line

The scoring was repeated using different assumptions on marketability and longevity. These scores are shown in the second set of scores in Table 4-4, with the complete scoring shown in Part 2 of Appendix A. The assumptions used during the second scoring are shown below.

- The developer believes the Server, Customer Queue, Priority Queue, Measurement, Event and Event List components related to simulating CPU Dispatching can be marketed to five customers.
- The two check-out counter simulations will be in use for a relatively long period of time.
- The Customer component will be replaced in all systems in the near future.

Since the marketing assumption is related to the CPU Dispatching system, the marketability score is adjusted using the ratio of the score for the CPU Dispatching system divided by the average over all the systems. Specifically, the marketability score contains the following ratios:

1. $$\frac{\text{Severity score for CPU Dispatching system}}{\text{Average Severity score over all systems}}$$

2. $$\frac{\text{Longevity score for CPU Dispatching system}}{\text{Average Longevity score over all systems}}$$

Using these assumptions, the Event List, Event, and Measurement components are the most critical. The rise in assurance level of the Event and Measurement components is due to their marketability. The Priority Queue component has also experienced a sharp rise in its assurance level due to its anticipated use in other systems. The drop in the assurance level of the Customer component is due to its relative short anticipated lifespan.

This examples illustrates that the relative level of assurance of components can change not only with the introduction of new systems that use the component, but also with anticipated use in unidentified systems or with a change in the anticipated lifespan. The assurance scoring should be repeated with major product line events such as revisions to the domain architecture, the introduction of new technologies, or the addition or elimination of systems to the product line.

CHAPTER 5 – FORMAL REASONING ABOUT COTS-BASED SYSTEMS[2]

Software product lines routinely employ commercial software developed by third parties as reusable components. COTS[3] software is almost always delivered in source code or executable form, and rarely is there direct access to the requirements or design documentation. When COTS software is employed in life-critical, mission-critical, or in any system with significant financial stakes, it is essential to ensure that the integrated system is reliable.

Typically, it is neither possible nor feasible to develop complete, formal descriptions of behaviors of COTS products. But unless the behaviors are described formally, it is not possible to use rigorous techniques for specification of or reasoning about COTS-based safety-critical systems. This dissertation describes a solution to this issue that is based on complete specification of partial functionality of a COTS product, and illustrates the solution approach using a realistic case study.

A fundamental issue to be tackled in using a COTS product is the specification of the safety-critical subsystem. To the extent required in mathematical modeling and specification of the critical subsystem, features of the COTS product need to be modeled mathematically and described in mathematical interface contract(s). To perform formal reasoning or rigorous validation of the implementation of the COTS-based subsystem, programmatic interface contract(s) or specification of the commercial product or legacy system is necessary. The programmatic interfaces are similar in spirit to those advocated in [Meyer 1992] and they make use of the models in the mathematical contracts. Together, the interface contracts isolate and precisely describe those aspects of the COTS product that affect the application system. In addition to enabling unambiguous understanding and formal reasoning, precise descriptions of the interfaces also guide the testing that must be performed on the COTS product and integration testing.

The case study [Addy 1999a] that forms the basis of this chapter concerns a subsystem of the Surface Movement Advisor (SMA) system, a joint Federal Aviation Administration (FAA) and National Aeronautics and Space Administration (NASA) effort. The objective of the SMA is to assist air-traffic controllers in the area of ground movement control. SMA uses a COTS database product, among others. Our experience in specifying and implementing a subsystem of SMA demonstrates that mathematical modeling of software subsystems is possible, even in the presence of significant COTS software usage.

5.1 Surface Movement Advisor

SMA is a proof-of-concept prototype demonstration to test the implementation of advanced information systems technologies to improve coordination and planning of ground

---

[2] Portions of this chapter appeared in the Proceedings of the Symposium on Software Reusability, May 1999, and is used here by permission of the publisher. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

[3] The term COTS is used to reference commercial software developed by a third party. However, much of the discussion in this chapter applies to any software previously developed outside the application development environment, including legacy and public domain software, and other non-developmental items.

airport-traffic operations. SMA is primarily a data fusion and dissemination system, integrating airline schedules, gate information, flight plans, radar feeds, and runway configuration (including departure split and landing direction). This integrated information is shared among airport ramp operators, managers, airline operators, and FAA controllers and supervisors. As a prototype system, SMA currently is targeted for the Atlanta Hartsfield International Airport.

The subsystem of SMA considered in this case study deals with prediction of the times of key events for flights. This subsystem is responsible for determining the most likely time for flight events such as time of pushback from the gate, takeoff time, landing time, and gate arrival time.

The case study is a shadow development effort, based on the SMA Systems Requirements Document, Build 1, of SMA [SMA 1995]. The focus of the study is the "departure part" of the prediction subsystem that is concerned with predicting flight takeoff times. All departing flights leave the airspace surrounding an airport through one of several designated points, called Departure Gate Areas (DGAs). The DGAs can be conceived as horizontal tunnels in the sky through which the flight must pass. The DGA that a flight will use is determined by the destination of the flight.

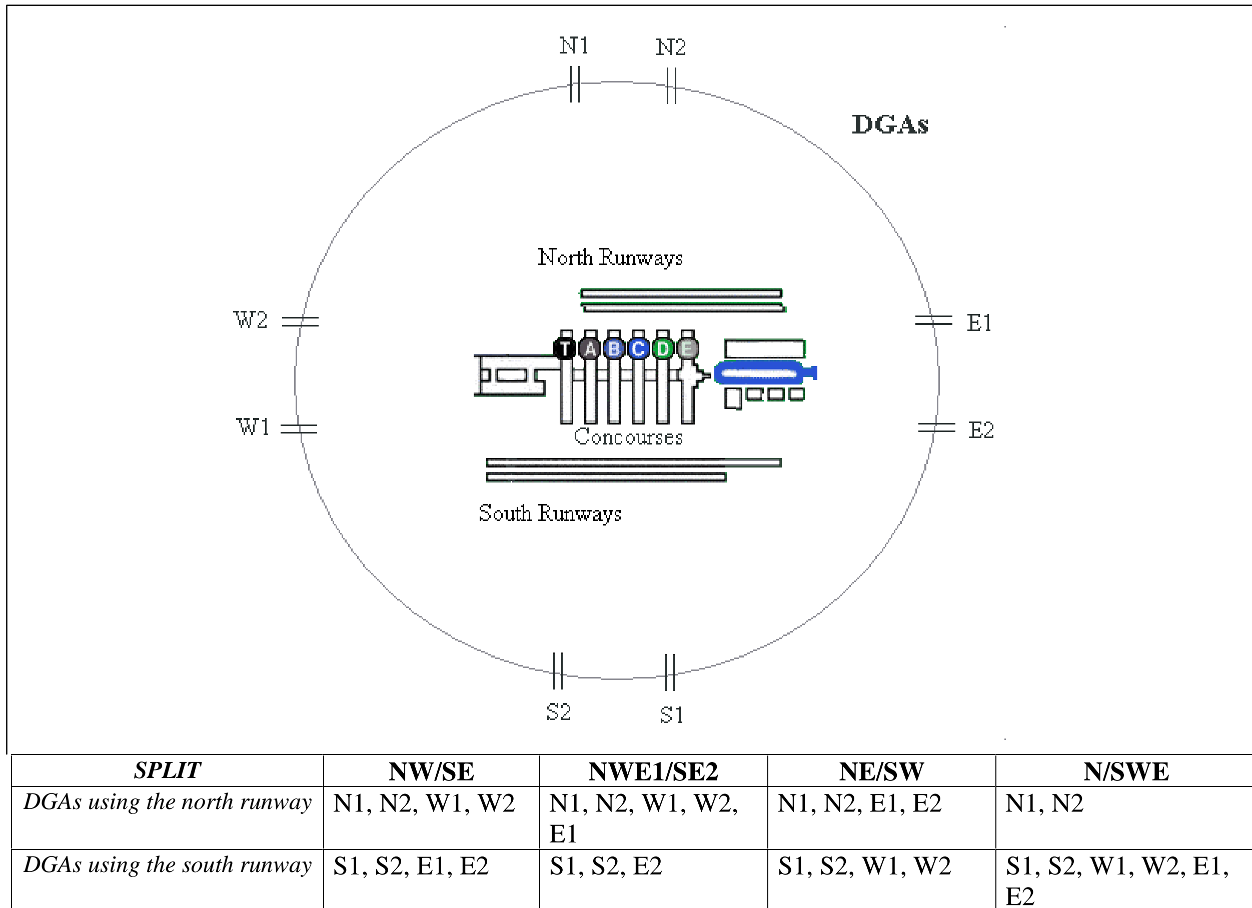
Each airport has a set of standard configurations, or "splits", that determines the runway that a flight will use based on the DGA of the flight. The airport has a number of pre-defined splits, but splits may also be created on an ad-hoc basis. The DGAs and some of the pre-defined splits at Atlanta Hartsfield International Airport are depicted in Figure 5-1. A key goal of the air-traffic controller is to minimize the time that the planes spend after pushback waiting to take off. This is done by attempting to keep the runways balanced in terms of utilization.

To make predictions on flight departures, SMA needs access to a large amount of flight and airline information. This information is managed using an Oracle database, a COTS product. The database contains (among other information) the scheduled, predicted, and actual pushback time for each flight, the call number for the plane, the aircraft type, the boarding gate, and the DGA the flight will use. The users can display the cumulative wait times for each runway, one split at a time, or can display a graphical comparison of all the splits. (The display can show the cumulative wait times for any window starting at the current time and ending 15 to 60 minutes in the future.)

5.2 Formal Specification of the Flight Takeoff Prediction Subsystem

The Prediction Subsystem is responsible for calculating the cumulative wait times and the predicted takeoff time for each flight, given the runway, the split to be used, and the time window. The focus of the problem is predicting the time of takeoff for a flight, taking into account the time the flight pushes back from the gate and considering other flights that might impact the time of takeoff.

An object-based specification for the takeoff time prediction subsystem, named Simulate_Runway_Machine_Template, is summarized in Figure 5-2, and the complete specification in RESOLVE notation [Sitaraman 1994] is contained in Appendix B. (Other formal notations, such as Larch, VDM, or Z [Wing 1990], could have been used equally well.) Simulate_Runway_Machine_Template provides a prediction simulation type, and operations to manipulate objects of the type. It is a result of "recasting" the prediction algorithm as an object. The idea of recasting graph and simulation algorithms as object-based machines, and the performance and software engineering benefits of the idea are discussed elsewhere [Weide 1994a].



SPLIT	NW/SE	NWE1/SE2	NE/SW	N/SWE
DGAs using the north runway	N1, N2, W1, W2	N1, N2, W1, W2, E1	N1, N2, E1, E2	N1, N2
DGAs using the south runway	S1, S2, E1, E2	S1, S2, E2	S1, S2, W1, W2	S1, S2, W1, W2, E1, E2

Figure 5-1. Departure Gate Areas and Several Predefined Splits for Atlanta Hartsfield International Airport (conceptual drawing, not to scale)

An object-based specification for the takeoff time prediction subsystem, named Simulate_Runway_Machine_Template, is summarized in Figure 5-2, and the complete specification in RESOLVE notation [Sitaraman 1994] is contained in Appendix B. (Other formal notations, such as Larch, VDM, or Z [Wing 1990], could have been used equally well.) Simulate_Runway_Machine_Template provides a prediction simulation type, and operations to manipulate objects of the type. It is a result of "recasting" the prediction algorithm as an object. The idea of recasting graph and simulation algorithms as object-based machines, and the performance and software engineering benefits of the idea are discussed elsewhere [Weide 1994a].

The summary of the specification shown in Figure 5-2 contains a context section showing a list of "imports" and an interface section of "exports". In the figure, the global context imports Standard_Partial_SMA_Departure_Database_Facility, because the current specification relies on the mathematical modeling of database attributes described in that module. The local context contains mathematical definitions or math operations (explained later) that make it possible to write assertions in the interface section concisely.

In the interface of this concept, objects of the type Simulate_Runway_Machine_State are modeled mathematically as a 6-tuple: sid and rid, respectively, denote the split identification and

```
concept Simulate_Runway_Machine_Template

  context

    global context

      facility Standard_ Partial_SMA_Departure_
        Database_Facility

    local context

-- mathematical definitions of
-- In_Line, Cumulative_Wait_Time_Def,
-- Safety_Delays_Met, Actual_Takeoff_Times_Used,
-- and Taxi_Times_Met

      math operation Well_Formed_Runway_Queue (
          q: string of FLIGHT_TIME_PAIR,
          sid: SPLIT_ID,
          rid: RUNWAY_NAME,
          db: SMA_DEPARTURE_DB_MODEL) : boolean
      explicit definition
        for all ft: FLIGHT_TIME_PAIR
          where (db.runway_used(ft.f) = rid or
            (db.runway_used(ft.f) = empty_string and
             db.assigned_runway
                   (sid,db.departure_gate_area(ft.f))   = rid)
          (In_Line(q, ft.f, db))
        and Safety_Delays_Met(q, rid, db)
        and Actual_Takeoff_Times_Used(q, db)
        and Taxi_Times_Met(q, rid, db)

      math operation Proper_Flights_In_Queue (
          sid: SPLIT_ID,
          rid: RUNWAY_NAME,
          tbegin: DISCRETE_TIME,
          tend: DISCRETE_TIME,
          q: string of FLIGHT_TIME_PAIR,
          db: SMA_DEPARTURE_DB_MODEL) : boolean
      explicit definition
        for all ft: FLIGHT_TIME_PAIR
          where IS_ENTRY_OF(q, ft) (
          tbegin <= ft.t <= tend)
        and
        there exists c: CONFIGURATION,
          q1: string of FLIGHT_TIME_PAIR
          where (db.c.id = sid and
            Well_Formed_Runway_Queue(q1, sid, rid, db)) (
           (Is_Substring (q, q1) )

  interface

    type Simulate_Runway_Machine_State is (
          sid: string of character,
          rid: string of character,
          tbegin: DISCRETE_TIME,
          tend: DISCRETE_TIME,
          q: string of FLIGHT_TIME_PAIR,
          ready_to_extract: boolean)
      exemplar m
      constraints
        Is_Allowed_Split_Name (m.sid) and
        Is_Allowed_Runway_Name (m.rid) and
        m.tend >= m.tbegin
      initialization ensures m.ready_to_extract = false

-- an operation to put the machine to a state able to accept
-- new values (i.e., not in extraction phase), operations to set
-- and obtain the values of the split ID, the runway ID, and
-- the beginning and ending times, and an operation to check
-- if the machine is in extraction phase.

    operation Simulate_Runway (
        alters   m: Simulate_Runway_Machine_State,
        preserves db: SMA_Database_Machine)
      requires m.ready_to_extract = false
      ensures Proper_Flights_In_Queue
                  (m.sid, m.rid, m.tbegin, m.tend, m.q,db)
        and m.sid = #m.sid and m.rid=#m.rid and
        m.tbegin=#m.tbegin and m.tend=#m.tend and
        m.ready_to_extract=true

    operation Extract_Next (
        alters   m: Simulate_Runway_Machine_State,
        produces  flight_number: Char_String,
        produces  takeoff_time: Integer)
      requires |m.q| /= 0 and m.ready_to_extract = true
      ensures #m.q = <flight_number, takeoff_time> * m.q
        and m.sid=#m.sid and m.rid=#m.rid and
        m.tbegin=#m.tbegin and m.tend=#m.tend and
        m.ready_to_extract=#m.ready_to_extract

    operation Cumulative_Wait_Time (
        preserves m: Simulate_Runway_Machine_State,
        preserves db: SMA_Database_Machine,
        produces wait_time: Integer)
      ensures Cumulative_Wait_Time =
        Cumulative_Wait_Time_Def (m.q, m.rid, db)

    operation Queue_Length_Of (
        preserves m: Simulate_Runway_Machine_State,
        produces  length: Integer)
      requires m.ready_to_extract = true
      ensures length = |m.q|

end Simulate_Runway_Machine_Template
```

Figure 5-2. Flight Departure Prediction Concept

runway identification for which simulation is to be done. tbegin and tend denote the window for simulation. The central part of the model is the field q that contains the results of the simulation; it is a string (or sequence) of ordered pairs denoting which flight is predicted to depart at what time. The purpose of the Boolean ready_to_extract in the model will become clear in the following discussion. When it is true, conceptually, results from simulation are available for extraction. The specification also states, using an exemplar object, the constraints and initialization guarantees on every object of the type. Notice that initially ready_to_extract is false. The interface provides the following operations on prediction simulation objects:

- operations to set/get simulation parameters (split, runway, start time, and end time);
- an operation to instruct the machine to simulate;
- an operation to extract the flight number and time at which the next flight is predicted to take off;
- an operation to get cumulative wait time for the current simulation; and
- status-checking operations to see if the information on the next flight can be extracted and if there are any more flights for take off in the simulated window.

In a typical use of the object, after simulation parameters are set, the Simulate_Runway operation will be called. This operation uses and preserves the database, but alters the machine state. It requires that the machine be in the insertion phase, i.e., m.ready_to_extract Boolean must be false. For the operation to work as specified in the ensures clause, the requires clause must hold when it is called.

The conceptual effect of calling the Simulate_Runway operation is that it ensures that appropriate string of flights with their actual or predicted takeoff times are now available in the prediction simulation queue “m.q”. The operation also sets the ready_to_extract flag to true indicating that the simulation results are available. In the ensures clause, #m denotes the value of parameter m before the call and m denotes its value after the call.

The effect of the Simulate_Runway operation has been specified formally using a mathematical definition Proper_Flights_In_Queue. This definition uses another definition, Well_Formed_Runway_Queue, that specifies when a mathematical string (or sequence) of flight/take-off time pair is a valid simulation, based on database information such as push back times, taxi times, and runway delays. The definition Proper_Flights_In_Queue is additionally concerned with a given window of time. Both of these definitions, as well as others not listed explicitly in the figure, are based on departure database information. The relevant database details are contained in db, modeled by SMA_DEPARTURE_DB_MODEL. Details of this model are the topic of the next subsection.

The Extract_Next operation requires that the results be ready for extraction, and it produces the predicted values of the next flight number and associated takeoff time. Cumulative_Wait_Time operation returns the total wait time of the flights in line (as specified formally in the mathematical operation Cumulative_Wait_Time_Def). The Queue_Length_Of operation returns the number of flights left in the prediction simulation window of time.

5.3 Mathematical Modeling of Aspects of A COTS Database

It is clear that the specification in Figure 5-2 can be meaningful only if there is a suitable mathematical modeling of database information. The mathematical description of SMA_DEPARTURE_DB_MODEL should contain all information relevant for predicting take-off times, but nothing more. Such a description, termed a mathematical interface contract, is

```
mathematics SMA_Database_Math_Machinery
  context
    global context
      mathematics SMA_Database_Airport_Information
  interface
    math subtype FLIGHT_ID is string of character
      exemplar fid
      constraint Is_Allowed_Flight_ID(fid)

-- similar math subtypes for AIRCRAFT_TYPE_NAME,
-- GATE_NAME, OPTIONAL_GATE_NAME, DGA_NAME,
-- RUNWAY_ID, OPTIONAL_RUNWAY_ID,
-- and SPLIT_ID

    math subtype GATE_RUNWAY_PAIR is (
        g: GATE_NAME,
        r: RUNWAY_ID )

    math subtype CONFIGURATION is (
        sid: SPLIT_ID,
        split: function from DGA_NAME to RUNWAY_ID)

    math subtype FLIGHT_TIME_PAIR is (
        f: FLIGHT_ID,
        t: DISCRETE_TIME)

    math subtype SMA_DEPARTURE_DB_MODEL is (
        aircraft_type: function from FLIGHT_ID to
          AIRCRAFT_TYPE_NAME
        gate: function from FLIGHT_ID to
          OPTIONAL_GATE_NAME
        departure_gate_area: function from FLIGHT_ID to
          DGA_NAME
        runway_used: function from FLIGHT_ID to
          OPTIONAL_RUNWAY_ID
        predicted_pushback_time: function from FLIGHT_ID
          to DISCRETE_TIME
        actual_pushback_time: function from FLIGHT_ID to
          OPTIONAL_DISCRETE_TIME
        actual_takeoff_time: function from FLIGHT_ID to
          OPTIONAL_DISCRETE_TIME
        delay_time: function from AIRCRAFT_TYPE_NAME
          to DISCRETE_TIME
        roll_time: function from AIRCRAFT_TYPE_NAME
          to DISCRETE_TIME
        taxi_time: function from GATE_RUNWAY_PAIR to
          DISCRETE_TIME
        configuration: set of CONFIGURATION )

end SMA_Database_Math_Machinery
```

Figure 5-3. Mathematical Modeling of Database Information

given in the interface of the mathematics module SMA_Database_Math_Machinery in RESOLVE notation in Figure 5-3.

Unlike a concept specification, such as the one in Figure 5-2, that provides program types and operations to manipulate programming objects, the purpose of a mathematics module is to define mathematical types and definitions useful for writing specifications. Mathematics modules are not implemented. They simply establish formal meanings for domain vocabulary to be used in other specifications. They are similar in spirit to Larch traits [Wing 1990].

In Figure 5-3, SMA_DEPARTURE_DB_MODEL is defined to be a math subtype. A math subtype is essentially a base mathematical type with zero or more constraints on the value space [Heym 1998, Rushby 1998]. In the definition of SMA_DEPARTURE_DB_MODEL, other math subtypes have been defined and used, though most of them have not been shown. This database model consists of a collection of functions needed in the prediction system. For example, predicted_pushback_time is a function from the character string FLIGHT_ID into DISCRETE_TIME. In this case, the base mathematics type for DISCRETE_TIME is integer, where the values are constrained to be non-negative integers.

Some of the information in the database is particular to the specific airport. This information has been isolated and specified separately in another mathematics module SMA_Database_Airport_Information (not shown). It contains information such as airport-specific gate names, flight identifications, and standard splits, and it is used to isolate airport-specific information to a separate module. Together the mathematical modules define *mathematical interface contracts* of the COTS product. For the prediction system, any database can be used as long as it contains at least the information corresponding to the mathematical modeling in Figure 5-3. The example illustrates that in general, formal specification of COTS-

based systems may require selected aspects of the COTS products to be modeled mathematically and captured formally.

5.4 Modular Reasoning about a COTS-Based Subsystem

This section describes a component-based implementation of the prediction subsystem. To show that this implementation meets its specification, and for the reasoning process to be *modular*, only on the specifications of reused components are essential [Ernst 1991, Leavens 1991, Weide 1994b]. To be able to reason in a modular fashion about a COTS-based implementation, it is essential to have a programmatic interface contract(s) or specifications for relevant parts of the COTS product. This is the topic of this section.

Figure 5-4 provides an illustration to explain the basic premise of modular reasoning. In the figure, an oval represents the specification of a component, whereas a rectangle represents an implementation. A thin arrow labeled "i" indicates an implements relationship and a thick arrow labeled "u" indicates a *uses* relationship. In order to verify that the implementation in the figure satisfies its specification, only the specifications of reused components (numbered 1, 2, and 3) are needed. No knowledge of the rest of the system in which the component will be used is necessary and no knowledge of details of implementations of reusable components is necessary. Modular reasoning essentially makes it possible to reason about one implementation at a time, and is therefore, scalable. This basic idea is independent of whether the reasoning process is formal or informal, automated or manual.

Figure 5-5 shows the structure of a COTS-based implementation of the flight departure prediction subsystem. The implementation uses the specification of a part of the (ideal, complete) COTS database specification, as well as other supporting specifications. As shown in the figure, the database interface has been separated into two concepts, one that specifies basic database retrieval operations and another that specifies a highly application-specific extensive database query. These interfaces need to be implemented based on the COTS software, and they serve as the contract(s) between the prediction subsystem and COTS database. (Math models used in the contracts are not explicitly shown in the figure.)

Figures 5-6 and 5-7 contain programmatic database interface contracts (or specifications) for Partial_SMA_Departure_Database_ Template and SMA_Database_Flight_Finder_Template.

The specification in Figure 5-6 makes use of the models in the mathematical unit shown in Figure 5-3 in the previous section. The global context of the figure explains this linkage by referring to SMA_Database_Math_Machinery. In the interface section, the Database object is modeled by SMA_DEPARTURE_ DB_MODEL. The operations specify basic database retrieval operations.

The other part of the Database interface, SMA_Database_Flight_Finder_Template, is shown in Figure 5-7. This interface specifies an extended database query, Select_Flights_To_Runway, for simulating the runway. The query locates all flights that either have used the specified runway or will use the runway based on the specified split, and orders them by takeoff time (if they have already departed) or time of pushback from the gate (actual pushback times followed by predicted pushback times). The exported type is the conceptual value of the result of this selection query. The other two operations are used to return the next flight ID from the ordered selection, and to check on the number of flights remaining from the selection.

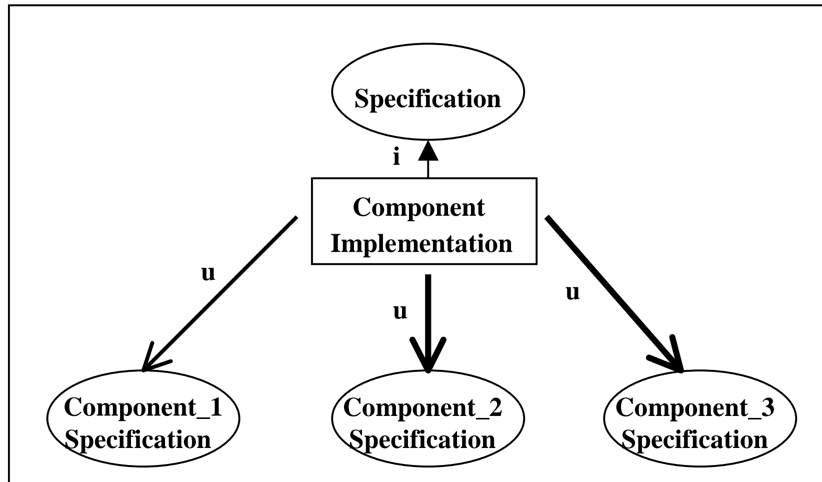


Figure 5-4. Illustration of Modular Reasoning

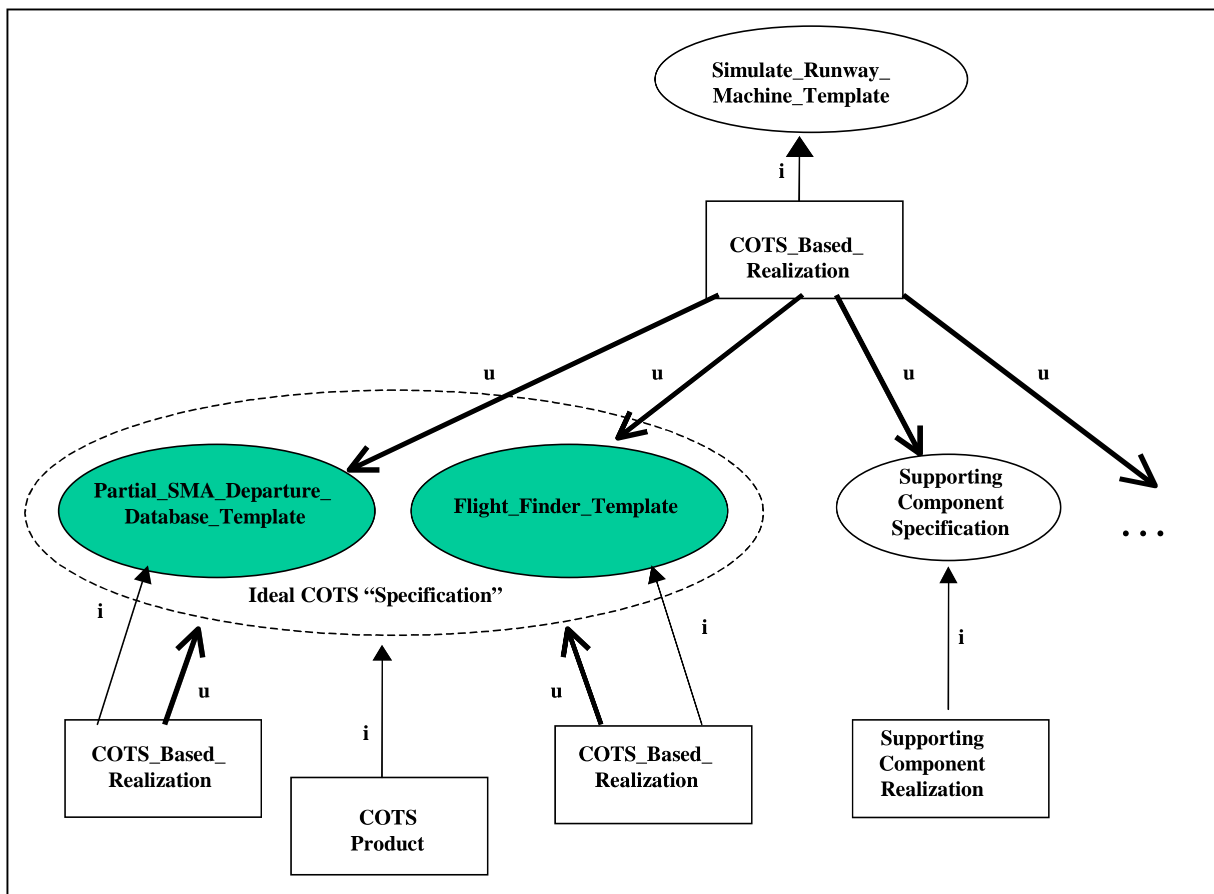


Figure 5-5. Modular Reasoning for the SMA Flight Departure Prediction Algorithm

```
concept Partial_SMA_Departure_ Database_Template

  context global context

    mathematics SMA_Database_Math_Machinery

  interface
    type SMA_Database is modeled by
      SMA_DEPARTURE_DB_MODEL
      exemplar db

    operation Aircraft_Type (
        preserves db: SMA_Database_Machine,
        preserves fid: Char_String): Char_String
      requires Is_Allowed_Flight_ID (fid)
      ensures db.aircraft_type(fid) = Aircraft_Type

    operation Taxi_Time (
        preserves db: SMA_Database_Machine,
        preserves gid: Char_String,
        preserves rid: Char_String): Integer
      requires Is_Allowed_Flight_ID (fid)
        and Is_Allowed_Gate_Name (gid)
      ensures db.taxi_time ((gid, rid)) = Taxi_Time

-- similar operations to obtain the Gate, Runway_Used,
-- Departure_Gate_Area, Predicted_Pushback_Time,
-- Actual_Pushback_Time, and Actual_Takeoff_Time
-- associated with a flight ID; the Delay_Time and
-- Roll_Time associated with an aircraft type; and the
-- Assigned_Runway associated with a split and DGA

end Partial_SMA_Departure_Database_Template
```

Figure 5-6. Database Interface Contract (part 1)

```
concept SMA_Database_Flight_Finder_Template

  context

    global context

      facility Standard_ Partial_SMA_Departure_
        Database_Facility

    local context
      math operation
        All_Flights_To_Runway_In_Pushback_Order (
          q: string of FLIGHT_ID,
          sid: SPLIT_NAME,
          rid: RUNWAY_NAME,
          db: SMA_DATABASE_MACHINE): boolean
      explicit definition
        all the flights that have taken off from the specified
        runway or that are assigned to the specified runway
        by the split are  in the string, and the flights are in
        order of (by precedent)
          (1) actual takeoff time (if the flight has taken off),
          (2) actual pushback time
              (if the flight has pushed back from the gate but
               has not taken off),
          (3) predicted pushback time
              (if the flight has not pushed back from the gate).

  interface
    type Flight_Finder_State is modeled by (
        q: string of FLIGHT_ID)
      exemplar ff
      initialization ensures    |ff.q| = 0

    operation Select_Flights_To_Runway (
        alters   ff: Flight_Finder_State,
        preserves sid: Char_String,
        preserves rid: Char_String,
        preserves db: SMA_Database_Machine)
      requires  Is_Allowed_Split_Name (sid) and
        Is_Allowed_Runway_Name (rid)
      ensures
        All_Flights_To_Runway_In_Pushback_Order
        (ff.q, sid, rid, db)

    operation Get_Next_Flight_To_Runway (
        alters   ff: Flight_Finder_State,
        produces  fid: Char_String)
      requires   |ff.q| > 0
      ensures #ff.q = <fid> * ff.q

    operation Number_Of_Flights_To_Runway (
        preserves ff: Flight_Finder_State,) : Integer
      ensures    Number_Of_Flights_To_Runway = |ff.q|

end SMA_DB_Departure_Get_Flights_Template
```

Figure 5-7. Database Interface Contract (part 2)

Together, the interfaces in Figures 5-6 and 5-7 accurately capture programmatic aspects of the COTS product, relevant to the SMA departure prediction subsystem.

5.5 Status

The specification for the SMA subsystem has been implemented in RESOLVE/C++ [Hollingsworth 1994]. The implementations of the database concepts depend on the particular database product and its structure, and consist primarily of Structured Query Language (SQL) queries.

The shadow development effort, on which this case study is based, uses an mSQL database rather than the Oracle database that is used by the developer. The implementations for an mSQL database have substantial differences from implementations using Oracle, but each implementation should meet the mathematical and programmatic interface contracts. A change in the choice of the underlying COTS database affects only the implementations of the database interface concepts. No other specifications or implementations will be impacted by changing the COTS package, or by changing the physical storage within the database instance.

A separate part of the application instantiates Simulate_Runway_Machine_Template, sets the runway, split, and times, and then obtains cumulative wait time for the runway and the predicted takeoff times for the flights. This part of the software is responsible for storing the data as necessary in the database and for displaying the data (hence there are no operations in these concepts to store or display data). The concepts and implementations are suitable for any airport. The only changes are relegated to an airport-specific data module.

5.6 Lessons Learned

The specifications described in this chapter are a result of an iterative process. An important objective of the overall effort is to ensure that the process and the results are more generally applicable than to the particular case study.

There is an interactive nature between the specification of the application and the specification of the (aspects of interest of the) COTS product. In the case study, the objective was to perform a mathematical modeling of the application subsystem, with the focus on determining the wait time of the line of flights at a runway. The COTS product (or rather, the nature of a generic COTS database) provided restrictions on the mathematics model. Hence, the mathematical model was developed so that the application used specific known capabilities of the COTS product, and the model served to formally specify the aspects of the COTS product that were needed by the application subsystem. This interaction between the application subsystem and the COTS product to be used appears to be a natural process; it seems unlikely that specifications would be developed in anticipation that a COTS product would be found to meet them.

In the case study, we followed a spiral process model, beginning with the specification of the application concept Simulate_Runway_Machine_Template. The mathematics module(s) were next in order, to define the basic subtypes and the functions of the database. The programmatic database interface contracts were the last concepts to be developed. The refinements continued until the specifications were suitable for developing the COTS-based subsystem.

The mathematical units are separated into reusable and application/airport-specific parts. Similarly, programmatic database interface contracts are split into two portions for ease in development and for anticipated reuse. The Partial_SMA_Departure_Database_Template specification is used to describe basic data retrieval operations for a database, whose form is

unlikely to change with modifications to either the application or the COTS database. This specification may be used in the specification of other parts of the SMA system with changes only by the addition of other retrieval operations. The SMA_Database_Flight_Finder_Template specification describes a specific query that is unique to the subsystem being specified, which is more likely to be modified and less likely to be of use in other parts of the system.

The implementation of the specifications in C++, using the mSQL database, was a straight forward effort following completion of the specifications. One interesting problem arose in the implementation, when testing revealed that the ordering clause of the selection operation did not work on non-numeric fields of the database (at least in the version of the database being used in the case study). The implementation of the database interface had to be modified to include a sorting algorithm. This is an example of how differences in a particular COTS product may affect the interface implementation, but the effect is localized, and therefore, does not affect other specifications or implementations.

CHAPTER 6 – CONCLUSIONS AND FUTURE WORK

This thesis has discussed the extension of the V&V process from the development of a single application system to the development of a product line of systems. The framework provides an overall structure for V&V activities in domain engineering, application engineering, and in the transition between domain and application engineering. The product line assurance scoring method allows the determination of the relative levels of assurance needed by the components within the product line, so that appropriate V&V resources can be given to the various components. Full specification of partial functionality of COTS products provides a method to formally reason about systems within the product line that are based on COTS products, a method to isolate the system from the particular brand and version of the COTS product, and a guide to testing the COTS products.

The framework for performing V&V within software product line engineering allows some V&V activities to be performed prior to the development of an individual application system, and hence provides the opportunity for finding and correcting errors earlier. This framework allows the V&V effort to be amortized over the systems within the product line. Just as with V&V in application system development, V&V should be performed as part of an overall risk mitigation strategy within the product line.

The product line assurance scoring method considers the operation risk of each component within each system in the product line, the length of time that the component is anticipated to be in use, and the potential use of the component in systems not yet identified. These considerations allow a determination of the relative level of assurance needed by the component, so that the V&V effort on the component can be scoped to the proper level of intensity.

Ideally COTS software products would have formal specifications of behavior that can be used in establishing the correctness of systems built using the products. COTS products of today do not have such specifications due to research, technical, and economic limitations. Given this state of practice, this thesis illustrates using a realistic case study how formal methods can be applied to COTS-based safety-critical systems. In specifying and implementing a product line of systems based on a COTS product, the thesis explains that it is sufficient to describe only aspects of the reused product that are relevant to the use of the product within the domain architecture. To the extent the COTS product is reliable, the approach is useful to ensure that the interactions between the individual systems and the COTS products are valid.

Description of interactions involves mathematical modeling of aspects of the COTS products as well as formal specification of programmatic interfaces based on those models. Some aspects of the modeling and some of the interface specifications are reusable in other applications, and hence should be isolated into separable units. . The interface contracts isolate the system under development from the COTS software. Changes and improvements to COTS software products have no impact on the system; where there is an impact it is seen through the interfaces. The interface contracts facilitate formal reasoning and also serve as a guide to the test process to validate the use of COTS software. These results are generally applicable though this dissertation has used a database COTS product as the central example.

The framework presented in this dissertation for performing V&V in software product line engineering is currently an outline with few details. V&V tasks that are currently performed at the application level need to be adapted for the domain level, and traceability tasks need to be adapted for the transition level. New methods not used on applications but appropriate for

domain models or architectures need to be considered. Since V&V should be performed as part of an overall risk mitigation strategy within the domain or product line, methods of domain criticality analysis need to be developed, with attention paid to support from emerging architecture description languages. The methods identified need to be validated by use in projects having an architecture-based software engineering approach to producing applications that require V&V.

The implementation of product line software engineering not only affects the process of V&V, it also impacts other software engineering support activities. These other areas of software engineering that are affected include Configuration Management, Testing, Quality Control, and certification. Activities in each of these areas must be adapted to address the entire domain or product line rather than a specific application system.

Software CM contains functions related to identification, management, change control, status, reviews, and deliveries of software systems. It is the focal point for managing and controlling the variants of the many types of software artifacts, both during initial development and in the subsequent evolution during deployment of the software systems. The existence of a systematic software reuse environment that is used to produce a product line of software systems increases the demands on a CM program by an order of magnitude, because the software components will now be used in multiple systems rather than in an individual system. For every version of a system, the CM process will need to track the version of each component used in the development of that particular system, along with all the dependencies of that version of the component with other components in the system [Bosch 1999]. The increase in complexity of the CM process for reuse-based software engineering is based on two considerations [Edwards 1997]:

- Any change to a component must be considered in terms of all systems which use (or may potentially use) the component.
- Multiple versions of a component must be stored, maintained, and available for use, due to compatibility issues with other components.

Testing is affected in much the same way that V&V is affected. Test cases for reusable components must consider all systems for which the component is intended. As a minimum, careful representation of the testing that was performed must be recorded to indicate exactly how the component was tested. Test documentation from domain level testing and from testing within individual systems pertaining to a component should be available to system developers, so that they may determine if further testing for a particular component is necessary for individual systems. This implies a need for uniform test documentation throughout the product line, including test cases, test inputs, and test results. There should be little impact on integration and system testing, as this activity is system specific.

The QA process ensures that the software development (and software evolution) process and the software products conform to established standards. This process normally uses activities such as reviews, inspections, and reviews to ensure conformance, rather than the level of analysis and testing performed by a V&V process or a testing process. Most often the QA effort is directed more toward process than product [Glass 1992]. Obviously the QA process must adapt to any change in the development process, and so it must to the implementation of a reuse-based software engineering environment. QA must ensure that the development is well-defined, documented, and followed. While this is not a new to QA, what differs is that there are now two processes: the domain or product line process and the system process. The two processes interact but have different goals and products. QA must ensure not only that the

individual processes are defined and followed, but that the process interaction is defined and followed as well.

The IEEE standard definition for *certification* is "the process of confirming that a system or component complies with its specified requirements and is acceptable for operational use" [IEEE 1990]. (Actually, this is the third of three definitions, where the first two address a written guarantee and a formal demonstration, respectively, of this confirmation.) The confirmation of compliance of an item with its specification can be performed by assessing some combination of the personnel developing the item, the process by which the item is developed, or the quality attributes of the item itself [Voas 1998]. The concept of software certification is in itself a controversial topic, with debate on certification of software engineers, the relationship between a good process and a good product, and the ability to assess the quality attributes of software products. These issues apply to software that is developed under any methodology, including reuse-based software engineering. However, reuse-based development introduces further issues into the certification process. The reliance on previously developed components (often Commercial-Off-The-Shelf components) means that the quality of the developers and the development process of the component may be unknown, leaving product certification as the only approach. Needing to understand the implementation of each component removes much of the advantage of incorporating reusable components in a black-box manner, through an understanding of the interface and functional descriptions. Even if the component is thoroughly understood, knowing and describing the exact conditions under which the component will be validated (and hence the conditions in which the component may be safely used) is a daunting task. The legal implications on warranty and liability from certification of a reusable component or of a system developed with reusable components are not as yet resolved. These implications can impact the developer of the component, the (re)user of the component (who is the developer of a software system), and the acquirer of the system.

Much work has been performed in defining processes and methods for developing reusable components and for developing systems using reusable components. These activities have extended into development support areas such as reuse libraries and component evaluation. This work is appropriate, but is insufficient for the purpose of commercial-level software development. To achieve the complete implementation of a systematic reuse-based software engineering process, all aspects of software engineering must be considered, including the support services such as V&V, CM, QA, testing, and certification. The issues related to software support activities must be identified and resolved in order for industrial-level software reuse to occur.

REFERENCES

Abowd 1997 — Gregory Abowd, et. al., "Recommended Best Industrial Practice for Software Architecture Evaluation," CMU/SEI-96-TR-025, Software Engineering Institute, Carnegie Mellon University, January 1997.

Addy 1999a — Edward A. Addy and Murali Sitaraman, "Formal Specification of COTS-Based Software: A Case Study," Proceedings of the Fifth Symposium on Software Reusability (Los Angeles CA USA, May 1999), ACM Press, 83-91.

Addy 1999b — Edward A. Addy, Ali Mili, and Sharif Yacoub, "A Case Study in Software Reuse," NASA/WVU Software Research Laboratory, Technical Report #NASA-IVV-99-002 (to be published in Empirical Methods in Software Engineering).

Addy 1998a — Edward A. Addy, "Report from the First Annual workshop on Software Architectures in Product Line Acquisitions," Software Engineering Notes, 23, 3 (May 1998), 32-38.

Addy 1996 — Edward A. Addy (1996), "V&V Within Reuse-Based Software Engineering", Proceedings of the Fifth Annual Workshop on Software Reuse Education and Training, Reuse '96, http://www.asset.com/WSRD/conferences/proceedings/results/addy/addy.html.

Addy 1998b — Edward A. Addy, "A Framework for Performing Verification and Validation in Reuse-Based Software Engineering," Annals of Software Engineering, Vol. 5 (1998), 279-292.

ARES 1997 — ARES Project (ESPRIT framework IV), Electronic Proceedings of the International Workshop on Development and Evolution of Software Architectures for Product Families, November 1996, http://hpv17.infosys.tuwien.ac.at/Projects/ARES/public/AWS/ .

Barbacci 1997 — Mario R. Barbacci, Mark H. Klein, and Charles B. Weinstock, "Principles for Evaluating the Quality Atrributes of a Software Architecture, CMU/SEI-96-TR-036, Software Engineering Institute, Carnegie Mellon University, May 1997.

Bass 1998a — Len Bass, et. al., Second Product Line Practice Workshop Report, CMU/SEI-98-TR-015, Software Engineering Institute, Carnegie Mellon University, April 1998.

Bass 1998b — Len Bass, Paul Clements, and Rick Kazman, Software Architecture in Practice, Addison-Wesley, Reading, MA, 1998.

Bass 1997 — Len Bass, et. al., Product Line Practice Workshop Report, CMU/SEI-97-TR-003, Software Engineering Institute, Carnegie Mellon University, June 1997.

Boehm 1976 — B. W. Boehm, "Software Engineering," IEEE Transactions on Computing, C-25 (December 1976), 1226-1241.

Bosch 1999 — Jan Bosch, "Product-Line architectures in Industry: A Case Study," Proceedings of the 21st International Conference on Software Engineering, (Los Angeles CA USA, May 1999), ACM Press, 544-554.

Callahan 1996 J. Callahan and G. Sabolish, "A Process Improvement Model for Software Verification and Validation," Journal of the Quality Assurance Institute, 10, 4 (October 1996), 24-32.

Carney 1997 D. Carney and P. Oberndorf, "The Commandments of COTS: Still in Search of the Promised Land," Crosstalk, 10, 5 (May 1997), 25-30.

Challenge 2000 Challenge 2000 Subcommittee of the FAA Research, Engineering, and Development Advisory Committee, "Use of COTS/NDI in Safety-Critical Systems," February 1996.

Chen 1997 Y. Chen and B. H. C. Cheng, "Formalizing and Automating Component Reuse," Proceedings of the 9th International Conference on Tools with Artificial Intelligence (Newport Beach CA USA, November 1997), IEEE Computer Society Press, 94-101.

Clements 1998 Paul C. Clements and Nelson Weiderman, Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families, CMU/SEI-98-SR-003, Software Engineering Institute, Carnegie Mellon University, May 1998.

DeBaud 1997 Jean-Marc DeBaud, "Towards a Customizable Domain Analysis Framework: Initial Lessons from the Field," Proceedings of the European Reuse Workshop 1997 (November 1997, Brussels, Belgium), 112-115.

Duke 1989 Eugene L. Duke, "V&V of Flight and Mission-Critical Software", IEEE Software, 6, 3 (May 1989), 39-45.

Dunham 1989 Dunham, Janet R., "V&V in the Next Decade", IEEE Software, 6, 3 (May 1989), 47-53.

Dunn 1993 Michael F. Dunn and John C. Knight (1993), "Certification of Reusable Software Parts," Technical Report CS-93-41, University of Virginia, Charlottesville, VA.

Edwards 1997 Stephen H. Edwards and Bruce W. Wiede, "WISR8: 8th Annual Workshop on SW Reuse", Software Engineering Notes, 22, 5 (September 1997), 17-32.

Ernst 1991 G. W. Ernst, et. al., "Modular Verification of Ada Generics," Computer Languages, 16, 3/4 (March/April 1991), 259-280.

Garlan 1996 David Garlan and Mary Shaw, Software Architecture: An Emerging Discipline, Printice Hall, 1996.

Garlan 1995 David Garlan (ed.), "First International Workshop on Architectures for Software Systems Workshop Summary", Software Engineering Notes, 20, 3 (July 1995), 84-89.

Glass 1992 Robert L. Glass, Building Quality Software, Prentice Hall, Englewood Cliffs, NJ, 1992.

Goguen 1996 Joseph A. Goguen, "Parameterized Programming and Software Architecture," Proceedings of the Fourth International Conference on Software Reuse (Orlando FL USA, April 1996), IEEE Computer Society Press, 2-10.

Heym 1998 W. Heym, et. al., Mathematical Foundations and Notation of RESOLVE, Tech Report OSU-CISRC-8/94/TR45, revised September 1998, Department of Computer and Information Science, The Ohio State University, Columbus, OH.

Hollingsworth 1994 J. E. Hollingsworth, et. al., "RESOLVE Components in Ada and C++," Software Engineering Notes, 19, 4 (October 1994), 52-63.

IEEE 1990 IEEE STD 610.12-1990, IEEE Standard Glossary of Software Engineering Technology, Institute of Electrical and Electronics Engineers, Inc., New York, NY.

IEEE 1992 IEEE STD 1012-1986 (R 1992), IEEE Standard for Software Verification and Validation Plans, Institute of Electrical and Electronics Engineers, Inc., New York, NY.

IEEE 1993 IEEE STD 1059-1993, IEEE Guide for Software Verification and Validation Plans, Institute of Electrical and Electronics, Inc., New York, NY.

NASC 1989 Tomahawk Nuclear Safety and Certification Program, PDA-14INST 8020.1A, Naval Air Systems Command, Washington, DC, November 1989.

Jacobson 1997 Ivar Jacobson, Martin Griss, and Patrik Jonsson, Software Reuse: Architecture, Process and Organization for Business Success, ACM Press, New York, NY, 1997.

Jeng 1994 J. Jeng and B. H. C. Cheng, "A Formal Approach to Reusing More General Components," Proceedings of the 9th Knowledge-Based Software Engineering Conference (Monterey CA USA, September 1994), IEEE Computer Society Press, 90-97.

Kazman 1995 Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements, "Scenario-Based Analysis of Software Architecture," IEEE Software, 13, 6, (1995) 47-55.

Kitchenham 1998 Barbara Kitchenham and Stephen Linkman, "Validation, Verification, and Testing: Diversity Rules," IEEE Software, 15, 4 (July/August 1998), 46-49.

Knight 1998 John C. Knight and Michael F. Dunn, "Software Quality through Domain-driven Certification," Annals of Software Engineering, Vol. 5 (1998), 293-315.

Leach 1997 R. J. Leach, Software Reuse, McGraw-Hill, New York, 1997.

Leavens 1998 G. T. Leavens, O. Nierstrasz, and M. Sitaraman, "1997 Workshop on Foundations of Component-Based Systems," Software Engineering Notes, 23, 1 (January 1998), 38-41.

Leavens 1991 G. T. Leavens, "Modular Specification and Verification of Object-Oriented Programs", IEEE Software, 8, 4 (July 1991), 72-80.

Lewis 1992 Robert O. Lewis, Independent Verification and Validation, A Life Cycle Engineering Process for Quality Software, John Wiley & Sons, New York, NY, 1992.

Lutz 1999 Robyn R. Lutz, "Extending the Product Family Approach to Support Safe Reuse," Proceedings of the Fifth Symposium on Software Reusability (Los Angeles CA USA, May 1999), ACM Press, 17-26.

Madachy 1997 Raymond J. Madachy, "Heuristic Risk Assessment Using Cost Factors," IEEE Software, 14, 3 (May/June 1997), 51-59.

Makowsky 1992 Lawrence C. Makowsky, "A Guide to Independent Verification and Validation of Computer Software," USA-BRDEC-TR//2516, United States Army Belvoir Research, Development and Engineering Center, Fort Belvoir, VA, 1992.

McCaugherty 1998 Dan McCaugherty, "Criticality Analysis and Risk Assessment (CARA)," presentation at West Virginia University, February 1998.

Meyer 1992 Bertrand Meyer, "Applying 'Design by Contract'," Computer, 26, 10 (October 1992), 40-51.

Musa 1989 John D. Musa and A. Frank Ackerman, "Quantifying Software Validation: When to Stop Testing?" ", IEEE Software, 6, 3 (May 1989), 19-27.

NASA 1989 Software Assurance Guidebook, National Aeronautics and Space Administration Software Assurance Technology Center, 1989.

Oberndorf 1997 P. A. Oberndorf, et. al., Workshop on COTS-Based Systems, Software Engineering Institute, CMU/SEI-97-SR-109, November 1997.

Poore 1992 J.H. Poore, Theresa Pepin, Murali Sitaraman, and Frances L. Van Scoy, "Criteria and Implementation Procedures for Evaluating Reusable Software Engineering Assets," DTIC AD-B166803, prepared for IBM Corporation Federal Sectors Division, Gaithersburg, MD, 1992.

Ran 1999 Alexander Ran, "Software Isn't Build from Lego Blocks," Proceedings of the Fifth Symposium on Software Reusability (Los Angeles CA USA, May 1999), ACM Press, 164-169.

Rushby 1998 J. Rushby, S. Owre, and N. Shankar, "Subtypes for Specifications: Predicate Subtyping in PVS," IEEE Transactions on Software Engineering, 24, 9 (September 1998), 709-720.

Sisti 1994 Frank J. Sisti and Sujoe Joseph, Software Risk Evaluation Method, Version 1.0, Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-94-TR-19, December 1994.

Shere 1998 Kenneth D. Shere, Software Engineering and Management, Prentice Hall, Englewood Cliffs, NJ, 1998.

Sitaraman 1994 M. Sitaraman and B. Weide, "Component-Based Software using RESOLVE," Software Engineering Notes, 19, 4 (October 1994), 21-67.

SMA 1995 Surface Movement Advisor Build 1 System Requirements Document, SMA-110, National Aeronautics and Space Administration, Ames Research Center, June 1995.

SPS 1996 Software Productivity Solutions, Inc., "Certification of Reusable Software Components, Volume 2 – Certification Framework," prepared for Rome Laboratory/C3CB, Griffiss AFB, NY, 1996.

Tracz 1996 Will Tracz, "Test and Analysis of Software Architectures," Proceedings of the International Symposium on Software Testing and Analysis (San Diego CA USA, January 1996), ACM Press, 1-3.

Unisys 1994 Unisys, Valley Forge Engineering Center, and EWA, Inc., "Component Provider's and Tool Developer's Handbook," STARS-VC-B017/001/00, prepared for Electronic Systems Center, Air Force Material Command, USAF, Hanscom AFB, MA, 1994.

Voas 1998 J. M. Voas, "Certifying Off-the-Shelf Components," Computer, 31, 6 (June 1998), 53-59.

Wallace 1996 Dolores R. Wallace, Laura M. Ippolito, and Barbara B. Cuthill, Reference Information for the Software Verification and Validation Process, NIST Special Publication 500-234, National Institute of Science and Technology, Gaithersburg, MD, 1994.

Wallace 1989a Dolores R. Wallace and Roger U. Fujii, Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards," NIST Special Publication 500-165, National Institute of Standards and Technology, Gaithersburg, MD, 1989.

Wallace 1989b Dolores R. Wallace and Roger U. Fujii, "Software Verification and Validation: An Overview", IEEE Software, 6, 3 (May 1989), 10-17.

Weide 1994a B. W. Weide, William F. Ogden, and Murali Sitaraman, "Recasting Algorithms to Encourage Reuse," IEEE Software, 11, 5 (September 1994), 80-88.

Weide 1994b B. W. Weide and Joseph E. Hollingsworth, On Local Certifiability of Software Components, Tech Report OSU-CISRC-1/94-TR04, Department of Computer and Information Science, The Ohio State University, 1994.

Weyuker 1998 E. J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," IEEE Software, 15, 5 (September/October 1998), 54-59.

Wing 1990 J. M. Wing, "A Specifier's Introduction to Formal Methods", Computer, 29, 9 (September 1990), 8-24.

APPENDIX A. ASSURANCE SCORES FOR THE COMPONENTS OF THE WAITING QUEUE SIMULATION PRODUCT LINE

Description of the Application Systems within the Waiting Queue Simulation Product Line

1. **CPU Dispatching**. We want to simulate the behavior of a CPU dispatching mechanism. We are interested in measuring fairness and throughput. There is a single priority queue, with maximum service time (quantum service, Q); once a process has exhausted its service time, it is queued back, with an increased priority.

2. **Self-Serve Carwash**. We have a set of self-serve interchangeable carwash stations. Arriving cars line up at the shortest queue (queues of equal length are interchangeable) and do not change queues subsequently; queues are FIFO, of course. Service is limited to a maximum value (but may take less time), and cars are expected to clear the station once the maximum time has expired. Arrival distribution is Markovian. We are interested in monitoring maximum waiting time (we don't want anybody to leave before being served) and throughput (we want to serve as many people as possible).

3. **Check-Out Counters**. We have a number of check-out counters at a supermarket, some of which are reserved for shoppers with 10 items or less. Shoppers with 10 items or less line up in the shortest express check-out queue; others line in the shortest queue reserved for them. Once they are lined up in some queue, shoppers do not leave the queue until it is their turn. Service time is determined by the shopper (size of his/her cart) and by the productivity of the cash register attendant. Arrival rate is Markovian distribution. The number of stations increases whenever the longest queue exceeds a threshold value L and decreases by one whenever the number of stations of each category is greater than one and the length of the queue is zero. Whenever a new cash register is open, shoppers at the end of the queue rush to line up at the station (talk of fairness!) until the length of the queue equals the shortest current queue of the same type (express checkout, regular checkout). We are interested in average waiting time and fairness.

4. **Immigration Posts**. We have a number of immigration stations at an airport, some of which are reserved for nationals, the others are for foreign citizens. There are two queues: one for nationals, the other for foreigners; each queue feeds into the corresponding set of stations, and there is no transfer between queues. The number of stations that handle nationals increases by one whenever the length of the nationals' queue exceeds some value; and the number of stations that handle foreigners increases by one whenever the length of the foreigners' queue exceeds some other (larger?) value. The arrival rate is a clustered distribution, as passengers come by planeloads. Service time for nationals is constant, and service time for foreigners is determined by the passenger and by the productivity of the immigration agent attendant. We are interested in monitoring throughput.

5. **Check-In Counters**. We have two FIFO queues for passengers at an airline check-in station: a queue for first class and a queue for coach. We have two categories of service stations: first class and coach; the number of stations does not change for the length of the experiment. The duration of the service is the same for all passengers and all stations of the same class, but differs from first class to coach. The arrival rate is Markovian distribution; passengers lines up at their designated queue and do not leave it until they are served. Whenever one queue is empty, the corresponding service stations may serve passengers of the other queue (typically: first class stations serve coach passengers when no first class passengers are awaiting). We are

interested in monitoring the average waiting time and the maximum waiting time for each class of passengers.

6. **Round Robin Dispatching**. Same as system 1, but with a FIFO queue; processes that exceed their time quantum are inserted at the back of the queue.

7. **Self-Serve Carwash, arbitrary service time**. Same as system 2, but without limit on the service time.

8. **Fair Check-Out Counters**. Same as system 3, but whenever a new counter is open, it is filled by shoppers at the front of the longest queues (although in practice they are least motivated to go through the trouble, their queue swapping will probably minimize average waiting time and maximize fairness).

9. **Multiqueue Immigration Posts**. Same as system 4, but with one queue for each post; use the policy of system 3 for queue swapping when a new post is created. Assume also that passengers go from queue to queue whenever their position in the current queue is farther (from the head) than the length of another queue.

10. **Fair Multiqueue Immigration Posts**. Same as system 9 with the policy of system 8 for queue swapping when a new post is created.

Assurance Scoring for the Waiting Queue Simulation Product Line, Part 1

Assumptions:
No component will be used in any systems other than those defined in this product line.
All components will be used for the same (normal) period of time.

Customer Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, max time	4	0	0	1.33	2	1.63	3.26
Check-out Counters	4	0	0	1.33	2	1.63	3.26
Immigration Posts	4	0	0	1.33	2	1.63	3.26
Check-in Counters	4	0	0	1.33	2	1.63	3.26
Round Robin Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, arbitrary time	4	0	0	1.33	2	1.63	3.26
Fair Check-out Counters	4	0	0	1.33	2	1.63	3.26
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.33	2.00		
Ci CRITICALITY SCORE							32.59
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							32.59

Customer Queue Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	2	0	0	0.67	2	0.81	1.63
Check-out Counters	4	0	0	1.33	2	1.63	3.26
Immigration Posts	2	0	0	0.67	2	0.81	1.63
Check-in Counters	2	0	0	0.67	2	0.81	1.63
Round Robin Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, arbitrary time	2	0	0	0.67	2	0.81	1.63
Fair Check-out Counters	2	0	0	0.67	2	0.81	1.63
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	2	0	0	0.67	2	0.81	1.63
AVERAGE				0.80	2.00		
Ci CRITICALITY SCORE							19.56
Ci MARKETABILITY SCORE							1.00
Ci ASSURANCE SCORE							19.56

Event Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.11

				CiSj SEVERITY SCORE	CiSj LONGEVITY SCORE	CiSj RISK SCORE	CiSj CRITICALITY SCORE
Application System	Mission	Safety	Security				
CPU Dispatching	4	0	0	1.33	2	1.48	2.96
Self-Serve Carwash, max time	4	0	0	1.33	2	1.48	2.96
Check-out Counters	4	0	0	1.33	2	1.48	2.96
Immigration Posts	4	0	0	1.33	2	1.48	2.96
Check-in Counters	4	0	0	1.33	2	1.48	2.96
Round Robin Dispatching	4	0	0	1.33	2	1.48	2.96
Self-Serve Carwash, arbitrary time	4	0	0	1.33	2	1.48	2.96
Fair Check-out Counters	4	0	0	1.33	2	1.48	2.96
Multiqueue Immigration Posts	4	0	0	1.33	2	1.48	2.96
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.48	2.96
AVERAGE				1.33	2.00		
Ci CRITICALITY SCORE							29.63
Ci MARKETABILITY SCORE							1.00
Ci ASSURANCE SCORE							29.63

Event List Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, max time	4	0	0	1.33	2	1.63	3.26
Check-out Counters	4	0	0	1.33	2	1.63	3.26
Immigration Posts	4	0	0	1.33	2	1.63	3.26
Check-in Counters	4	0	0	1.33	2	1.63	3.26
Round Robin Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, arbitrary time	4	0	0	1.33	2	1.63	3.26
Fair Check-out Counters	4	0	0	1.33	2	1.63	3.26
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.33	2.00		
Ci CRITICALITY SCORE							32.59
Ci MARKETABILITY SCORE							1.00
Ci ASSURANCE SCORE							32.59

Measurement Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.11

				CiSj SEVERITY SCORE	CiSj LONGEVITY SCORE	CiSj RISK SCORE	CiSj CRITICALITY SCORE
Application System	Mission	Safety	Security				
CPU Dispatching	4	0	0	1.33	2	1.48	2.96
Self-Serve Carwash, max time	4	0	0	1.33	2	1.48	2.96
Check-out Counters	4	0	0	1.33	2	1.48	2.96
Immigration Posts	4	0	0	1.33	2	1.48	2.96
Check-in Counters	4	0	0	1.33	2	1.48	2.96
Round Robin Dispatching	4	0	0	1.33	2	1.48	2.96
Self-Serve Carwash, arbitrary time	4	0	0	1.33	2	1.48	2.96
Fair Check-out Counters	4	0	0	1.33	2	1.48	2.96
Multiqueue Immigration Posts	4	0	0	1.33	2	1.48	2.96
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.48	2.96
AVERAGE				1.33	2.00		
Ci CRITICALITY SCORE							29.63
Ci MARKETABILITY SCORE							1.00
Ci ASSURANCE SCORE							29.63

Priority Queue Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.33

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	3	0	0	1.00	2	2.67	5.33
AVERAGE				1.00	2.00		
Ci CRITICALITY SCORE							5.33
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							5.33

Queue Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
Self-Serve Carwash, max time	3	0	0	1.00	2	1.22	2.44
Check-out Counters	4	0	0	1.33	2	1.63	3.26
Immigration Posts	3	0	0	1.00	2	1.22	2.44
Check-in Counters	3	0	0	1.00	2	1.22	2.44
Round Robin Dispatching	3	0	0	1.00	2	1.22	2.44
Self-Serve Carwash, arbitrary time	3	0	0	1.00	2	1.22	2.44
Fair Check-out Counters	4	0	0	1.33	2	1.63	3.26
Multiqueue Immigration Posts	3	0	0	1.00	2	1.22	2.44
Fair Multiqueue Immigration Posts	3	0	0	1.00	2	1.22	2.44
AVERAGE				1.07	2.00		
Ci CRITICALITY SCORE							23.63
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							23.63

Queue Category Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.33

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.89	1.78
Self-Serve Carwash, max time	2	0	0	0.67	2	0.89	1.78
Check-out Counters	3	0	0	1.00	2	1.33	2.67
Immigration Posts	3	0	0	1.00	2	1.33	2.67
Check-in Counters	4	0	0	1.33	2	1.78	3.56
Round Robin Dispatching	2	0	0	0.67	2	0.89	1.78
Self-Serve Carwash, arbitrary time	2	0	0	0.67	2	0.89	1.78
Fair Check-out Counters	3	0	0	1.00	2	1.33	2.67
Multiqueue Immigration Posts	4	0	0	1.33	2	1.78	3.56
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.78	3.56
AVERAGE				0.97	2.00		
Ci CRITICALITY SCORE							25.78
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							25.78

Queue Facility Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	3	0	0	1.00	2	1.22	2.44
Check-out Counters	3	0	0	1.00	2	1.22	2.44
Immigration Posts	4	0	0	1.33	2	1.63	3.26
Check-in Counters	3	0	0	1.00	2	1.22	2.44
Round Robin Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, arbitrary time	3	0	0	1.00	2	1.22	2.44
Fair Check-out Counters	3	0	0	1.00	2	1.22	2.44
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.03	2.00		
Ci CRITICALITY SCORE							25.26
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							25.26

Server Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj SEVERITY	CiSj LONGEVITY	CiSj RISK	CiSj CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	2	0	0	0.67	2	0.81	1.63
Check-out Counters	3	0	0	1.00	2	1.22	2.44
Immigration Posts	3	0	0	1.00	2	1.22	2.44
Check-in Counters	2	0	0	0.67	2	0.81	1.63
Round Robin Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, arbitrary time	2	0	0	0.67	2	0.81	1.63
Fair Check-out Counters	3	0	0	1.00	2	1.22	2.44
Multiqueue Immigration Posts	3	0	0	1.00	2	1.22	2.44
Fair Multiqueue Immigration Posts	3	0	0	1.00	2	1.22	2.44
AVERAGE				0.90	2.00		
Ci CRITICALITY SCORE							22.00
Ci MARKETABILITY SCORE							1.00
Ci ASSURANCE SCORE							22.00

Server Category Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	3	0	0	1.00	2	1.22	2.44
Check-out Counters	4	0	0	1.33	2	1.63	3.26
Immigration Posts	3	0	0	1.00	2	1.22	2.44
Check-in Counters	3	0	0	1.00	2	1.22	2.44
Round Robin Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, arbitrary time	3	0	0	1.00	2	1.22	2.44
Fair Check-out Counters	4	0	0	1.33	2	1.63	3.26
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.13	2.00		
Ci CRITICALITY SCORE							27.70
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							27.70

Server Facility Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	2	0	0	0.67	2	0.81	1.63
Check-out Counters	4	0	0	1.33	2	1.63	3.26
Immigration Posts	3	0	0	1.00	2	1.22	2.44
Check-in Counters	3	0	0	1.00	2	1.22	2.44
Round Robin Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, arbitrary time	3	0	0	1.00	2	1.22	2.44
Fair Check-out Counters	4	0	0	1.33	2	1.63	3.26
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.03	2.00		
Ci CRITICALITY SCORE							25.26
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							25.26

Assurance Scoring for the Waiting Queue Simulation Product Line, Part 2

Assumptions
Developer believes the Server, Customer Queue, Priority Queue, Measurement, Event and Event List components related to simulating CPU Dispatching can be marketed to five customers
The two Check-Out Counter simulations will be in use for a relatively long period of time
The Customer component will be replaced in all systems in the near future

Customer Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	4	0	0	1.33	1	1.63	1.63
Self-Serve Carwash, max time	4	0	0	1.33	1	1.63	1.63
Check-out Counters	4	0	0	1.33	1	1.63	1.63
Immigration Posts	4	0	0	1.33	1	1.63	1.63
Check-in Counters	4	0	0	1.33	1	1.63	1.63
Round Robin Dispatching	4	0	0	1.33	1	1.63	1.63
Self-Serve Carwash, arbitrary time	4	0	0	1.33	1	1.63	1.63
Fair Check-out Counters	4	0	0	1.33	1	1.63	1.63
Multiqueue Immigration Posts	4	0	0	1.33	1	1.63	1.63
Fair Multiqueue Immigration Posts	4	0	0	1.33	1	1.63	1.63
AVERAGE				1.33	1.00		
Ci CRITICALITY SCORE							16.30
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							16.30

Customer Queue Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	2	0	0	0.67	2	0.81	1.63
Check-out Counters	4	0	0	1.33	3	1.63	4.89
Immigration Posts	2	0	0	0.67	2	0.81	1.63
Check-in Counters	2	0	0	0.67	2	0.81	1.63
Round Robin Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, arbitrary time	2	0	0	0.67	2	0.81	1.63
Fair Check-out Counters	2	0	0	0.67	3	0.81	2.44
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	2	0	0	0.67	2	0.81	1.63
AVERAGE				0.80	2.20		
Ci CRITICALITY SCORE							22.00
Ci MARKETABILITY SCORE							1.14
Ci ASSURANCE SCORE							25.00

Event Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.11

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	4	0	0	1.33	2	1.48	2.96
Self-Serve Carwash, max time	4	0	0	1.33	2	1.48	2.96
Check-out Counters	4	0	0	1.33	3	1.48	4.44
Immigration Posts	4	0	0	1.33	2	1.48	2.96
Check-in Counters	4	0	0	1.33	2	1.48	2.96
Round Robin Dispatching	4	0	0	1.33	2	1.48	2.96
Self-Serve Carwash, arbitrary time	4	0	0	1.33	2	1.48	2.96
Fair Check-out Counters	4	0	0	1.33	3	1.48	4.44
Multiqueue Immigration Posts	4	0	0	1.33	2	1.48	2.96
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.48	2.96
AVERAGE				1.33	2.20		
Ci CRITICALITY SCORE							32.59
Ci MARKETABILITY SCORE							1.36
Ci ASSURANCE SCORE							44.44

Event List Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, max time	4	0	0	1.33	2	1.63	3.26
Check-out Counters	4	0	0	1.33	3	1.63	4.89
Immigration Posts	4	0	0	1.33	2	1.63	3.26
Check-in Counters	4	0	0	1.33	2	1.63	3.26
Round Robin Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, arbitrary time	4	0	0	1.33	2	1.63	3.26
Fair Check-out Counters	4	0	0	1.33	3	1.63	4.89
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.33	2.20		
Ci CRITICALITY SCORE							35.85
Ci MARKETABILITY SCORE							1.36
Ci ASSURANCE SCORE							48.89

Measurement Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.11

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	4	0	0	1.33	2	1.48	2.96
Self-Serve Carwash, max time	4	0	0	1.33	2	1.48	2.96
Check-out Counters	4	0	0	1.33	3	1.48	4.44
Immigration Posts	4	0	0	1.33	2	1.48	2.96
Check-in Counters	4	0	0	1.33	2	1.48	2.96
Round Robin Dispatching	4	0	0	1.33	2	1.48	2.96
Self-Serve Carwash, arbitrary time	4	0	0	1.33	2	1.48	2.96
Fair Check-out Counters	4	0	0	1.33	3	1.48	4.44
Multiqueue Immigration Posts	4	0	0	1.33	2	1.48	2.96
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.48	2.96
AVERAGE				1.33	2.20		
Ci CRITICALITY SCORE							32.59
Ci MARKETABILITY SCORE							1.36
Ci ASSURANCE SCORE							44.44

Priority Queue Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.33

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	3	0	0	1.00	2	2.67	5.33
AVERAGE				1.00	2.00		
Ci CRITICALITY SCORE							5.33
Ci MARKETABILITY SCORE							6
Ci ASSURANCE SCORE							32.00

Queue Component, Part 1

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	1		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
Self-Serve Carwash, max time	3	0	0	1.00	2	1.22	2.44
Check-out Counters	4	0	0	1.33	3	1.63	4.89
Immigration Posts	3	0	0	1.00	2	1.22	2.44
Check-in Counters	3	0	0	1.00	2	1.22	2.44
Round Robin Dispatching	3	0	0	1.00	2	1.22	2.44
Self-Serve Carwash, arbitrary time	3	0	0	1.00	2	1.22	2.44
Fair Check-out Counters	4	0	0	1.33	3	1.63	4.89
Multiqueue Immigration Posts	3	0	0	1.00	2	1.22	2.44
Fair Multiqueue Immigration Posts	3	0	0	1.00	2	1.22	2.44
AVERAGE				1.07	2.22		
Ci CRITICALITY SCORE							26.89
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							26.89

Queue Category Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	2
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.33

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.89	1.78
Self-Serve Carwash, max time	2	0	0	0.67	2	0.89	1.78
Check-out Counters	3	0	0	1.00	3	1.33	4.00
Immigration Posts	3	0	0	1.00	2	1.33	2.67
Check-in Counters	4	0	0	1.33	2	1.78	3.56
Round Robin Dispatching	2	0	0	0.67	2	0.89	1.78
Self-Serve Carwash, arbitrary time	2	0	0	0.67	2	0.89	1.78
Fair Check-out Counters	3	0	0	1.00	3	1.33	4.00
Multiqueue Immigration Posts	4	0	0	1.33	2	1.78	3.56
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.78	3.56
AVERAGE				0.97	2.20		
Ci CRITICALITY SCORE							28.44
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							28.44

Queue Facility Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	3	0	0	1.00	2	1.22	2.44
Check-out Counters	3	0	0	1.00	3	1.22	3.67
Immigration Posts	4	0	0	1.33	2	1.63	3.26
Check-in Counters	3	0	0	1.00	2	1.22	2.44
Round Robin Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, arbitrary time	3	0	0	1.00	2	1.22	2.44
Fair Check-out Counters	3	0	0	1.00	3	1.22	3.67
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.03	2.20		
Ci CRITICALITY SCORE							27.70
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							27.70

Server Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj SEVERITY SCORE	CiSj LONGEVITY SCORE	CiSj RISK SCORE	CiSj CRITICALITY SCORE
Application System	Mission	Safety	Security				
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	2	0	0	0.67	2	0.81	1.63
Check-out Counters	3	0	0	1.00	3	1.22	3.67
Immigration Posts	3	0	0	1.00	2	1.22	2.44
Check-in Counters	2	0	0	0.67	2	0.81	1.63
Round Robin Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, arbitrary time	2	0	0	0.67	2	0.81	1.63
Fair Check-out Counters	3	0	0	1.00	3	1.22	3.67
Multiqueue Immigration Posts	3	0	0	1.00	2	1.22	2.44
Fair Multiqueue Immigration Posts	3	0	0	1.00	2	1.22	2.44
AVERAGE				0.90	2.20		
Ci CRITICALITY SCORE							24.44
Ci MARKETABILITY SCORE							1.01
Ci ASSURANCE SCORE							24.69

Server Category Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

				CiSj	CiSj	CiSj	CiSj
				SEVERITY	LONGEVITY	RISK	CRITICALITY
Application System	Mission	Safety	Security	SCORE	SCORE	SCORE	SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	3	0	0	1.00	2	1.22	2.44
Check-out Counters	4	0	0	1.33	3	1.63	4.89
Immigration Posts	3	0	0	1.00	2	1.22	2.44
Check-in Counters	3	0	0	1.00	2	1.22	2.44
Round Robin Dispatching	4	0	0	1.33	2	1.63	3.26
Self-Serve Carwash, arbitrary time	3	0	0	1.00	2	1.22	2.44
Fair Check-out Counters	4	0	0	1.33	3	1.63	4.89
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.13	2.20		
Ci CRITICALITY SCORE							30.96
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							30.96

Server Facility Component, Part 2

DRIVER	SCORE		DRIVER	SCORE		DRIVER	SCORE
Product			Process			Personnel	
Complexity	2		Maturity	2		Experience	1
Precedent	1		Complexity	1		Stability	1
Testability	1						
Platform	1						
Stability	1		COMPONENT LIKELIHOOD SCORE				1.22

Application System	Mission	Safety	Security	CiSj SEVERITY SCORE	CiSj LONGEVITY SCORE	CiSj RISK SCORE	CiSj CRITICALITY SCORE
CPU Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, max time	2	0	0	0.67	2	0.81	1.63
Check-out Counters	4	0	0	1.33	3	1.63	4.89
Immigration Posts	3	0	0	1.00	2	1.22	2.44
Check-in Counters	3	0	0	1.00	2	1.22	2.44
Round Robin Dispatching	2	0	0	0.67	2	0.81	1.63
Self-Serve Carwash, arbitrary time	3	0	0	1.00	2	1.22	2.44
Fair Check-out Counters	4	0	0	1.33	3	1.63	4.89
Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
Fair Multiqueue Immigration Posts	4	0	0	1.33	2	1.63	3.26
AVERAGE				1.03	2.20		
Ci CRITICALITY SCORE							28.52
Ci MARKETABILITY SCORE							1
Ci ASSURANCE SCORE							28.52

Appendix B. RESOLVE Specification of the SMA Flight Takeoff Time Prediction Subsystem

```
concept Simulate_Runway_Machine

    context

    global context

        mathematics SMA_Database_Math_Machinery
        facility SMA_Database
        facility SMA_DB_Departure_Get_Flights_Capability

    local context

        math operation Is_Substring (
            s1: string of character,
            s2: string of character
            ) : boolean
        explicit definition
            there exists s3, s4: string of character (s2 = s3 * s1 * s4)

        math operation Actual_Takeoff_Times_Used (
            q: string of FLIGHT_TIME_PAIR,
            db: SMA_Database_Machine
            ) : boolean
        explicit definition
            for all fid: FLIGHT_ID (
                if there exists q1, q2: string of FLIGHT_TIME_PAIR, t: DISCRETE_TIME (
                    q = q1 * <(fid,t)> * q2 and db.actual_takeoff_time(fid) > -1
                )
                then t = db.actual_takeoff_time(fid)
                end if
            )

        math operation Safety_Delays_Met (
            q: string of FLIGHT_TIME_PAIR,
            rid: RUNWAY_ID,
            db: SMA_Database_Machine
            ) : boolean
        explicit definition
            for all f1, f2: FLIGHT_ID (
                if there exists q1, q2: string of FLIGHT_TIME_PAIR,
                t1, t2: DISCRETE_TIME (
                    q = q1 * <f1,t1> * <f2,t2> * q2
                    )
                then t2 >= t1 + db.delay_time(db.aircraft_type(f1)) +
                        db.roll_time(db.aircraft_type(f2))
                end if
            )
```

```
math operation Taxi_Times_Met (
        q: string of FLIGHT_TIME_PAIR,
        rid: RUNWAY_ID,
        db: SMA_Database_Machine
        ) : boolean
explicit definition
        for all fid: FLIGHT_ID (
                if there exists q1,q2: string of FLIGHT_TIME_PAIR, t: DISCRETE_TIME (
                        q = q1 * <fid,t> * q2
                )
                then t >= Pushback_Time(fid, db) + db.taxi_time(db.gate(fid), rid) +
      db.roll_time(db.aircraft_type(fid))
                end if
        )

math operation Well_Formed_Runway_Queue (
        q: string of FLIGHT_TIME_PAIR,
        m: Simulate_Runway_Machine_State,
        db: SMA_Database_Machine
        ) : boolean
explicit definition
        for all fid: FLIGHT_ID (
                if db.runway_used(fid) = m.rid or
                        (db.runway_used(fid) = empty_string and
                        db.assigned_runway(m.sid, db.departure_gate_area(fid)) = m.rid)
                then In_Line(q,fid, db)
                        and Actual_Takeoff_Times_Used(q, m, db) and
                        Safety_Delays_Met(q, m.rid, db) and Taxi_Times_Met(q, m.rid, db)
                end if
        )

math operation Proper_Flights_In_Queue (
        m: Simulate_Runway_Machine_State,
        db: SMA_Database_Machine
        ) : boolean
explicit definition
        if there exists c: CONFIGURATION, q1: string of FLIGHT_TIME_PAIR (
                db.c.id = m.sid and Well_Formed_Runway_Queue(q1, m, db)
                )
        then Is_Substring (m.q, q1) and
                for all ft: FLIGHT_TIME_PAIR (ft is in elements (m.q))
                         m.tbegin <= ft.t <= m.tend

math operation Cumulative_Wait_Time (
        q: string of FLIGHT_TIME_PAIR,
        r: RUNWAY_ID,
        db: SMA_Database_Machine) : DISCRETE_TIME
explicit definition
        if there exists ft: FLIHT_TIME_PAIR, q1: string of FLIGHT_TIME_PAIR (
                q = <ft> * q1
                 )
        then Cumulative_Wait_Time=ft.t -Pushback_Time(ft.f, db) -db.taxi_time(db.gate(ft.f), r)
                         - db.roll_time(ft.f) + Cumulative_Wait_Time(q1, r)
        else
                        Cumulative_Wait_Time = 0
        end if
```

```
interface

        type Simulate_Runway_Machine_State is modeled by (
                sid: string of character
                rid: string of character
                tbegin:  DISCRETE_TIME
                tend:  DISCRETE_TIME
                q: string of FLIGHT_TIME_PAIR
                cum_wait: integer
                ready_to_extract: boolean
                )
        exemplar m
        constraint
                Is_Allowed_Split_Name (sid) and
                Is_Allowed_Runway_Name (rid) and
                tend >= tbegin and
                cum_wait >= 0 and
        initialization
                ensures   m.ready_to_extract = false

        operation Prepare_Simulation (
                alters     m: Simulate_Runway_Machine_State
                )
        ensures  m.ready_to_extract = false and
                                m.sid=#m.sid and m.rid=#m.rid and
                                m.tbegin=#m.tbegin and m.tend=#m.tend and
                                m.q=#m.q and m.cum_wait=#m.cum_wait

        operation Set_Split (
                alters     m: Simulate_Runway_Machine_State,
                consumes  split: Char_String
                )
        requiresIs_Allowed_Split_Name (split) and m.ready_to_extract = false
        ensures  m.sid = split and m.rid=#m.r and
                                m.tbegin=#m.tbegin and m.tend=#m.tend and
                                m.q=#m.q and m.cum_wait=#m.cum_wait and
                                m.ready_to_extract=false

        operation Set_Runway (
                alters     m: Simulate_Runway_Machine_State,
                consumes  runway: Char_String
                )
        requiresIs_Allowed_Runway_Name (runway) and
                                        m.ready_to_extract=false
        ensures  m.rid=runway and m.sid=#m.s and
                                m.tbegin=#m.tbegin and m.tend=#m.tend and
                                m.q=#m.q and m.cum_wait=#m.cum_wait and
                                m.ready_to_extract=false
```

```
operation Set_Times (
        alters     m: Simulate_Runway_Machine_State,
        consumes  begin_time: Integer,
        consumes  end_time: Integer
        )
requires begin_time >= 0 and end_time >= begin_time and
                m.ready_to_extract=false
ensures  m.tbegin=begin_time and m.tend=end_time and
                m.sid=#m.sid and m.rid=#m.rid and m.q=#m.q and
                m.cum_wait=#m.cum_wait and
                m.ready_to_extract=false

operation Simulate_Runway (
        alters     m: Simulate_Runway_Machine_State,
        preserves db: SMA_Database_Machine
        )
requires m.ready_to_extract = false
ensures  Proper_Flights_In_Queue(m, db) and
                m.sid = #m.sid and m.rid=#m.rid and
                m.tbegin=#m.tbegin and m.tend=#m.tend and
                m.ready_to_extract=true

operation Extract_Next (
        alters     m: Simulate_Runway_Machine_State,
        produces  flight_number: Char_String,
        produces  takeoff_time: Integer
        )
requires |m.q| /= 0 and m.ready_to_extract = true
ensures  #m.q = <flight_number, takeoff_time> * m.q and
                m.sid=#m.sid and m.rid=#m.rid and
                m.tbegin=#m.tbegin and m.tend=#m.tend and
                m.cum_wait=#m.cum_wait and
                m.ready_to_extract=#m.ready_to_extract

operation Cumulative_Wait_Time (
        preserves m: Simulate_Runway_Machine_State,
        preserves db: SMA_Database_Machine,
        produces  wait_time: Integer
        )
ensures   wait_time = Cumulative_Wait_Time (m.q, m.rid, db)

operation Queue_Length_Of (
        preserves m: Simulate_Runway_Machine_State,
        produces  length: Integer
        )
requires m.ready_to_extract = true
ensures  length = |m.q|

operation Runway_Of (
        preserves m: Simulate_Runway_Machine_State,
        produces  runway: Char_String
        )
requires m.ready_to_extract = true
ensures  runway = m.rid
```

```
operation Split_Of (
            preserves m: Simulate_Runway_Machine_State,
            produces split: Char_String
            )
      requires m.ready_to_extract = true
      ensures split = m.sid

      operation Begin_Time_Of (
            preserves m: Simulate_Runway_Machine_State,
            produces begin_time: Integer
            )
      requires m.ready_to_extract = true
      ensures begin_time = m.tbegin

      operation End_Time_Of (
            preserves m: Simulate_Runway_Machine_State,
            produces end_time: Integer
            )
      requires m.ready_to_extract = true
      ensures end_time = m.tend

end Simulate_Runway_Machine
```

mathematics SMA_Database_Math_Machinery

context

global context

state constants
allowed_flight_IDs **is modeled by set of string of character**
allowed_aircraft_types **is modeled by set of string of character**
allowed_gate_names **is modeled by set of string of character**
allowed_DGA_names **is modeled by set of string of character**
allowed_runway_names **is modeled by set of string of character**
allowed_split_names **is modeled by set of string of character**

interface

math operation Is_Allowed_Flight_ID (
s: **string of character**
): **boolean**
explicit definition
s **is in** allowed_flight_IDs

math operation Is_Allowed_Aircraft_Type (
s: **string of character**
): **boolean**
explicit definition
s **is in** allowed_aircraft_types

math operation Is_Allowed_Gate_Name (
s: **string of character**
): **boolean**
explicit definition
s **is in** allowed_gate_names

math operation Is_Allowed_DGA_Name (
s: **string of character**
): **boolean**
explicit definition
s **is in** allowed_DGA_names

math operation Is_Allowed_Runway_Name (
s: **string of character**
): **boolean**
explicit definition
s **is in** allowed_runway_names

math operation Is_Allowed_Split_Name (
s: **string of character**
): **boolean**
explicit definition
s **is in** allowed_split_names

math subtype DISCRETE_TIME **is integer**
exemplar t
constraint
t >= 0

math subtype OPTIONAL_DISCRETE_TIME **is integer**
exemplar t
constraint
t >= -1

math subtype FLIGHT_ID **is string of character**
exemplar fid
constraint
Is_Allowed_Flight_ID(fid)

math subtype AIRCRAFT_TYPE_NAME **is string of character**
exemplar a
constraint
Is_Allowed_Aircraft_Type (a)

math subtype GATE_NAME **is string of character**
exemplar g
constraint
Is_Allowed_Gate_Name (g)

math subtype OPTIONAL_GATE_NAME **is string of character**
exemplar g
constraint
Is_Allowed_Gate_Name (g) **or** g = empty_string

math subtype DGA_NAME **is string of character**
exemplar dga
constraint
Is_Allowed_DGA_Name (dga)

math subtype RUNWAY_ID **is string of character**
exemplar rid
constraint
Is_Allowed_Runway_Name (rid)

math subtype OPTIONAL_RUNWAY_ID **is string of character**
exemplar rid
constraint
Is_Allowed_Runway_Name (rid) **or** g = empty_string

math subtype SPLIT_ID **is string of character**
exemplar sid
constraint
Is_Allowed_Split_Name (sid)

math subtype GATE_RUNWAY_PAIR **is** (
g: GATE,
r: RUNWAY_ID
)

math subtype CONFIGURATION **is** (
sid: SPLIT_ID,
split: **function from** DGA_NAME **to** RUNWAY_ID
)

```
        math subtype FLIGHT_TIME_PAIR is (
                f: FLIGHT_ID ,
                t: DISCRETE_TIME
        )

        math subtype SMA_DEPARTURE_DB_MODEL is (
                aircraft_type: function from FLIGHT_ID to AIRCRAFT_TYPE_NAME
                gate: function from FLIGHT_ID to OPTIONAL_GATE_NAME
                departure_gate_area: function from FLIGHT_ID to DGA_NAME
                runway_used: function from FLIGHT_ID to OPTIONAL_RUNWAY_ID
                predicted_pushback_time: function from FLIGHT_ID to DISCRETE_TIME
                actual_pushback_time: function from FLIGHT_ID to OPTIONAL_DISCRETE_TIME
                actual_takeoff_time: function from FLIGHT_ID to OPTIONAL_DISCRETE_TIME
                delay_time: function from AIRCRAFT_TYPE_NAME to DISCRETE_TIME
                roll_time: function from AIRCRAFT_TYPE_NAME to DISCRETE_TIME
                taxi_time: function from GATE_RUNWAY_PAIR to DISCRETE_TIME
                config: set of CONFIGURATION
        )

        math operation In_Line (
                q: string of FLIGHT_TIME_PAIR,
                fid: FLIGHT_ID
                ) : boolean
        explicit definition
                there exists t:DISCRETE_TIME, q1,q2: string of FLIGHT_TIME_PAIR (
                        q = q1 * <fid,t> * q2
                )

        math operation Pushback_Time (
                fid: FLIGHT_ID
                db: SMA_DEPARTURE_DB_MODEL
                ): DISCRETE_TIME
        explicit definition
                if db.actual_pushback_time(fid) > 0
                then Pushback_Time = db.actual_pushback_time(fid)
                else Pushback_Time = db.predicted_pushback_time (fid)
                end if

end SMA_Database_Math_Machinery
```

```
concept SMA_Database

	context

	global context

		mathematics SMA_Database_Math_Machinery

	interface

		type SMA_Database_Machine is modeled by math supertype of
			SMA_DEPARTURE_DB_MODEL

		operation Aircraft_Type (
		preserves db: SMA_Database_Machine,
			preserves fid: Char_String
			): Char_String
		requiresIs_Allowed_Flight_ID (fid)
		ensures  (fid, Aircraft_Type) is in db.aircraft_type

		operation Gate (
		preserves db: SMA_Database_Machine,
			preserves fid: Char_String
			): Char_String
		requiresIs_Allowed_Flight_ID (fid)
		ensures  (fid, Gate) is in db.gate

		operation Runway_Used (
			preserves db: SMA_Database_Machine,
			preserves fid: Char_String
			): Char_String
		requiresIs_Allowed_Flight_ID (fid)
		ensures  (fid, Runway_Used) is in db.runway_used

		operation Departure_Gate_Area (
			preserves db: SMA_Database_Machine,
			preserves fid: Char_String
			): Char_String
		requiresIs_Allowed_Flight_ID (fid)
		ensures  (fid, Departure_Gate_Area) is in db.departure_gate_area

		operation Predicted_Pusback_Time (
			preserves db: SMA_Database_Machine,
			preserves fid: Char_String
			): Integer
		requiresIs_Allowed_Flight_ID (fid)
		ensures  (fid, Predicted_Pushback_Time) is in db.predicted_pushback_time

		operation Actual_Pushback_Time (
			preserves db: SMA_Database_Machine,
			preserves fid: Char_String
			): Integer
			requiresIs_Allowed_Flight_ID (fid)
			ensures  (fid, Actual_Pushback_Time) is in db.actual_pushback_time
```

```
operation Actual_Takeoff_Time (
        preserves db: SMA_Database_Machine,
        preserves fid: Char_String
        ): Integer
        requiresIs_Allowed_Flight_ID (fid)
        ensures (fid, Actual_Takeoff_Time) is in db.actual_takeoff_time

operation Delay_Time (
        preserves db: SMA_Database_Machine,
        preserves aircraft_type: Char_String
        ): Integer
        requiresIs_Allowed_Aircraft_Type (aircraft_type)
        ensures (aircraft_type, Delay_Time) is in db.delay_time

operation Roll_Time (
        preserves db: SMA_Database_Machine,
        preserves aircraft_type: Char_String
        ): Integer
        requiresIs_Allowed_Aircraft_Type (aircraft_type)
        ensures (aircraft_type, Roll_Time) is in db.roll_time

operation Taxi_Time (
        preserves db: SMA_Database_Machine,
        preserves gid: Char_String,
        preserves rid: Char_String
        ): Integer
requiresIs_Allowed_Flight_ID (fid) and Is_Allowed_Gate_Name (gid)
ensures ((gid, rid), Taxi_Time) is in db.taxi_time

operation Assigned_Runway (
        preserves db: SMA_Database_Machine,
        preserves sid: Char_String,
        preserves dga: Char_String
        ): Char_String
requiresIs_Allowed_Split_Name (sid) and Is_Allowed_DGA_Name (dga)
ensures (sid, (dga, Assigned_Runway)) is in db.configuration
```

```
end SMA_Database
```

```
concept SMA_DB_Departure_Get_Flights_Capability

        context

        global context

                mathematics SMA_Database_Math_Machinery

        interface

                type Get_Flights_Machine_State is modeled by (
                                q: string of string of character
                        )
                exemplar gf
                initialization
                        ensures  |gf.q| = 0


                operation Select_Flights_To_Runway (
                        preserves db: SMA_Database_Machine
                        alters    gf: Get_Flights_Machine_State,
                        preserves sid: Char_String,
                        preserves rid: Char_String
                        )
                requiresIs_Allowed_Split_Name (sid) and Is_Allowed_Runway_Name (rid)
                ensures for all fid: string of character where (Is_Allowed_Flight_ID(fid)) (
                                fid is in elements (gf.q) iff db.runway_used(fid) = rid
                                or (db.runway_used(fid) = empty_string
                                and db.assigned_runway(sid,db.departure_gate_area(fid)) = rid)
                                )
                        and (if there exists s1, s2: string of string of character, f1, f2: string of character (
                                gf.q = s1 * <f1> * <f2> * s2
                                )
                                then
                                        (db.actual_takeoff_time(f1) <> -1 and
                                         db.actual_takeoff_time(f2) <> -1 and
                                         db.actual_takeoff_time(f1) < db.actual_takeoff_time(f2)) or
                                        (db.actual_takeoff_time(f1) > -1 and
                                         db.actual_takeoff_time(f2) = -1) or
                                        (db.actual_takeoff_time(f1) = -1 and
                                         db.actual_takeoff_time(f2) = -1) and
                                         db.actual_pushback_time(f1) <> -1 and
                                         db.actual_pushback(f2) <> -1 and
                                         db.actual_pushback_time(f1) < db.actual_pushback_time(f2)) or
                                        (db.actual_takeoff_time(f1) = -1 and
                                         db.actual_takeoff_time(f2) = -1) and
                                         db.actual_pushback_time(f1) > -1 and
                                         db.actual_pushback_time(f2) = -1) or
                                        (db.actual_takeoff_time(f1) = -1 and
                                         db.actual_takeoff_time(f2) = -1) and
                                         db.actual_pushback_time(f1) = -1 and
                                         db.actual_pushback_time(f2) = -1 and
                                         db.predicted_pushback_time(f1) <= db.predicted_pushback_time(f2))
                                end if)
```

```
        operation Get_Next_Flight_To_Runway (
                alters     gf: Get_Flights_Machine_State,
                produces  fid: Char_String
                )
        requires|gf.q| > 0
        ensures
                        #gf.q = <fid,t> * gf.q

        operation Number_Of_Flights_To_Runway (
                preserves gf: Get_Flights_Machine_State,
                ) : Integer
                ensures   Number_Of_Flights_To_Runway = |gf.q|

end SMA_DB_Departure_Get_Flights_Capability
```