Department of

Computer Science

Domain Based Testing : A Reuse Oriented Test Method

Richard T. Mraz

Technical Report CS-94-104

February 9, 1994

Colorado State University

RESEARCH PROPOSAL

DOMAIN BASED TESTING : A REUSE ORIENTED TEST METHOD

Submitted by

Richard T. Mraz

Department of Computer Science

In partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Semester - Fall 1993

RESEARCH PROPOSAL ABSTRACT

DOMAIN BASED TESTING : A REUSE ORIENTED TEST METHOD

In general, the test data generation problem is equivalent to the Halting Problem; therefore, it is undecidable. This does not have to be the case for specific problem domains. We propose a solution to the test data generation problem for command-based systems, and we call our method Domain Based Testing (DBT). DBT uses Domain Analysis and a Domain Model to automate test generation. Domain Analysis was originally developed to support software reuse. It is one way to extract common information about a problem domain. The result of Domain Analysis is a Domain Model. Domain Models represent the reuse problem domain and they serve as a mechanism to create instances of the reusable components. Instead of using the Domain Model for reuse, we use it as a structure from which test cases can be generated. Part of the Domain Model for DBT includes the syntax and semantics of the command language. Historically, grammars have been successful at test generation for compilers. On the other hand, grammars have not been successful at general purpose test generation because of the combinatorial explosion of semantic rules that must be written and maintained. Domain Based Testing addresses these issues in two ways. First, command language syntax is separate from command language semantics. Second, the test generation process is divided into three phases, (1) Scripting, (2) Command Template Generation, and (3) Parameter Value Selection. DBT is able to handle the complexity of the semantic rules by distributing them across all three phases. Domain Based Testing can be classified as a method that supports a wide variety of Black-Box test strategies. Because it is based on ideas from software reuse, DBT also provides a good structure for *test case reuse*. In fact, test cases can be archived at each phase of test generation. The DBT Domain Model also provides a unique structure from which *regression test suites* can be selected.

Richard T. Mraz
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Semester - Fall 1993

CONTENTS

1 Proposal Overview **1**
1.1 Introduction . . . 1
1.2 Problem Statement . . . 2
1.3 Overview of the Proposal . . . 5

2 Existing Work **6**
2.1 Introduction . . . 6
2.2 Formal Language . . . 6
2.2.1 Background - Language Requirements . . . 7
2.2.2 Background - Using Formal Language for Test Generation . . . 8
2.2.3 Formal Language - Specification and Design Issues . . . 9
2.2.4 How can we use Formal Languages for Domain Based Testing . . . 11
2.3 Software Reuse . . . 12
2.3.1 Background - Domain Analysis . . . 13
2.3.2 How Domain Analysis Can be Used for Domain Based Testing . . . 15
2.4 Requirements Analysis . . . 16
2.4.1 Background - Object Behavior Analysis . . . 17
2.4.2 How OBA Can be Used for Domain Based Testing . . . 19
2.5 Software Testing - Test Case Design . . . 21
2.5.1 Background - Structural Testing . . . 21
2.5.2 Background - Black-Box Testing . . . 22
2.5.3 Background - Regression Testing . . . 23
2.5.4 How Software Testing can be used for Domain Based Testing . . . 24
2.6 Comparisons between Domain Based Testing and Other Methods . . . 26
2.6.1 Domain Testing . . . 26
2.6.2 Category-Partition Testing . . . 27
2.7 Summary . . . 29

3 Domain Based Testing **31**
3.1 Introduction . . . 31
3.2 Command Language Analysis . . . 31
3.3 Object Analysis . . . 33
3.3.1 Objects and Their Elements . . . 33
3.3.2 Relationships Between Objects and Their Elements . . . 39
3.4 Command Definition . . . 44
3.5 Script Definition (Command Sequencing) . . . 47
3.5.1 Script Classes . . . 48
3.5.2 Script Rules . . . 49
3.5.3 Summary . . . 50

4 Test Generation Process Model **51**
4.1 Introduction 51
4.2 System Specification 51
4.3 System Configuration 51
4.4 Test Criteria 53
4.5 Test Generation 54

5 Research Plan **56**
5.1 Introduction 56
5.2 Domain Based Testing : Design Issues 56
5.3 Domain Based Testing : Experimental Evaluations 58
5.4 Domain Based Regression Testing 60
5.5 Research Schedule 62
5.6 Contributions of this Work 62

6 REFERENCES **63**

LIST OF FIGURES

2.1 Research Basis for Domain Based Testing 7
2.2 A Reuse Infrastructure [HC91] 15
2.3 Requirements and Test Suite Hierarchies [vMO93] 24

3.1 Anatomy of an Object 34
3.2 Analyzing Robot Manufacturing Commands for Objects 34
3.3 Objects for the Robot Manufacturing Example 35
3.4 Object Element Classification 36
3.5 Parameter Attributes 36
3.6 Mode Parameter - `SET TOOL FAILURE` Command 37
3.7 Generic Object Model 41
3.8 Robot Manufacturing Example - Object Hierarchy 42
3.9 Object Splitting - What to do when constraints are local to one object 43
3.10 Syntax Diagram for the `POWER` Command 45
3.11 State Transition Diagram for a Generic Script 48

4.1 Test Generation Process Model 52
4.2 Window Based Test Generation Tool 55

5.1 Test Case Reuse Concept 59
5.2 Requirements and Test Suite Hierarchies [vMO93] 60
5.3 DBT Regression and Test Suite Hierarchies 61

LIST OF TABLES

2.1 Attribute Grammar and W-Grammar Examples 10
2.2 Comparison of Object Behavior Analysis and Domain Based Testing 20
2.3 Some Control Flow Coverage Measures 22
2.4 Some Data Flow Coverage Measures 23
2.5 Comparison of Category-Partition Testing and Domain Based Testing 30

3.1 Domain Analysis Steps for Domain Based Testing 32
3.2 Robot Manufacturing Commands 33
3.3 Command Glossary Entry for the `DRILL` command 33
3.4 Object Glossary Entry for the **Robot** Object 35
3.5 Object Glossary Entry for the **Robot** Object 39
3.6 Object Element Glossary 40
3.7 Types of Relationships 41
3.8 Command Glossary Entry for the `DRILL` command 45
3.9 Command Glossary Entry for the `SAW` Command 46
3.10 State Transition Table for the *robot-status* Precondition 46
3.11 Intracommand Rules 47
3.12 Robot.script 48
3.13 Script Classes 49
3.14 Script Rule: Parameter Value Selection 50

4.1 Operations for System Specification 52
4.2 Operations for System Configuration 53
4.3 Operations for Test Criteria 54
4.4 Operations for Test Generation 55

5.1 Design Issues to Investigate 57
5.2 Design Alternatives : Semantic Rule Resolution 58
5.3 Domain Based Coverage Measures 59
5.4 Domain Based Regression Testing Rule Matrix 61
5.5 Domain Based Testing Research Schedule 62

Chapter 1

PROPOSAL OVERVIEW

1.1 Introduction

Software testing consumes "at least half of the labor expended to produce a working program" [Bei90]. Therefore, a fundamental consideration in program testing is one of economics. Testing is expensive in time, machine, and labor costs. Research over the past 30 years has focused on reducing the time to test a software product, reducing the number of test case executions needed to test the software, and increasing the productivity of test engineers [Mye79] [Mye76] [Bei90] [GH88] [ABC82]. Typical goals are to maximize the yield on each test case, and maximize the number of errors found with the fewest number of test cases.

The literature shows that software testing is a difficult problem [Mye79] [Bei90]. The complexity of the problem can be demonstrated by examining White-Box Testing, Black-Box Testing, and Formal Methods. White-Box Testing is a testing strategy based on the internal structure of the program. Test cases are created by looking at the logic of the program's source code. The ultimate test suite will execute every path in the program at least once. Unfortunately, loop control structures make this criterion almost impossible to achieve. Even for small programs with a single loop, the number of paths through the program is too large to test. Therefore, exhaustive testing of execution paths cannot be achieved. Black-Box Testing identifies the conditions where a program does not behave according to its specification. The ultimate Black-Box test suite would use every possible input condition as a test case. The test suite would include not only valid input conditions but all possible input conditions. Potentially, we could have infinite test cases. We could restrict some of the input values because programs run on finite machines. Yet, the sets would be so large that we can consider them as infinite for testing purposes. Again, exhaustive testing of all inputs cannot be achieved. Because exhaustive testing is not feasible, test engineers must choose a subset of all possible test cases. The subset is sometimes called the "reliable test set." If the program is correct with respect to the "reliable test set," then we assume it is correct with respect to the entire input domain. Unfortunately, the problem of choosing the "reliable test set" is not easy, and Howden shows that it is equivalent to solving the Halting Problem [vM93a]. In general, test data generation is undecidable, but it does not have to be for specific problem domains. We propose an automated test generation method for a specific domain, *Command-Based Systems*. Based on research from the software reuse community, we call our method Domain Based Testing (DBT).

For command-based systems, test case generation can be automated using the syntax of the command language. For truly meaningful tests, one must also consider semantic content, too. Research shows that using grammars to generate tests for compilers has been successful [Pay78] [BS82] [CRV+80] [FvBK+91]. From this work, others attempted to automate test generation for general designs and implementations [BF79] [DH81]. Their results were not as promising. In retrospect, we cite two reasons for their lack of success. First, all syntax and semantic information were coded into the grammar. Such grammars are called attribute grammars or W-grammars. They are complex grammars that can represent context-sensitive information, but they do not make automated test generation easy. The number of semantic rules that must be defined and maintained during test generation becomes unmanageable. The second reason test case generation for general design didn't work was that all phases of the test generation were placed into a single mechanism. This makes it difficult to modify the system under test because grammar productions must be re-written. It also limits the applicability of the test generation system. It may be good for test generation, but it can't be used for other software testing issues.

Understanding both the benefits and the shortcomings of using a grammar for test generation, Domain Based Testing relies on three phases (1) Scripting, (2) Command Template Generation, and (3) Parameter Value Selection. Each phase addresses a specific part of the test generation process. Scripting focuses on command sequencing, Command Generation creates command templates, and Parameter Value Selection fills each command with parameter values. A grammar will be used to represent the syntax of the command language, but the semantic information will be encoded as rules that span all three DBT phases. This strategy will help us manage the complexity of test case generation, improve test case reuse, and create a unique structure from which regression test suites can be selected.

1.2 Problem Statement

The objective of our research is to develop a Domain Based Testing system that automatically generates test cases for command-based systems. Domain Based Testing is a reuse oriented testing method based on Domain Analysis and Domain Modeling. Domain Analysis has long been used to develop reusable software components [BP89] [HC91]. This research proposes to extend Domain Analysis to software testing. One objective of software test research has been to increase the productivity of the test engineer. DBT is designed to address this objective by automating test case generation, archiving test cases for reuse, and by using the DBT Domain Model for regression test suite selection.

Several issues must be resolved for this research to be successful. First, Domain Analysis for DBT must be defined. The Domain Analysis examines the system under test and it generates a Domain Model from which tests can be generated. The command language must be examined, static and dynamic behaviors of the software system must be represented, and syntax and semantic rules must be defined. Once the Domain Model is developed, we need procedures for test case generation. We call this the Test Generation Process Model. The process model must provide an easy to use, interactive environment for the tester. Testers must be able to modify the system configuration, semantic rule definition, and parameter values. The process model should provide mechanisms for the tester to modify test criteria.

The third issue to resolve is Domain Based Regression Testing. Software is constantly undergoing modification, update, and change. Regression testing is a term used when a software product is tested after modification. The goal of regression testing is to make sure old features still work and the "fixes" don't cause new problems. Most of the time, it is economically infeasible to re-run all of the test cases from the original system. Therefore, one must choose a subset of test cases that have a high potential to detect errors. The Domain Model for DBT defines a structure from which regression test rules can be defined. Effective regression test suites reduces the number of test cases that need to be executed. Automating some of the test selection also improves tester productivity.

The goals of our research can be summarized as follows:

- Define a General Testing Approach for Testing Command-Based Systems
- Obey both Syntax and Semantic Rules
- Automate Test Case Generation
- Generate Test Cases Efficiently
- Create an Environment for Test Case Reuse
- Create a Structure for Regression Testing Rules

Domain Based Testing should be a general purpose testing method for command-based systems. To meet this goal, we must develop Domain Analysis procedures, a Domain Model representation, and a Test Generation Process. Tests should obey both the syntax of the command language as well as semantic rules for command sequencing and parameter value selection. This goal will increase the "yield" on meaningful test cases, and it may reduce the number of commands each test case executes. Domain Based Testing can be realized through a test generation tool. Testing is a time consuming and sometimes tedious job. Hand-written test cases tend to be error prone and inefficient. Using an automated test generation tool allows testers to focus on high level test scenarios instead of low level details. We envision an interactive tool used by the test engineer to achieve this goal. Testers will work from a high level of abstraction and the test tool will handle command syntax and parameter value selection. While generating test cases, we must be concerned about efficiency. We know from previous research that semantic rules can overwhelm test generation for arbitrary designs. Domain Based Testing must be able to handle semantic rules and generate test cases within a reasonable amount of time. Because the testing tool is interactive, test engineers will expect a short *Script-to-Test Case* cycle time. If the time between scripting and test case generation is too long, testers will not use the tool. Domain Based Testing also provides unique ideas for test case reuse. Because Domain Base Testing is based on ideas from the software reuse community, we should expect promising results from test case reuse. For example, one could save tests at each stage of generation (Script, Command Generation, Parameter Value Selection). Later, archived tests can be recalled for future tests, modified for new configurations, or re-generated using a new release of the command language syntax. The last goal of this research is to define rules for Domain Based Regression Testing. Regression testing is used to make sure modifications to a software system does not corrupt old features and the new features perform as expected. It is not economically feasible to re-run all of the previous test cases. The Domain Model for Domain Based Testing provides a unique structure from which regression test suites can be selected.

1.3 Overview of the Proposal

The remainder of the proposal is divided into five chapters. Chapter 2 presents background material and a literature review. In Chapter 3, we develop the Domain Analysis procedures for Domain Based Testing. A general approach for DBT analysis is provided and it shows what needs to be represented for automated test generation. Chapter 4 discusses the Test Generation Process Model. Here, we talk about some implementation and design issues for Domain Based Testing. In Chapter 5, we present our research plan for assessing Domain Based Testing and the ideas for constructing Domain Based Regression Testing rules.

Chapter 2

EXISTING WORK

2.1 Introduction

This chapter provides background information and a literature review for our research. As shown in Figure 2.1, Domain Based Testing (DBT) exploits ideas from (1) Formal Languages, (2) Software Reuse, (3) Requirements Analysis, and (4) Software Testing. A formal language is used to represent the syntax of the command language for DBT. Results from earlier work show that grammars can be used for automated test generation, but one must be careful about design and implementation issues. Software Reuse is a popular research topic with respect to capturing common information, designs, and programs from a problem domain. Extending these ideas to software testing serves as a basis for Domain Based Testing. Requirements Analysis is needed by DBT because one must analyze the system under test. For Domain Based Testing, we will not need a general purpose analysis method because our problem domain is restricted to the testing phase of the life cycle. The last area discussed in this chapter is Software Testing. Some ideas in software testing can be used directly by DBT while others are used for design and implementation decisions. We also compare Domain Based Testing to other testing methods.

In the following sections, we review all four areas. Our format for each section is to present background information and literature review first. Then, we describe how the information can be useful for Domain Based Testing.

2.2 Formal Language

Over the past 20 years, formal languages have been used for automated test generation. Some methods have been successful when constrained by a specific problem domain. For example, most of the early research investigated automated test generators for compilers [Pay78] [BS82] [CRV$^+$80]. Others tried to use formal languages to generate test plans or to generate test cases for generic designs and implementations [BF79] [DH81]. The efforts to extend formal languages into generic test case generation were not too successful because of the combinatorial explosion of semantic

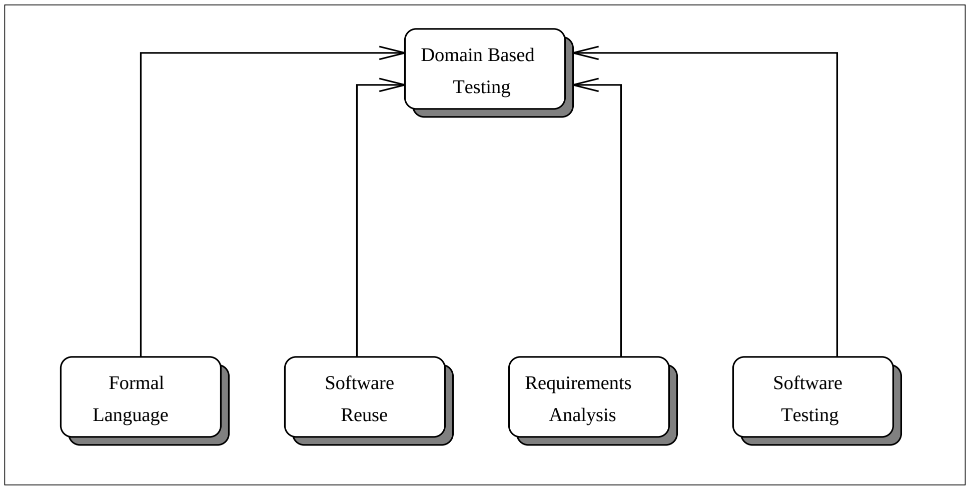


Figure 2.1: Research Basis for Domain Based Testing

rules that must be applied. In the next two sections, background information about using formal languages for test case generation is presented. We also outline the requirements of a command language that must be met for Domain Based Testing. Finally, we show how we can use formal languages in the proposed research.

2.2.1 Background - Language Requirements

In general, not all command languages can be used to automatically generate test cases. While tests can be generated using the syntax of the language, one must also consider semantic content. To be a worthwhile candidate for Domain Based Testing, the command language must pass two tests. First, parameters must have a high semantic content, and second, the parameters must map to physical or logical objects in the system.

The first requirement states that most of the semantic information for the commands must be encoded in the parameters of the language. A counterexample demonstrates the point. Consider a compiler with a command language interface. The following command compiles source program `example.c` using the code optimizer, renaming the object file, and linking the math library.

```
cc -O -o example example.c -lm
```

The compiler has parameters in its command language interface, but little semantic content is found in the parameters. Instead, we must go to the contents of the `example.c` file. Because compiler command languages do not have a high degree of semantic information in their parameters, they do not represent a good choice for automated test generation using a command language.

The second requirement states that the parameters of the command language must be associated with objects of the system. Associating parameters with objects is important for several reasons. First, the effects of the parameter or command can be identified by the objects it influences. For instance, as commands are issued, the values of the parameters can update the state of the system. Second, the objects of the system can be organized into a hierarchy. From the hierarchy, constraints on parameter values can be identified, and relationships between parameters can be defined. Finally, as objects are modified, added, or deleted, regression testing rules can be applied to identify new test cases, test cases that need to be redefined, and test cases that no longer apply.

Let's examine a system that would be a good candidate for automated test generation using a command language. Consider a manufacturing plant with robotic machining devices. Each robot can use tools to cut, drill, and to shape various products. Suppose we have a command language interface to the robots. Commands perform such things as machine set up, tool selection, robot arm movement, and status reports. This system would be a candidate for automated test generation for several reasons. First, command language parameters hold the semantic content of the language. This would meet the first requirement for the command language. For example, the command "`ROBOT r23 DRILL using Drill-Bit2 at (100,100)`" explicitly identifies the command to execute, what drill bit to use, and the location of the hole to drill. We do not have to examine the contents of a file to understand the semantics of the command. Second, the parameters of the command language map to objects in the system. We imagine objects such as **Robot Tool**, **Coordinates**, **Drill**, **Router**, and **Saw** as possible objects. Furthermore, as commands are issued, the state of the system can be updated by examining the values of the parameters. In the "Drill" command example above, the robot state can be set to *Busy*. The objects, once arranged in a hierarchy, could identify parameter constraints. For instance, suppose the *robot-tool* parameter is currently set to **Drill**. Then, another parameter such as *drill-coordinates* may be constrained to prevent damage to another assembly or to obey safety rules for plant employees.

2.2.2 Background - Using Formal Language for Test Generation

The idea of using formal language definitions to automatically generate test cases is not new [BS82] [BF79]. Early research by Purdom shows how to efficiently generate sentences to test parsers [Pur72]. His research concentrated on generating sentences from a context-free language such that each production in the grammar is used at least once. His algorithms were presented to make sure "production coverage" is met with a minimal number of sentences. This method can also be used to debug grammars. In 1978 Payne developed a method to specify messages in a real-time system using a formal grammar [Pay78]. From the grammar, messages are automatically generated to perform "overload" tests on the real-time system. Along with encoding the syntax of the messages in BNF, Payne also associated probabilities with the terminals and non-terminals of the BNF. The probabilities were used to alter the frequencies of each syntactic unit.

Celentano *et. al.* extended the work by Purdom [CRV+80]. They designed and implemented an automatic sentence generator to test compilers. Using syntax-directed translation, their system could create (1) Totally incorrect, (2) Lexically correct, (3) Syntactically Correct, (4) Compile-Time Correct, and (5) Run-Time correct tests. The semantic rules for the tests were encoded in the grammar. Because their tests were based on Purdom's minimal production coverage criteria, the number of sentences in the test cases were manageable. Empirical results from testing a PL/1 compiler were successful. However, they also reported poor performance while testing an interpreter. The interpreter was defined as a finite state machine with several states and many transitions. Because the test generator tries to minimize the number of sentences, the test cases were too short and too simple to exercise the interpreter.

In a more recent paper, Duncan and Hutchison report findings from using an attribute grammar to automatically generate test cases for designs and implementations [DH81]. Their system generated test cases to compare implementations with their specification. Each test case listed the inputs of the test and it defined the expected output. The system could perform structural tests, module tests, and system tests. Empirical results were shown from (1) Testing conditional statements in Ada, (2) Testing a Sort Algorithm, and (3) Testing a Text Reformatter. Unfortunately, follow up interviews with Hutchison revealed that their initial concept did not work as well as planned. During test case generation, the combinatorial explosion of semantic rules was overwhelming. Because of the large number of rules maintained by the parse tree and on the stack, automatically generating test cases for arbitrary problems did not work [vM93c].

From the work cited above, we know that formal language can be used for automatic test generation. Not only can the syntax be represented but in some cases semantic information can also be used. For specific problem domains, tests can be generated quite efficiently. For more general problem domains, the combinatorial explosion of semantic rules makes automatic test generation impossible for reasonably sized problems.

2.2.3 Formal Language - Specification and Design Issues

Two results from early work with formal languages and test generation must be understood. Both influence design decisions for Domain Based Testing. First, the syntax and semantics of the

Attribute Grammar	
Production	Semantic Rules
$L \rightarrow E$**newline**	*print(E.val)*
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val$ x $F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **digit**	$F.val :=$ **digit**.*lexval*

W-grammar	
Metaproductions	$N :: n \mid Nn$
	$ABC :: a \mid b \mid c$
Hyper-rules	$s : Na \mid Nb \mid Nc$
	$nNABC : letterABCNABC$
	$nABC : letterABC$

Table 2.1: Attribute Grammar and W-Grammar Examples

command language must be represented for good, meaningful, high quality test cases. Second, using the grammar to handle all syntax and semantic rules is not a good design strategy.

It is almost trivial to generate sentences from a grammar with no regard to semantic content. While some of the sentences are meaningful tests, most of them will fail because they do not consider semantic rules for command preconditions, parameter choice rules, or command sequencing. In an early prototype for Domain Based Testing, we found that less than 50% of the test cases where worthwhile because so many of them violated semantic rules [Cra93]. To increase the "yield" on the number of meaningful commands in a test case, Domain Based Testing must handle both the syntax and the semantics of the command language.

The second result seems intuitive, but let's analyze the shortcomings of using a formal language for both syntax and semantics. Consider two types of grammars that allow semantic information to be encoded with the productions of the language. The first grammar is called a W-grammar and it is named after Aad Van Wijngaarden [CU84]. W-grammars encode semantic information by deriving language productions from higher level productions and a set of derivation rules. The second grammar is called an attribute grammar. Attribute grammars are a special form of syntax-directed translation where semantic rules are associated with each production, but the syntax defined by the BNF and the semantic rules are separate. Attribute grammars can evaluate semantic rules at each node of the parse tree from the bottom up (leafs to the root). Table 2.1 shows examples of an attribute grammar and a W-grammar [ASU86] [CU84]. The attribute grammar produces an evaluation of an arithmetic expression and the W-grammar generates sentences from the context-sensitive language $\{a^n b^n c^n \mid n \geq 1\}$.

From the point of view of a compiler writer, W-grammars can describe complex context-free grammars. But, the expressive power of W-grammars is not free because no general parsing algorithm has been defined for them. For our problem, automated test generation, we can relax some of the parsing issues. Compilers check to make sure strings are in the language defined by the grammar. For automated test generation, we need to solve the converse problem. Given a language defined by a grammar, generate strings of the language. Because sentence generation is easier, W-grammars could be used for automated test generation.

With the parsing problem resolved, we must also consider other design issues. For example, W-grammars encode semantic rules in the BNF of the language. We can assume that most test engineers can read and understand language syntax defined by BNF. In contrast, it may be difficult for them to develop a W-grammar that properly encodes the semantic rules for the system. W-grammars are not easy to write, and small mistakes in the grammar will generate incorrect sentences. Furthermore, combining semantic rules into the syntax of the language may not be prudent from the stand point of test case generation. Suppose a test engineer wants to turn off all or some of the semantic rules. Using a grammar to encode these rules would require the tester to rewrite the BNF without the semantic information. This is not an acceptable solution to our test generation problem.

One could argue that we could resolve some of these issues by abandoning W-grammars and moving to attribute grammars. An attribute grammar is a special form of syntax directed translation where semantic rules are associated with language productions. Because the two are separate, we can solve the problem of turning rules on and off by associating a Boolean Flag with each rule. On the other hand, previous research shows that syntax directed translation may not be a good design choice either. Duncan and Hutchison used an attribute grammar to automatically generate tests for arbitrary designs. Semantic rules of the grammar can be resolved with a bottom-up traversal of the parse tree. Unfortunately, they discovered that the number of semantics rules that had to be defined and maintained on the stack during parsing became overwhelming. This suggests that all of the semantic information should not be handled in a single mechanism during test generation.

2.2.4 How can we use Formal Languages for Domain Based Testing

The results of this background investigation reveals three useful features for Domain Based Testing. First, we can use experience from other researchers to analyze problem domains for Domain Based Testing. Next, a grammar can be used to specify the command language. Third, we know that the syntax specification and semantic rules should not be combined.

Not all problem domains are suitable for Domain Based Testing. We know that the problem must have a command language interface or one must be able to define a "command language" for the system. In addition to the command language, the parameters of the language must hold most of the semantic content of the language. For systems with predefined command languages, we may not be able to change the language such that it meets these requirements. On the other hand, if one is defining a "command language" for an Abstract Data Type (ADT) or a reusable software component, then these requirements may influence how the language is designed if one wants to use DBT.

We know from prior research, that specific problem domains are more suited to automatic test generation. As cited above, the more successful research into using formal language to automatically generate test cases has been associated with narrowly defined problems. At the other extreme, the literature shows that trying to use a grammar to generate tests for general problem domains does not work very well. Some reasons cited for this shortcoming is that the semantic rules overwhelm the problem because of combinatorial explosion of the number of rules that must be written, applied, and stored. Therefore, we will use grammars for test generation, but we will focus on the problem domain of software testing.

We also know that combining the syntax of the language and the semantic rules into one representation is not a good design. Celentano's results show that a phased approach to test generation worked well for compiler testing [CRV$^+$80]. Therefore, Domain Based Testing should split test generation into a reasonable set of phases such that each phase focuses on specific aspects of test generation. In addition, we would like the flexibility to turn semantic rules on and off. Encoding the rules into the grammar is cumbersome and it requires the test engineers to understand complex grammars such as W-grammars. Given this information, Domain Based Testing will use a grammar to define the *syntax* of the language and we will use other mechanisms to capture semantic rules.

2.3 Software Reuse

Over the past ten years, software reuse has been a topic of study and empirical test [Kru92] [BP89] [Big92]. Historically, reuse has been confined to shared libraries, reusable programs, and reusable components. Recently, software reuse concepts have been applied throughout the software life cycle. In general, reuse looks beyond single projects or systems. Instead, information common to a set of similar systems is exploited. Using knowledge about similar systems is a good idea from an engineering and from an economic point of view. To a software engineer, one can build complex systems from sets of proven building blocks. To the project manager, one can influence project costs, time, and schedule by using reusable software instead of "reinventing the wheel." Besides exploiting information common to a problem domain, applying software reuse methods creates a natural feedback mechanism. Every project that employs software reuse can be analyzed with respect to the current reuse knowledge base. The results of this analysis may identify new reusable components, it may identify a better way to catalog the existing components, or it may justify earlier reuse decisions.

For software reuse to be successful, one must be able to extract common information about a problem domain, specify the operations of the domain, and package the information such that one can build a new system based on the reuse knowledge. One way to capture this information is to perform a *Domain Analysis*. Prieto-Diaz defines Domain Analysis as, "a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems" [HC91]. The result of a Domain Analysis is called a *Domain Model*. Domain Models represent the reuse problem domain and they serve as a mechanism to create instances of the reusable components. Hooper summarized the importance of Domain Models when he stated, "Even more leverage is gained from reuse if domain analysis can derive common architectures, generic models, or specialized languages that characterize software in a special problem area" [HC91]. The *special problem area* for the proposed research is called Domain Based Testing (DBT). In the next sections, we present background information about a Domain Analysis and we show how Domain Analysis can be used for DBT.

2.3.1 Background - Domain Analysis

The name *Domain Analysis* was originally used by Neighbors in his 1981 PhD dissertation on software [HC91]. Since then, Domain Analysis has been associated with the development of reusable software components [Gom91] [HC91] [Tra92] [TCY93]. Some reuse is apparent while other reuse is more difficult to identify. For example, it is easy to imagine a collection of reusable data structures, sorting algorithms, and searching algorithms. These are general purpose programming tools and they can be used across a wide variety of problems. Hooper calls this "horizontal" reuse [HC91]. On the other hand, identifying a set of reusable components within a narrowly defined problem domain may be difficult. For instance, it seems plausible that a set of algorithms and libraries could be developed and reused to automate navigation for ships, cars, trains, and aircraft. This represents a "vertical" reuse problem.

Domain Analysis extracts information from narrowly defined, "vertical" reuse situations. The emphasis is on analyzing a family of systems [1] instead of one particular system. Domain Analysis often concentrates on those objects that are common in a problem domain. They are sometimes called "kernel objects." Then, optional or enhancements to the kernel objects are added to address

[1]A family of systems is a collection of systems that share common characteristics [Gom91]

the variations in the family of systems. Domain Analysis commonly includes a thorough analysis of the problem, a list of domain terminology, and descriptions of the entities and operations in the problem domain. Domain Analysis does not constrain the reuse engineer to a single analysis technique. Because reuse is applied across a wide variety of problem domains and because it is used at all phases of software development, one should choose the analysis method that best fits the problem.

The result of a Domain Analysis is called a *Domain Model*. According to Gomaa, "A Domain Model is a problem-oriented architecture for the application domain that reflects the similarities and variations of the members of the domain" [Gom91]. Similar to Domain Analysis, Domain Models are not constrained to a single representation. Many authors suggest that one should use a representation most natural to the problem. Some of the more popular ways to represent a Domain Model are listed below [BP89] [Gom91] [HC91].

1. Data Flow Diagrams
2. Natural Language
3. Entity Relationship Diagrams
4. Objects
5. Class Hierarchies
6. Thesaurus/Classification Scheme
7. Predicate Logic
8. Semantic Nets
9. Knowledge Based System
10. Predicate Logic
11. Production Rules
12. Frames

Sometimes more than one representation may be needed. Domain Models represent complex problems because they capture information that is common to a number of similar systems. Booch points out that "It is impossible to capture all of the subtle details of a complex software system in just one kind of diagram" [Boo91]. Therefore, more than one representation may be needed to fully specify a Domain Model. Using multiple representations is useful in many ways. For instance, multiple views may be needed for different phases of the life cycle or for different users. Multiple representations can also be used to build an abstraction hierarchy. Because abstraction is one of the most powerful tools to understand complex systems, it seems natural to use them for Domain Models. Multiple representations can also help with modifications or extensions to the Domain Model. Using multiple views of the problem, one can isolate changes to the Domain Model as new information is learned about the problem.

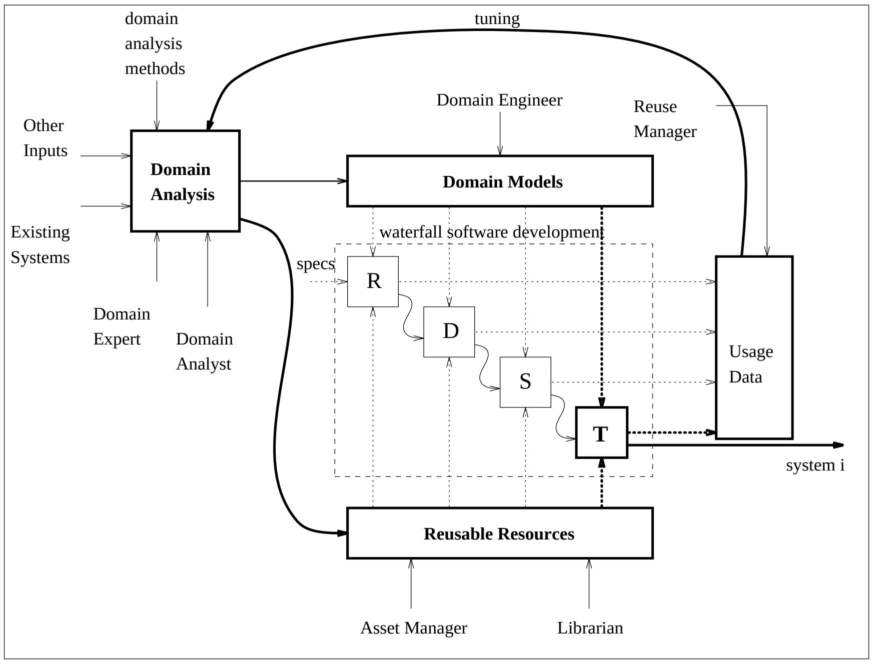


Figure 2.2: A Reuse Infrastructure [HC91]

2.3.2 How Domain Analysis Can be Used for Domain Based Testing

Domain Analysis provides two useful features that can be used by Domain Based Testing. First, Domain Analysis can be used to create a method to *reuse* test cases. Second, Domain Analysis and the resulting Domain Model serve as an infrastructure to combine all of the components of DBT into a single method. Software reuse can be used at every phase of a software life cycle. Prieto-Diaz suggested one "reuse infrastructure" for waterfall software development (see Figure 2.2) [HC91]. Even though this scenario uses waterfall software development, one could substitute another life cycle model.

As the figure shows, the result of a Domain Analysis is a Domain Model. Domain Models are subsequently used at each phase of software development. Notice that feedback loops are provided to Domain Analysis to update the Domain Models. For Domain Based Testing, we propose to focus on the *Testing* phase. In Figure 2.2, boxes highlighted with **BOLD** lines denote our narrowed focus. From this point of view, Domain Based Testing can be used as a simple test generation tool, a generator for "reusable" test case design, or to identify regression test suites.

In software reuse, Domain Analysis and Domain Models collect the many facets of software reuse under one model. Likewise, Domain Analysis and the resulting Domain Model serve as mechanisms to combine all of the components of Domain Based Testing into a single infrastructure. For instance, Domain Analysis provides the analysis tools needed to identify *what* needs to be represented in the Domain Model. Currently, DBT uses three different representations of the problem (1) Scripting (Command Sequencing), (2) Command Template Generation, and (3) Parameter Value Selection. This provides three levels of abstraction for the domain analyst and the test engineer. Command sequencing captures the dynamic behavior of executing a series of commands. This is a high level view of test generation where most test engineers will work to create test cases. At the command level, command syntax, semantic rules about the commands, and parameter value selection rules must be represented. At the lowest level of abstraction, parameter values must be represented and parameter value selection rules must be written.

Using three levels of abstraction is extremely useful. First, the domain engineer can analyze complex problems by viewing the system at multiple levels. Syntax and semantic information can be separated, and static and dynamic views of the system can be split. Second, it helps test engineers by allowing them to "ignore" low level details such as command syntax and parameter value choices. Instead, they can focus on high level test scenarios. Third, the design and implementation of Domain Based Testing can benefit from the multiple views of the system. For instance, one can substitute a new Script Generator into the system without disrupting the Command or Parameter phases. Finally, using an abstraction hierarchy for test generation makes it easier to isolate where changes and updates must be made as the problem domain evolves.

2.4 Requirements Analysis

The proposed research is based on creating a Domain Model for testing purposes. Many Domain Models, ours in particular, use an object oriented specification. To help us understand what we need to specify in the Domain Model, we borrow ideas from one of the more recent object oriented analysis techniques called Object Behavior Analysis (OBA). Developed by Rubin and Goldberg, OBA is an analysis method that emphasizes "behaviors" or what takes place in the system [RG92]. From the analysis, objects that initiate behavior can be identified, objects that offer services can be described, and the interactions among the objects can be specified. We do not need a full Object Behavior Analysis for Domain Based Testing. Rather, we extract from OBA what is needed to analyze objects, object relationships, and object behavior. We will use this information to perform a Domain Analysis and to create a Domain Model. In the next section, we provide a short overview of OBA and in the subsequent section, we describe how we use OBA for our proposed research.

2.4.1 Background - Object Behavior Analysis

Object Behavior Analysis is an object oriented analysis method that emphasizes object behaviors. It is an iterative, five step process with multiple entry points. The five steps are listed below,

0. Setting the Analysis Context
   - 0.1 Identify goals and objectives
   - 0.2 Identify appropriate resources for analysis
   - 0.3 Identify core activity areas
   - 0.4 Generate preliminary analysis plan
1. Understand the Problem
   - 1.1 Scenario Planning
   - 1.2 Scripting
   - 1.3 Build Glossaries
   - 1.4 Deriving Attributes
2. Defining Objects
   - 2.1 Generate Modeling Cards
3. Classifying Objects and Identifying Relationships
   - 3.1 Describe Contract Relationships
   - 3.2 Organize Objects into Hierarchies
4. Modeling System Dynamics
   - 4.1 Generate State Definition Glossaries
   - 4.2 Determine Object Life Cycle
   - 4.3 Determine Sequencing of Operations

The output of Object Behavior Analysis consists of (1) Scripts, (2) Glossaries, (3) Object Models, and (4) System Dynamic Models. Scripts record scenarios about how one uses the system. Glossaries capture definitions, ranges of values, and trace information for objects, object attributes, and object state information. Object Models are used to define structural and contractual relationships between objects. Finally, system dynamic models show object life cycles and sequence of operations.

Step #0 of Object Behavior Analysis identifies the goals, resources, primary areas for analysis, and a preliminary analysis plan. The authors use this step to make sure a clear set of objectives are stated, a clear boundary for the analysis is set, and a plan is established for the remaining analysis. Because these tasks are typically not included as one of the iterative steps in the analysis, it is labeled as step "zero."

In Step #1, system analysts develop *use scenarios* to understand the problem, to determine what the system is supposed to do, and with whom it it supposed to do it. By interviewing users and domain experts, knowledge about end-user and detailed system dynamics can be understood. Use scenarios are recorded in tables called scripts. From the scripts, the *parties* of the domain are defined. An *initiator* is a party that requests *actions* from other parties. A *participant* is a party that provides a *service* to other parties. Each script is recorded as a table of 4-tuples where each row contains an initiator, an action, a participant, and a service. The sequence of 4-tuples formally denotes a particular use scenario. Along with each script, preconditions and postconditions are listed. Preconditions denote what must be true for the script to be applicable, and the postconditions denote what is true after the script completes. Scripts can be linked together by matching preconditions and postconditions to show how actions might progress through the system.

The next step is to define attributes for the parties of the scripts. An attribute is a logical property of a party that is associated with the requirements to fulfill one or more of its contracts. The attribute may be needed to invoke a service or it may be used to fulfill the service request. Rubin and Goldberg use a table to list the relationships between the attributes of various objects [RG92]. They found that attributes were difficult to classify and define using the more popular diagraming techniques (i.e. ER-Diagrams). Instead, a table shows the required information where entries are augmented with a diagram if it is appropriate. The results from OBA Step #1 are sets of scripts that define use scenarios of the system, a set of entities called *initiators* and a set of entities called *participants*. All of this information is recorded in a Parties Glossary, an Attribute Glossary, and a Service Glossary.

The next step is to define the objects of the system. While many parties may be identified in the previous step, not all of them will be represented in the final system. Some of them will be external *analysis objects* and the others will be *system objects*. Each system object is recorded on an Object Modeling Card. The card names the object, shows relationships to other objects, and lists several analysis traces. The names used on the modeling cards must match the names used in the various glossaries.

In Step #3, relationships among the objects are identified. Typically, the relationships are hierarchical in nature where one object uses attributes or behaviors from other objects. By drawing these relationships, one can identify objects that need to be refined and objects that can be combined. Objects that are too complicated are split into two or more objects. This is called *factorization*. In contrast, some objects may offer similar services. Combining them through a common parent object is called *abstraction*. As new objects are defined through factorization and abstraction, one must iterate back through the scripts and use scenarios to annotate them accordingly. Object Modeling Cards must be written and the various glossaries must be updated. During the iterative process, the trace lists on the modeling cards are used to keep track of the original objects, the newly refined objects, and the abstracted objects. Rubin and Goldberg suggest that notations such as Booch's can also be used to denote the object relationships [RG92] [Boo91].

In Step #4 of Object Behavior Analysis, we define the dynamics of the system. In short we must model those parts that change over time. To do this, the states of the objects must be identified, the events that cause the system to change states must be listed, and the order in which the events take place must be known. States are associated with each object where a state represents a situation or condition of an object. Most object states can be identified from script preconditions and postconditions. States and state transitions are noted using a State Glossary. Other system dynamics must also be modeled. For example, we need to show the life cycle of the objects, how the object moves from state to state, and the events that cause the state transitions. Typically, this information is stored in state transition diagrams or state transition tables. The final step in OBA is to capture the ordering of the operations within a script. Operation order can also reduce the complexity of the problem. For example, if there are five steps in a script and order doesn't matter, then there are 5! or 120 ways to execute the script. If order matters, then we need to identify the operation order that is important.

2.4.2 How OBA Can be Used for Domain Based Testing

This short introduction to Object Behavior Analysis reveals how to conduct a formal object-oriented analysis for an arbitrary problem domain. Object-oriented analysis is important to Domain Based Testing because we must perform a Domain Analysis to create an object based Domain Model. Therefore, we can extract information from OBA about how to conduct a thorough Domain Analysis. Furthermore, we can tailor the concepts from OBA to meet the specific needs of Domain Based Testing. This makes sense because we no longer need to analyze an arbitrary problem. Instead, we are analyzing a problem for a specific intent, namely, automated test generation. To tailor OBA, steps may be eliminated, modified, or extended to meet the needs of DBT.

Object Behavior Analysis provides a good analysis foundation for Domain Based Testing in several ways. First, OBA shows how to represent both the static and dynamic nature of the problem. For the static view, OBA shows how to define objects, object attributes, and object relationships. For the dynamic view, OBA shows how to capture state information, state transition behavior, and operation sequencing. Second, OBA emphasizes its iterative, multiple-entry point approach. An iterative Domain Analysis and Domain Modeling works well.

We do not propose to use a complete Object Behavior Analysis during Domain Analysis. We need to borrow the useful parts of OBA and modify or extend other parts such that OBA is tailored to the automated test generation domain. In Table 2.2, the relationship between the concepts in

Object Behavior Analysis	Domain Based Testing
Objects	Analyze the Command Language
Object Attributes	More precise classification
Object Relationships	Object Hierarchy
	Parameter Value Constraints
Object State	State "Attributes"
Script	Command Sequencing Scripts
	Script Classes
Script Pre/Post Conditions	Pre/Post Conditions associated with commands
Script Operation Ordering	Order fixed by the script

Table 2.2: Comparison of Object Behavior Analysis and Domain Based Testing

Object Behavior Analysis and Domain Based Testing are listed. For instance, OBA shows how to identify objects of the system for an arbitrary problem domain. For Domain Based Testing, we will narrow our focus and select objects from the command language.

Identifying object attributes is a typical step in almost any object-oriented analysis (OOA) or object-oriented design (OOD). It is also used in Object Behavior Analysis. Likewise, Domain Analysis and Domain Based Testing will need to keep track of object attributes. But, the classification for attributes is too general for automated test generation. Therefore, DBT conducts a more detailed analysis of the object attributes where they are classified much more precisely.

Once objects and their attributes are identified, relationships between objects must be shown. OBA uses object relationships to split objects that are too complex and to conjoin objects that perform similar functions. During Domain Analysis, one may need to combine or split an object. Furthermore, Domain Based Testing will use object relationships to record semantic information about how to choose values for the parameters of the command language. For instance, relationships can be shown as an object hierarchy where the arcs are annotated with information about how parameter values from one object constrain parameter values of another object.

Scripting is an important part of Object Behavior Analysis. Scripts capture scenarios about how the system will be used. Almost every step in OBA is based on information in the scripts. Scripting will also be used in Domain Based Testing, but it will be used in a different way. OBA uses scripting as an analysis tool. Domain Based Testing will use scripts as a test generation mechanism where the dynamic behavior of the test scenario is captured. In Object Behavior Analysis, preconditions and postconditions are associated with each script. For Domain Based Testing, preconditions and postconditions must be represented, but they are defined at a lower level of abstraction. Therefore, DBT will associate preconditions and postconditions with individual commands.

In OBA, preconditions identify object state information that must be true before the command can be issued. Postconditions identify the state of the system after the script executes. For automated test case generation, object state information is needed for semantically correct command sequences. Therefore, during Domain Analysis, object states must be defined, state transitions must be identified, and the pre/post conditions for each command must be shown.

Finally, Domain Based Testing must define an order in which commands can be executed. The rules for generating command sequences is called scripting. Scripts in OBA and DBT are similar in that they capture dynamic behavior of the system. They are different with respect to the rules about the sequence of "operations" (commands) with the script. For Object Behavior Analysis, scripts are used to identify objects operations. For Domain Based Testing, the operations are defined *a priori* by the system's command language. In OBA, a separate step makes sure that script operations are analyzed with respect to their execution order. If there are no constraints among the operations, then any permutation of the operations is a valid command sequence for that script. For DBT, the execution order for the commands is *fixed* by the script. By fixing the sequence of commands within a script, test engineers control test case generation easily and the application of semantic rules becomes easier. Using scripts in this manner also allows us to create script classes such that the scripts can be combined, merged, and modified.

2.5 Software Testing - Test Case Design

Test Case Design refers to the various ways to choose a subset of all possible test cases. If exhaustive testing were possible, one would not have to consider test case design. Instead, we need ways to choose a subset such that each test has a high probability of detecting errors. One way to achieve this goal is to define several test case design strategies where each one focuses on particular types of errors. Because each strategy focuses on specific types of errors, one should use a collection of strategies to test software. In the next sections, a few test case design strategies are presented. Some of them provide information about new ways to employ Domain Based Testing.

2.5.1 Background - Structural Testing

White-Box Testing or Structural Testing is based on the internal structure of the program. Test cases are created by looking at the logic of the program's source code. Coverage measures are used to describe the degree to which test cases exercise the source code of the program. Coverage measures are classified according to the control flow or data flow of the program. For instance, the ultimate control flow coverage measure is to execute every path in the control flow graph at least once. Unfortunately, loop control structures make this criterion almost impossible to achieve. Even

Coverage Measure	Description
Statement Coverage - P_1	Execute every statement at least once
Branch Coverage - P_2	Each branch is traversed at least once
Condition Coverage	Each condition in a decision takes on all possible outcomes at least once
Decision/Condition Coverage	Each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each entry point is executed at least once
Multiple/Condition Coverage	All possible condition outcomes in each decision and all entry points are invoked at least once

Table 2.3: Some Control Flow Coverage Measures

for small programs with a single loop, the number of paths through the program is too large to test. Beizer calls this measure P_∞ [Bei90]. As one can imagine, there are many coverage measures ranging from P_1 to P_∞. As more paths are covered, one gets closer to P_∞. Some control flow measures are listed in Table 2.3 [Mye79].

Data flow testings is a unit testing method based on the data flow of the program. Data flow coverage uses the following definitions [vM93a] [CPRZ89].

`DEF(`x`)` is the set of vertices at which variable x is defined.
`USE(`x`)` is the set of vertices at which variable x is used.
`p-use` is a vertex at which variable x is used in a conditional branch statement
`c-use` is a vertex at which variable x is used in a computation

A *definition* of variable x is associated with a node when a statement at that node assigns a value to x. Variable x is *used* at a vertex whenever a statement at that node accesses x's value. Special cases of definition and use are sometimes needed. For example, *p-use* is a subset of the `USE(`x`)` set where x is used in a predicate or in a conditional branch statement. Using these definitions, Table 2.4 lists several data flow coverage measures.

Domain Based Testing does not have the luxury of examining the structure of the program. We must base all test case generation on the command language. Yet, we can borrow the idea of a *coverage measure* from White-Box testing. Instead, of basing coverage on control flow or data flow, we might be able to define Domain Based Coverage Measures from the structure of the Domain Model.

2.5.2 Background - Black-Box Testing

Another way to test software is to think of the program as a "black-box." Test cases are derived by examining the program's specification instead of its structure. For example, *Equivalence Partitioning* is a test case design strategy where the input domain is divided into equivalence

Coverage Measure	Description
All-Paths	Every path must be traversed
All-Edges	Every edge must be traversed
All-Nodes	Every node must be traversed
All-Defs	All Definitions must be traversed
All-Uses	At least one subpath from each variable definition to every p-use and every c-use of that variable definition must be traversed
All-C-Uses	At least one path from each variable definition to every c-use of that variable definition must be traversed
All-P-Uses	At least one path from each variable definition to every p-use of that variable definition must be traversed
All-DU-Pairs	Every simple subpath from each variable definition to every p-use and every c-use of that variable must be traversed

Table 2.4: Some Data Flow Coverage Measures

classes. Each equivalence class is defined such that any value in the class is representative of all values in the class. Choosing a value from an equivalence class can be used in a test case. If the results from the test are correct, then all inputs in the equivalence class are assumed to be correct. *Boundary-Value Testing* is a test case design strategy that tests conditions at the boundaries of the input domains and output ranges. For instance, suppose the valid set of input values is $\{0 \ldots 10\}$, then a boundary-value test would create a test suite with the following test cases $\{-1,0,5,10,11\}$. Boundary-Value testing can be used to test output ranges, too. Suppose the output of a function is in the range $\{0 \ldots 100\}$, then input values should be selected to generate outputs of 0 and 100. One should also try to select input values that would cause the output to be less than 0 or greater than 100. Boundary-Value testing is also useful when testing data structures. Consider a structure that has a capacity of 256 records. One should test the system with 0, 1, 256, and 257 records for proper behavior. Some of the Black-Box strategies can be based on the specification or on the structure of the program. For example, Boundary-Value testing could examine the program's logic to define boundary conditions. For this proposal, we will consider their Black-Box testing definitions.

2.5.3 Background - Regression Testing

Regression Testing is a testing strategy associated with evolutionary software development [Bei90] [Mye79] [vMO93] [Sne93]. Software is always undergoing change. New features may be added, old options may be deleted, and bugs may be fixed. As the software evolves, the tester must make sure the old feature still work and the "fixes" don't cause new problems. Most of the time,

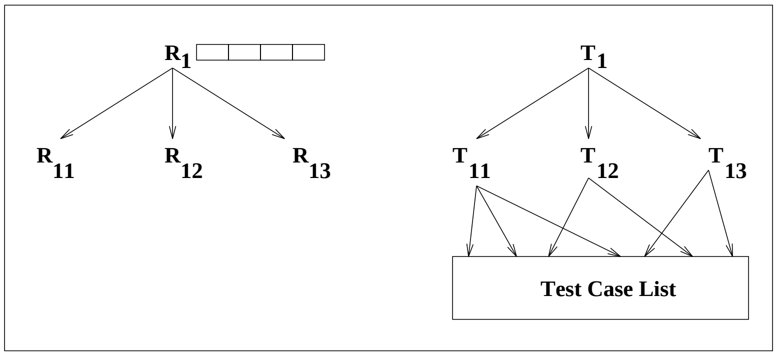


Figure 2.3: Requirements and Test Suite Hierarchies [vMO93]

it is economically infeasible to re-run all of the test cases from the original system. Therefore, one must choose a subset of test cases that have a high potential to detect errors.

The typical scenario for regression testing is to isolate the change in the software product. Test cases that are relevant to the modification are chosen. From this set, any test case that no longer applies can be deleted. Some of the existing tests may need modification. Finally, new tests are added to complete the "regression test suite." Von Mayrhauser and Olender defined rules for regression testing of requirement modifications [vMO93]. In their paper, requirements are represented hierarchically where a node represents a requirement and the children of a node represent subrequirements. Associated with the requirements hierarchy, a duplicate *Test Suite* hierarchy. Each requirement has a corresponding test suite. An example of this structure is shown in Figure 5.2 [vMO93]. Note that a test suite is comprised of a set of individual test cases. As the figure shows, test cases can be shared by more than one test suite. Qualitative requirements are represented as an attribute vector associated with individual requirements. Each entry in the attribute vector also has a corresponding test suite in the test suite hierarchy.

Using the two hierarchies, rules for building regression test suites are given. For instance, rules are defined for requirements addition, deletion, and modification. Suppose a new requirement is added to the requirements hierarchy. A path from the root of the tree down to the new requirement identifies all of the requirements that may need to be retested. Similarly, when deleting a requirement, the requirement and all of its children (subrequirements) are deleted. In addition, the test suites associated with the deleted requirements must be removed. The rules will be classified according to how aggressive or conservative they are.

2.5.4 How Software Testing can be used for Domain Based Testing

Three useful ideas from Software Testing can be used for our research. First, *coverage measures* can be extended to evaluate the quality of Domain Based Test generation. Second, Domain Based

Testing must be able to support higher level test case design, and third, Domain Based Regression Testing rules can be developed. We emphasize that *test criteria selection* is one facet of Domain Based Testing. It should not be confused with *test case generation*. Domain Analysis and the Domain Model concentrate on test case generation. Test criteria is handled separately through our Testing Process Model.

White-Box Testing uses coverage measures to evaluate the quality of a test case. While DBT does not use the structure of the source code to define test cases, DBT can extend the idea of a coverage measure by using the structure defined by the Domain Model. The Domain Model uses three levels of abstraction to specify test cases (1) Scripting, (2) Commands (object methods), and (3) Parameters (object elements). From this structure, one could define the following coverage measures:

- Object Coverage
- Path Coverage (with respect to the Object Hierarchy)
- Command Coverage
- Command Rule Coverage
- Script Coverage
- Script Rule Coverage

At first, we will measure the coverage of test cases generated by DBT and correlate the results with error detection. From these results, worthwhile coverage measures can be identified. The results may also be helpful in defining improved coverage measures for DBT.

The second way Domain Based Testing can benefit from existing work in Software Testing is to support a variety of Black-Box test strategies. A single test case design strategy is not sufficient for thorough testing. Because each test criteria focuses on particular types of errors, DBT must not force test engineers into one test strategy or methodology. Domain Based Testing is a Black-Box testing method. We do not have the program's logic nor the implementation to guide our testing effort. DBT must rely on the command language interface to generate test cases. Because the command language is parameterize, the test engineer can change test cases by modifying the semantic rules, or by changing the set of parameter values. Some of the Black-Box strategies that DBT could support include *valid/invalid* test cases, *restricted input domain* tests, (boundary-value/partitioned tests), *special input/output* test cases, and *error handling* tests.

The third way DBT can benefit from Software Testing research, is to leverage from the well-defined approach to regression testing as outlined above. DBT uses a Domain Model to structure test case generation. Part of the model is an object hierarchy. The hierarchy identifies relationships between objects and it defines rules for selecting parameter values. In the regression testing technique described above, requirements and test suites are represented hierarchically. Combining these two ideas, Domain Based Regression Testing can be defined. First, the object hierarchy can be substituted for the requirements hierarchy, and test suites can be associated with objects of the system. We can also replace the qualitative requirements vector with the object elements. From this new structure, regression test rules can be applied to handle changes in the Domain Model. For instance, a new object may be added to the model, objects may be deleted from the hierarchy, or an object may be modified. For all three cases, rules can be identified to generate regression test suites.

2.6 Comparisons between Domain Based Testing and Other Methods

2.6.1 Domain Testing

Boris Beizer describes a test strategy called *Domain Testing* [Bei90]. Even though the name is similar to our proposed research, the two testing methods are not the same. Domain Testing views programs as mathematical functions where input values are classified as correct or incorrect. Programs accept input vectors and partition them such that similar cases take similar processing paths. Invalid input can be handled by adding an "error" processing path or an "input vector rejected" path. Domain Testing can be used for functional or structural testing. If tests are based on a specification, then it is functional testing. If tests use internal program structure, then it is a structural method. Most Domain Testing theory is based on structural methods.

Domain Testing earned its name because programs are considered to be input vector classifiers. Input vectors form the *domain* of the program in the mathematical sense. Each domain is specified with a set of boundaries. Each boundary is defined using at least one predicate. The predicate returns `TRUE` if the input vector is a member of the domain, otherwise it returns `FALSE`. Beizer uses the following example [Bei90]:

IF `x > 0` **THEN** `ALPHA` **ELSE** `BETA`

Numbers greater than zero belong to the **ALPHA** domain and numbers less or equal to zero belong to the **BETA** domain.

Domain Testing identifies errors in the classification of input vectors. For example, suppose the `if-statement` above used `‘‘>=’’` instead of greater-than. Proper test vector selection, would misclassify input value `x=0`. Domain Testing assumes the program processes the data correctly, but it may misclassify input vectors. It also assumes that once the input vector is correctly classified, then the input will be processed correctly. To detect processing errors, Domain Testing should be used along with other testing strategies.

Our research is called *Domain Based Testing*. Its name is derived from the Domain Analysis and Domain Model used to represent the system under test. Domain Analysis and Domain Models have been used in the Software Reuse communities for the past ten years. Domain Based Testing uses ideas from software reuse to develop a testing method. Until a more descriptive name can be derived, we will use Domain Based Testing for this research.

2.6.2 Category-Partition Testing

Category-Partition Testing is a test case design strategy created by Ostrand and Balcer [OB88]. We present this design method separately to compare and contrast it with Domain Based Testing. Category-Partition testing creates *functional test suites* from specifications. The specification can be written in a formal language or in a natural language. The goal of Category-Partition testing is to define a systematic approach to writing *test specifications* and to generate test case descriptions using an automated tool. The results of this approach is to generate tests with a good coverage of the input domain, tests with good error detection, and test suites with a reduced number of test cases.

The five steps for category-partition testing are listed below. Steps A through D create the test specification and Steps E and F generate the test suites.

A. Analyze the specification.

B. Partition the categories into choices.

C. Determine constraints among the choices.

(Step **C** and the following two steps frequently occur repeatedly as the tester refines the test specification to achieve the desired level of functional tests.)

D. Write and process test specification

E. Evaluate generator output.

F. Transform into test scripts.

The first step in Category-Partition Testing is to analyze the specification. The test engineer identifies *functional units* that can be tested independently. For each functional unit, input parameters and environmental conditions are identified. An input parameter is an explicit input to the functional unit, and an environmental condition is a system state needed at the time of executing the test. For each parameter and environment condition, the tester makes a third pass through the specification to define *categories.* A *category* is "... a major property or characteristic of a parameter or environment condition" [OB88]. Categories identify how the functional unit behaves when presented with respect to parameter values and conditions. Consider a functional unit with an array data structure. The categories for this unit may include {*array size*, *type of element*, *maximum element value*, and *minimum element value*} All of the information in Step A can be derived from the specification.

In the second step, the tester partitions the categories into *choices*. Choices are sets of similar values where any value in the set is representative of all values in the set. This classification is similar to equivalence partitioning where the input domain is classified according to equivalence classes. Test cases will be built from choosing among the list of choices from each category. While identifying choices, the tester is not restricted to the specification. One can use past experience, error conditions, special cases, or the structure of the source code. Consider the category *array size* from the previous example. Choices may include $\{0, 1, 2\text{-}100, >100\}$.

Step C identifies constraints between choice values. Parameters and environment conditions have relationships to one another. Sometimes choices for one parameter may influence choices for others. Without restrictions, the total number of test cases would equal the product of the cardinality of each category set, $\prod_i \mid category_i \mid$. In most applications, this would generate too many tests, and many of the tests would be meaningless because choices may conflict. To resolve both issues, test engineers identify constraints between choices. During test generation, the total number of test cases is reduced and many of the meaningless tests should be eliminated. Using the same array example, suppose the *array size* choice = 0. Then, it is meaningless to generate a test with choices for *maximum element value* and *minimum element value*.

In Step D, a formal Test Specification is written for each functional unit. The format of the specification captures the categories, environmental conditions, choices, and constraints. From the specification, a generator produces templates called *test frames*. Each frame "consists of a set of choices from the specification, with each category contributing either zero or one choice" [OB88].

Step E allows the test engineer to examine the test frames, modify the test specification, and to generate a new set of test frames. This may be needed because some tests are missing, some tests may need additional constraints, and sometimes too many test frames are generated. If changes are needed, then the tester returns to Step D.

In the last step, the tester converts the test frames into test cases. Currently, the test frames specify the conditions for each test, but it does not write the complete test case. The tester must write (by-hand) the actual program, set of commands, set up data files, and environment conditions for each test frame. After the test cases are written, the tester can organize them into groups called *test scripts*.

Table 2.5 cites some similarities and differences between Category-Partition Testing and Domain Based Testing. Consider the goals and objective for each method. Category-Partition Testing creates functional tests from a specification. Each script tests a single functional unit by automatically generating test frames from a formal test specification. The total number of test cases is reduced by using constraints. In contrast, Domain Based Testing creates test scripts from a "command language." Scripts contain a sequence of syntactic and semantically correct commands. DBT does not focus on a single test case design strategy. Instead, test engineers are able to create a wide variety of test cases. No restrictions are placed on the number of tests that are generated. Because of this freedom, one would be able to create *stress* tests easier with DBT than with Category-Partition Testing.

Each method also uses different Test Generation Models. Test generation for Category-Partition testing iteratively refines the formal test specification until the number of test frames is acceptable and the input domain coverage is sufficient. The actual test cases and the compilation of the test scripts are hand written. Domain Based Testing provides a testing process model with three levels of test case generation. Testers start at the script level where a list of command names is generated. At this level, the test engineer is free to specify command sequencing, the number of commands to generate, and the types of commands. From this list of commands, the second phase fills in each command name with a syntax template. This template does not specify parameter values. In the last phase, a complete test case is generated by filling in parameter values.

Despite these differences, Category-Partition Testing and Domain Based Testing share some things in common. For instance, both methods use constraints to generate meaningful test cases. Category-Partition employs constraints to denote that one choice from a category cannot be used in the same test frame with other choices from other categories. Test frames are generated by enumerating all possible combinations of choices. Many of these frames are meaningless because of conflicts between choices. Using constraints to eliminate these frames, Category-Partition testing reduces the total number of tests. On the other hand, Domain Based Testing uses constraints to generate semantically correct commands. For instance, choosing the value of one parameter may influence the choice of another parameter. DBT is more flexible in the types of constraints that can be applied. For instance, parameter values can be eliminated from being selected, we can reduce the set of valid choices, and we can force parameters to the value of a previously bound value.

2.7 Summary

The foundation for Domain Based Testing is build from four areas of computer science (1) Formal Languages, (2) Software Reuse, (3) Requirements Analysis, (4) Software Testing. In this chapter, background information about each area was presented. Along with this survey of existing work, we described how each area will be used for our proposed research. In summary, Domain Based Testing is based on performing a specialized Domain Analysis and creating a Domain Model

	Category-Partition	Domain Based Testing
Goals	Functional Testing Reduce Number of Test Cases Specification Based	Wide Variety of Test Case Design Strategies Syntactic/Semantically Correct Commands Command Language Based
Generation	Automate Test Frame Generation Generate all possible choice combinations Test Cases hand written Test Suites hand written	Completely Automated Three Levels: Scripts, Command, Parameter
Constraints	Limit Number of Test Frames Remove meaningless Test Frames Error and Special cases Constraints Local to one Functional Unit	Three Levels : Script, Command, Parameter Generate Semantically Correct Tests Variety of Semantic Rules

Table 2.5: Comparison of Category-Partition Testing and Domain Based Testing

for the system under test. The Domain Model represents the problem at three levels of abstraction (1) Scripting, (2) Command Template Generation, and (3) Parameter Value Selection. Scripts define the dynamic behavior of the system by recording the sequencing of commands. At the command level, a grammar (BNF) defines the syntax of the command language. Semantic rules are recorded as pre/post conditions for each command, and parameter value selection rules are defined for each command. The Domain Model describes all of parameters of the command language using objects and an object hierarchy. From this hierarchy, rules for choosing semantically correct values for the parameters can be derived. Domain Based Testing is not limited to test case generation. Because of its structure and levels of abstraction, DBT also makes a good platform for *test reuse* and *regression testing.*

Chapter 3

DOMAIN BASED TESTING

3.1 Introduction

Domain Based Testing (DBT) is a software test method based on Domain Analysis and Domain Modeling. A DBT specification requires three closely related components (1) Object Model Definition, (2) Command Language Definition, and (3) Scripting Definition. In this chapter, we present the steps for analyzing a system under test and we show how to create a Domain Based Test Specification. Table 3.1 lists the steps for conducting a Domain Analysis for DBT. Even though the steps are listed in sequence, one should apply them iteratively. The results of the specification is a Domain Model from which test cases can be automatically generated.

In the remainder of the chapter, we *define* Domain Based Testing, *what* must be analyzed when building a Domain Based Testing system, and *what* functions are needed for automated test generation. All static and dynamic behaviors of the software product are considered. We encourage a thorough analysis of the problem as we present it here. Some domains may not need all aspects of a generic specification. Rather, we can tailor it for a given software product and its domain. Throughout the chapter, we will use a simple robot manufacturing system as an example.

3.2 Command Language Analysis

The first step is to analyze the "command language" of the system under test. Domain Based Testing uses a "command language" to automatically generate test cases. Many systems have a command language interface, and sometimes a "command language" can be defined for the system. For example, editors such as *emacs* and *vi* use commands to move the cursor, insert characters, and manipulate files. These represent systems with predefined command languages. Software components such as abstract data types (ADTs) and reusable software components do not have formal a command language interface, but their interface specifications could be used to define one for test purposes. In object oriented systems, classes define templates for objects. Methods or operations exported by the class also represent a form of "command language."

1. Command Language Analysis
1.1. Identify/Define a Command Language Interface
1.2. Check Semantic Content
1.3. Check Parameter to Object Mapping
1.4. Create Command Language Glossary
2. Object Analysis
2.1. Identify Objects and Their Elements
2.2. Identify Object Relationships
2.3. Create Object Glossary and Object Element Glossary
3. Command Definition
3.1. Command Language Representation
3.2. Identify Pre/Post Conditions
3.3. Identify Intracommand Rules
4. Script Definition (Command Sequencing)
4.1. Script Analysis
4.2. Script Classes
4.3. Script Rules

Table 3.1: Domain Analysis Steps for Domain Based Testing

Once the command language is identified or defined for the system under test, it must also be analyzed for semantic content. As described in the background chapter, most of the semantic information should be encoded in the parameters of the command language. If the semantic information is not encoded in the parameters, then other test strategies must be used to test the software. Consider a compiler. Most of the semantic content is stored in the source file and not in the command line parameters used to invoke the compiler. Effective tests for the compiler would focus on the syntax and semantics of the programming language instead of the compiler's command language interface.

In addition to semantic content, the parameters of the command language should map to objects in the system. As pointed out in the background chapter, this mapping is needed to keep track of object instances, the state of the objects, and the state of the system. Without a good mapping between the parameters and objects of the system, we may not have enough information to generate semantically correct commands.

Let's examine a command-based system that would be a good candidate for automated test generation using a command language. Consider a manufacturing plant with robotic machining devices. Each robot uses tools to cut, drill, and to shape various products. Suppose we have a command language interface to the robots. Commands perform such things as machine set up, tool selection, robot arm movement, and status reporting. Because command language parameters hold the semantic content of the language, and because the parameters map to physical objects, this system would meet the initial requirements for Domain Based Testing.

Command	Description
`DRILL` <*robot-id*> `USING` <*drill-bit*> `AT` <*coord*>	Drill a hole
`SAW` <*robot-id*> `USING` <*saw-id*> `FROM` <*coord1*> `TO` <*coord2*>	Saw from Point-1 to Point-2
`SET TOOL FAILURE` <*robot-id*> { `Shutdown` | `Standby` }	Set operating Mode
`POWER` <*robot-id*> { `Online` | `Offline` }	Turn a robot On/Off
`ACCEPT` <*robot-id*> `FROM` <*conveyor-id*>	Accept a new job
`RELEASE` <*robot-id*> `TO` <*conveyor-id*>	Release a job

Table 3.2: Robot Manufacturing Commands

Table 3.2 lists commands for the robot manufacturing plant. We will use these commands to demonstrate each step in the DBT Domain Analysis. From this list, a *command glossary* should be created. Information about the command, a short description, and the syntax of the command should be listed. For example, one entry for the robot manufacturing system command glossary is shown in Table 3.8.

Command Name	`DRILL`
Syntax	Drill-Cmd ::= `DRILL` <*robot-id*> `USING` <*drill-bit*> `AT` <*coordinates*>
Description	Instruct a robot to drill a hole using a specific drill bit at a particular position.

Table 3.3: Command Glossary Entry for the `DRILL` command

3.3 Object Analysis

Object Analysis is the second step in the DBT Domain Analysis. The results of this step identify *objects* of the system, *object elements*, and *relationships* between the objects. We use object oriented analysis to specify the problem domain because it is a popular way to represent Domain Models for software reuse [BP89] [Gom91] [HC91]. Because object models tend to focus on the "problem" or "problem space", they are excellent specification tools [Boo83] [Boo91] [RG92]. Domain Based Testing will use the results of this analysis to choose semantically correct parameter values and we will use its structure for Domain Based Regression Testing.

3.3.1 Objects and Their Elements

The next step in developing a specification for Domain Based Testing is to define objects and their elements of the problem domain. Typically, objects denote physical or logical entities. In Object Oriented Analysis/Object Oriented Design (OOA/OOD), analysts and designers use a variety of rules to identify objects [Boo83] [Boo91] [Mul89]. Domain Based Testing focuses on the *command language* and *user documentation* for object identification. Domain Based Testing uses

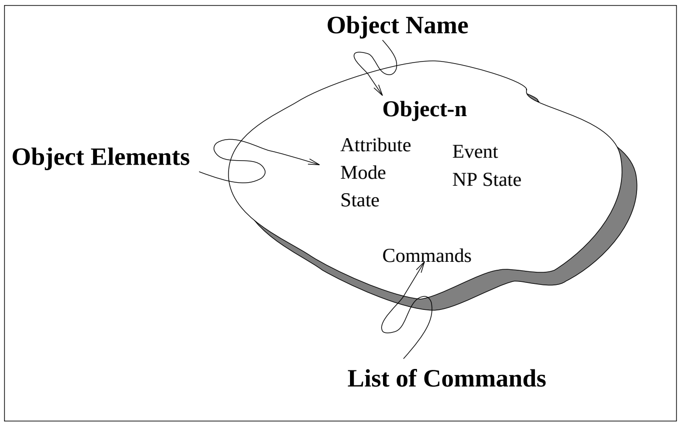
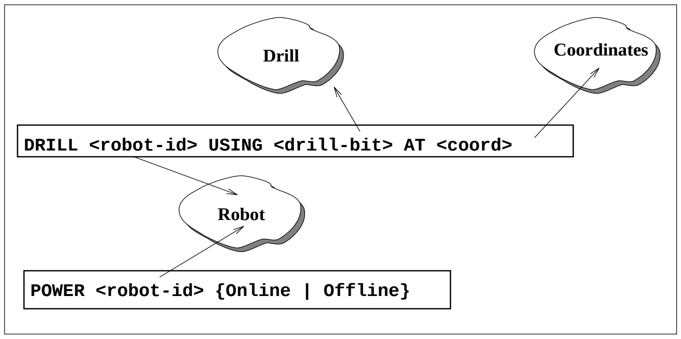


Figure 3.1: Anatomy of an Object



Drill
Coordinates
DRILL <robot-id> USING <drill-bit> AT <coord>
Robot
POWER <robot-id> {Online | Offline}

Figure 3.2: Analyzing Robot Manufacturing Commands for Objects

the term *object* to describe the template or type-definition of the problem domain entity. When a specific entity is needed, we refer to the entity as an *object instance*. Figure 3.1 shows a generic object blob. Each object has a name, object elements, and commands. In the next few sections, each part of the object will be defined in more detail.

For command-based systems, we can easily identify parameters of the command language and then identify whether they are part of or describe a property of a physical or logical object. This gives us a first cut of the objects and their particular attributes. Figure 3.2 shows two commands from the robot manufacturing example. The two commands identify three objects **Robot**, **Drill**, and **Coordinates**. The first two objects represent physical entities while the third represents a logical object. Analyzing other commands reveal additional objects such as those listed in Figure 3.3.

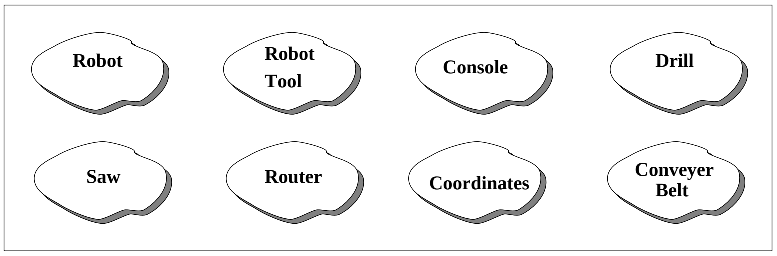


Figure 3.3: Objects for the Robot Manufacturing Example

Object	**Robot**
Description	Represents the type-definition for a manufacturing robot
Commands	`DRILL, POWER, SET TOOL FAILURE, ACCEPT, RELEASE`

Table 3.4: Object Glossary Entry for the **Robot** Object

From this analysis, an *Object Glossary* is created. The glossary should list the object names along with a short description of the object. We also list the commands associated with each object. From OOA/OOD, these commands act as "methods" for our objects. Table 3.4 shows an *Object Glossary* entry for the **Robot** object.

The next step in the Domain Analysis is to identify *object elements*. Object elements are similar to the concept of *object attributes* in OOA/OOD. In traditional object oriented analysis and design, one must identify the attributes of each object. Attributes define qualities and properties of the object. Attributes may place constraints upon an object such as limiting the range of possible values, forcing the selection of a particular value, or indicating dynamic behavior of the object. Rarely do OOA/OOD methods refine the concept of an object's attribute. But, for Domain Based Testing we found that classifying attributes more precisely simplifies test generation. We call the information associated with an object its *object elements*.

Domain Analysis for DBT classifies object elements into five mutually exclusive categories as shown in Figure 3.4. First, object elements are categorized as (1) *parameters* or (2) *non-parameters*. *Parameters* are those object elements that appear as parameters in the command language. Conversely, *non-parameters* do not appear as parameters of the command language. Each element can be split into more detailed categories. For example, *parameter attributes*, *mode parameters*, and *state parameters* identify specific types of parameters in the command language. Our classification *parameter attribute* should not be confused with the general classification *object*

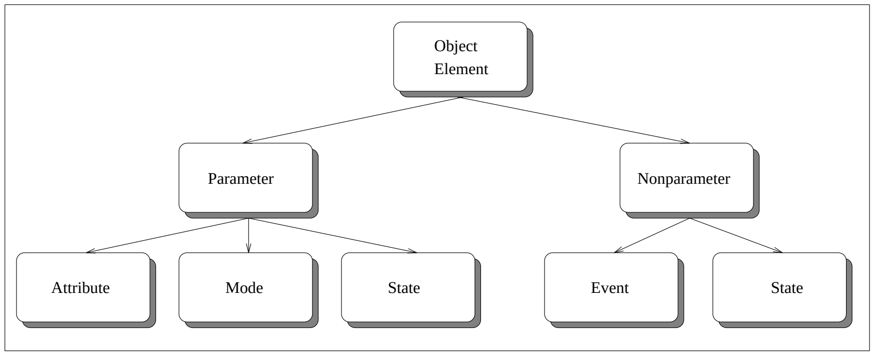


Figure 3.4: Object Element Classification

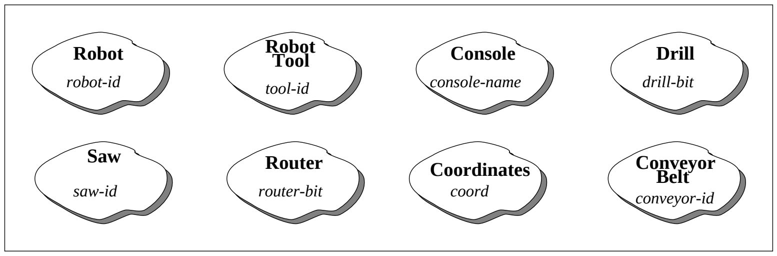


Figure 3.5: Parameter Attributes

attribute used in OOA/OOD. Our classification is much more specific. Short discussions of each object element is presented in the next few sections. We describe the general concepts first and then give an example to reinforce the idea.

Parameter Attribute

In Figure 3.4, an *attribute* is classified as a parameter of the command language. *Parameter Attributes* uniquely identify instances of objects. For example, the **Robot** object represents an object template and the *robot-id* parameter identifies a particular robot. Each object may have one or more parameter attributes, although in our analysis most objects have only one. Recall the robot manufacturing example above. After analyzing the command language for objects, one could identify the following parameter attributes, *robot-id, tool-id, console-name, drill-bit, saw-id, router-bit, coord,* and *conveyor-id* (see Figure 3.5).

Mode Parameter

A *mode* is a parameter of the command language that sets an operating mode or a warning mode for an object. An operating mode defines how the system behaves when an error occurs, or when an event takes place. A warning mode can identify the type of warning messages, where the

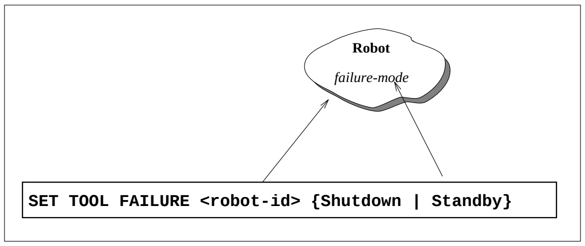


Figure 3.6: Mode Parameter - `SET TOOL FAILURE` Command

warnings appear, or the conditions for issuing warnings. Because they appear as parameters of the command language, operating modes can be changed by issuing the appropriate command.

In some problem domains, it is important that the current value for the mode is stored explicitly. Suppose the mode parameter for an object instance is set to `Manual`. It may be semantically incorrect to issue another command to set the mode to the same state (i.e. `Manual`). Therefore, we must keep track of the object instance and its mode parameter value. The current value must be eliminated as a choice when generating commands that influence the mode parameter.

Returning to the robot manufacturing example, consider the `SET TOOL FAILURE` command (see Figure 3.6). This command sets an operating mode for a particular robot. If a tool fails, then the robot will automatically `Shutdown` or it will go to a `Standby` mode.

State Parameter

A *state parameter* is a parameter of the command language that holds the state of the object. Because *state parameters* are part of the command language, one can change their value by issuing a command with an appropriate parameter value. Parameter state is important semantic information for test case generation. For instance, an object may need to be in a particular state as a precondition for a command or instruction. If the object is not in the proper state, then an appropriate sequence of commands can be issued to ensure semantic correctness. Similar to the *mode* parameter, we may need to explicitly store the *state parameter* along with each object instance.

Returning to the robot manufacturing example, suppose we can turn a robot on or off with the with the following command,

```
POWER <robot-id> { Online | Offline }
```

Furthermore, assume that a robot must be `On` before issuing any drill commands. By associating a state parameter, *robot-status*, with the **Robot** object we can make sure semantically correct commands are executed. Consider robot “r23”. Suppose it is currently turned `Off` (*robot-status* = `Offline`), and we need to issue a drilling command. The following sequence of commands will make sure the test case is semantically correct.

```
POWER r23 Online
DRILL r23 USING Drill-Bit2 AT (100,50,20)
```

Nonparameter Event

A *Nonparameter Event* is an event that cannot be controlled by parameters of the command language. Typically, an *event* is the result of the dynamics of the problem domain. Even though the command language cannot control *events* explicitly, we may need the information to generate semantically correct sequences of commands.

Consider the **Conveyor Belt** object in the robot manufacturing problem. One *event* for this object would be a boolean flag that signals a conveyor belt `FULL`. Suppose a robot is busily accepting new jobs for processing and releasing those jobs to the same conveyor belt. It would be semantically incorrect to release another job to the conveyor once it becomes full.

Nonparameter State

Nonparameter State elements do not appear as parameters of the command language. They represent object state that cannot be set through a parameter choice. They are results of the side-effects of executing a command or a sequence of commands. Experience shows that nonparameter state elements change as a postcondition of a command or a script.

Like the state parameter, nonparameter states are needed to make sure preconditions for a command or a sequence of commands are satisfied. The difference between the two is state parameters can be changed by issuing a command and choosing the appropriate parameter value. Nonparameter states are set as a result or a side-effect of executing a command or a sequence of commands. It may not be possible to change the value of a nonparameter state element on demand. Most likely, nonparameter state elements are used to identify which object instance are a valid parameter value choices.

Suppose the following command is issued: `DRILL r23 USING Drill-Bit2 AT (100,50,20)` The results of executing this command places the robot “r23” and “Drill-Bit2” in a `BUSY` state. Therefore, two nonparameter state elements *robot-state* and *drill-state* are associated with the **Robot** and **Drill** objects, respectively. If the preconditions for another command require the robot to be `AVAILABLE`, then robot “r23” cannot be used as a parameter value in the next command. This example shows how the dynamic behavior of object instances can influence test case generation.

Object	**Robot**
Description	Represents the type-definition for a manufacturing robot
Commands	`DRILL, POWER, SET TOOL FAILURE, ACCEPT, RELEASE`
Parameter Attribute	*robot-id*
Parameter Mode	*failure-mode*
Parameter State	*robot-status*
Nonparameter Event	
Nonparameter State	*robot-state*

Table 3.5: Object Glossary Entry for the **Robot** Object

Object and Object Element Glossaries

To complete this phase of the Domain Analysis, we must update the Object Glossary and we must create an Object Element Glossary. The Object Glossary maintains information about each object. Initially, each entry records the name of the object and the list of commands associated with the object. At this point, object element information needs to be added. Table 3.5 shows the **Robot** entry from the Object Glossary.

A set of glossaries are also created for each object element. This glossary includes detailed information about each element. While writing the object element glossary, it is crucial to analyze the range of values for each element and the way the values are represented. Such detailed information will be needed for design decisions, implementation decisions, and test case generation. Table 3.6 shows several entries from the Object Element Glossary.

3.3.2 Relationships Between Objects and Their Elements

For command-based systems, we can easily identify parameters of the command language and then identify whether they are part of or describe a property of a physical or logical object. This gives us a first cut of the objects and their elements, but not relationships between them. In Domain Based Testing, relationships between objects define semantic rules about parameter values. These relationships are captured in an *object hierarchy.* Figure 3.7 shows an arbitrary object hierarchy with four objects. The relationship between objects is shown by an arrow from one object to another. For Domain Based Testing, these arrows denote *parameter inheritance rules.* In the figure, **Object1** may influence parameter values in **Object2** or **Object3** or **Object4**. Detailed parameter relationships are shown as labels on the arcs. In the example, `element1` from `object1` has a relationship with `element2` of `object2`. In the following sections, we discuss the types of relationships between objects, we suggest ways to make the inheritance rules uniform, and we

Parameter Attribute		
robot-id		
	Full Name	Robot Identifier
	Definition	Names an Instance of robot
	Values	Range = r00...r99
	Object	Robot
	Representation	Range of values
	Number of Values	1 to 100
Parameter Mode		
failure-mode		
	Definition	Stores the failure mode of a robot
	Values	Shutdown | Standby
	Object	Robot
	Representation	Enumeration
Parameter State		
robot-status		
	Definition	Indicates whether a robot is on or off
	Values	Online | Offline
	Object	Robot
	Representation	Enumeration
Nonparameter Event		
conveyor-full		
	Definition	Indicates whether conveyor belt is full
	Values	TRUE | FALSE
	Object	Conveyor Belt
	Representation	Boolean
Nonparameter State		
number-jobs		
	Definition	Number of jobs currently on the conveyor belt
	Values	Natural Numbers = 0 ... Max Capacity
	Object	Conveyor Belt
	Representation	Integer

Table 3.6: Object Element Glossary

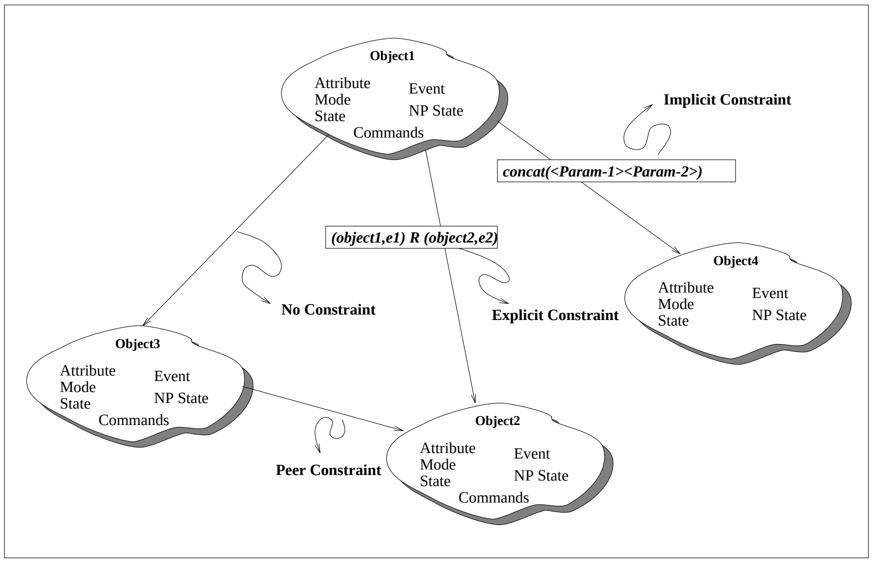


Figure 3.7: Generic Object Model

Type	Representation	Description
No Constraint	No annotation on the arc	Choices for the first parameter do not constraint choices for the 2nd
Explicit Constraint	$a \rightarrow b$	Parameter a constrains b
Implicit Constraint	concat(a,b)	Parameter is created by a concatenation of values
Peer Constraint	Arrow between objects	Relationship between objects at the same level
Explicit Concatenation	a `AND` $b \rightarrow c$	Explicit Constraints spanning more than one level of the hierarchy
IntraObject Constraints		Split the object into two objects

Table 3.7: Types of Relationships

provide rationale for the object hierarchy. Figure 3.8 shows the object hierarchy for the robot manufacturing example. This hierarchy will be used to explain some of the object relationship rules.

Types of Relationships

A wide variety of relationships between objects exists, and Table 3.7 summarize the types of rules that have been identified. The first relationship, *No Constraint*, is denoted as an arrow from an object at one level of the hierarchy to an object at the next lower level. No annotations are placed on the arc. Sometimes, objects may be related physically. For example, one object may be constructed from several instances of another object. Despite the physical relationship parameters from the higher level object may not constrain parameter values at the lower level.

The second type of relationship is called and *Explicit Constraint*. This is the most common relationship found in our preliminary research. The relation states that the value of an attribute

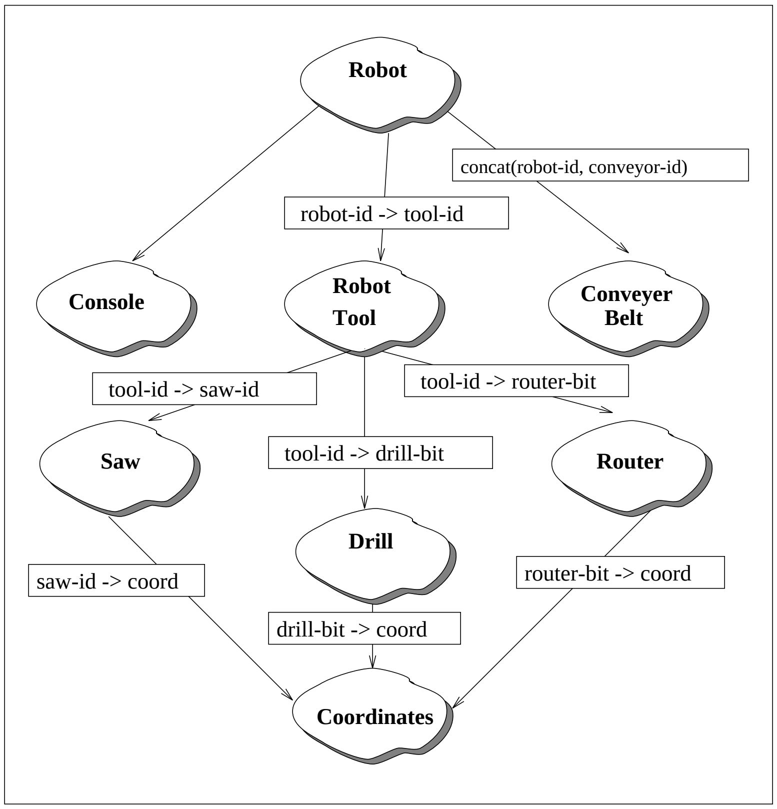


Figure 3.8: Robot Manufacturing Example - Object Hierarchy

from one object constrains the choices for another object. Because this is so common, we use the notation $a \rightarrow b$ to denote parameter a constrains the choices of parameter b. During parameter value selection, the test case generator must explicitly handle this rule. It should be noted that the *explicit constraint* does not address constraints that span more than one level. In the robot manufacturing object hierarchy, *robot-id* constraints the *tool-id* parameter. Suppose robot "r23" is selected in a command. The robot might be a special purpose machine with various sawing tools, but it does not have any drills or router bits.

The third relationship, *Implicit Constraints*, are a special case of an *explicit constraint.* They are actually resolved by the syntax of the command language, but we present them here for completeness. Sometimes parameter values are constructed by concatenating information from a higher level with information at the current level. Through concatenation, parameter inheritance is implicitly carried down to lower level objects. Suppose the *conveyor-id* is a concatenation of the *robot-id* and the conveyor belt number. For instance, "r23c2" represents robot "r23" and conveyor belt "c2." Because this constraint is resolved by the syntax of the command language, it does not have to be explicitly handled during parameter value selection.

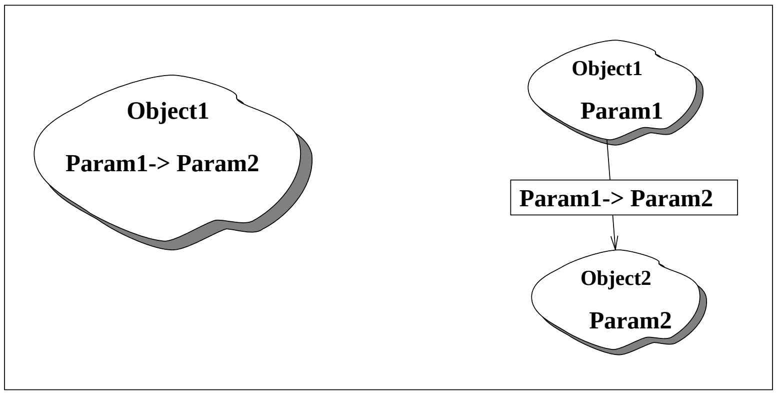


Figure 3.9: Object Splitting - What to do when constraints are local to one object

Peer Constraints address relationships between objects at the same level in the object hierarchy. It may be possible that objects at the same level constrain one another, but we have not witnessed this constraint in practice. To resolve a *peer constraint*, we recommend that objects at the same level be ordered from left to right by some priority assignment. While choosing parameters during test generation, we will impose a "left-to-right" resolution rule to make sure parameter attributes are resolved in the correct order. If objects have equal priority, then order them left-to-right arbitrarily.

The fifth relationship, *Explicit Concatenation*, defines how to handle constraints that span more than one level in the object hierarchy. During parameter value selection, all explicit constraints are concatenated from the root (top) of the hierarchy down to the object in question. Consider the path in the robot object hierarchy {**Robot** ... **Robot Tool** ... **Drill** ... **Coordinates**}. As we choose parameter attributes along the path, we must *concatenate* the constraints from all previous levels. For example, when choosing a value for the *drill-bit*, we must consider the compound constraint, (`robot-id AND tool-id` $\rightarrow$ `drill-bit`).

The final relationship *IntraObject Constraint*, is handled by the Domain Analysis. During Domain Analysis, one may find that the range of values for one object element constrains the values of another object element *within the same object*. This is called an *IntraObject Constraint.* We treat this constraint as anomalous because it identifies a relationship between parameters within a single object instead of showing the relationship on the arc between two objects. To remedy this anomaly, a new object is created with the new object subordinate to the other. The object elements that caused the split can be placed into separate objects and a detailed constraining relationship is placed on the arc connecting the objects. This keeps the object model uniform with regards to relationship constraints. Figure 3.9 shows the steps to split an object.

Rationale for an Object Hierarchy

The object hierarchy serves three primary purposes. First, the hierarchy stores semantic information, second, it provides a structure for choosing parameter values, and third, it makes test generation more efficient. Semantic information captured by the object relationships is needed for parameter value selection. In the last phase of test case generation, parameter values are selected for each command. Without this semantic information, we could rarely generate meaningful test cases.

The hierarchy also provides a structure for choosing parameter values. All parameters are selected by traversing a path from the top of the hierarchy down to the object in question. At each node (object) in the path, a parameter associated with that object can be selected. When traversing an arc to the next lower level, all of the inheritance rules on the arc are applied. Consider the object hierarchy for the robot manufacturing example. Suppose one needs a value for the *drill-bit* parameter. The path {**Robot** ... **Robot Tool** ... **Drill**} must be followed. Starting at the root of the tree, one chooses a *robot-id* parameter. Traversing the arc down to the **Robot Tool** object, the explicit constraint *robot-id* → *tool-id* is applied. From the constrained list of choices, the *tool-id* can be set to `DRILL`. Next, the explicit concatenation rule is applied (*robot-id* `AND` *tool-id* → *drill-bit*). Finally, the *drill-bit* parameter can be selected.

The last benefit of the object hierarchy is to make test generation more efficient. The object hierarchy represents the type-definitions for each object in the software product. If all *object instances* were stored explicitly, then we may not be able to efficiently generate test cases. First, for any reasonably sized system, the memory requirements to store all *instances* would be impractical. Second, at any given time, very few of the *object instances* are truly needed for test generation. Therefore, the object hierarchy is used as a structure to generate test cases. As parameter values are needed, *object instances* are maintained temporarily. When no longer needed, the *instances* can be deleted.

3.4 Command Definition

The third step in the Domain Analysis for Domain Based Testing is to formally define the syntax and the semantics of the *command language*. Some of the information has already been identified in the Command Glossary. Consider the `DRILL` command in Table 3.8. One field in the entry specifies the *syntax* of the command. In this example, we use Backus Naur Form (BNF). To generate a command from the BNF, we take a random-walk through the production. An alternative to BNF is to use an editor to create syntax diagrams for each command. Syntax diagrams are

Command Name	`DRILL`
Syntax	Drill-Cmd ::= `DRILL` <*robot-id*> `USING` <*drill-bit*> `AT` <*coordinates*>
Description	Instruct a robot to drill a hole using a specific drill bit at a particular position.

Table 3.8: Command Glossary Entry for the `DRILL` command

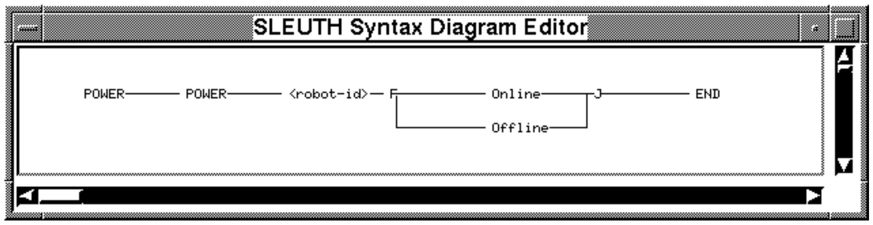


Figure 3.10: Syntax Diagram for the `POWER` Command

graph representations of the syntax of the language. In Figure 3.10 a window based editor is shown [1]. Using this editor, the syntax for each command could be defined, the test engineer would have a user-friendly tool to define the syntax, and a random-walk through the internal representation could be used to generate a test case.

Besides syntax, *preconditions* and *postconditions* for each command must be defined. *Preconditions* identify the conditions that must hold **before** the command can execute. *Postconditions* list the conditions that are true **after** the command executes. From our experience, pre/post conditions list values for *parameter state* and *nonparameter state* elements. Parameter States can be used to create proper command sequencing, and Nonparameter States can be used to selected valid parameter values. Consider the `SAW` command in Table 3.9. The preconditions for the command require the robot to be turned "on," and the robot and the saw to be available. If the robot is currently turned "off," we can issue the following sequence of commands to make sure the test case makes sense.

```
POWER r23 Online
DRILL r23 USING Drill-Bit2 AT (100,50,20)
```

[1] Sleuth is an automated test tool based on Domain Based Testing. The syntax diagram editor is one part of the tool [Wal93]

Command	`SAW`
Syntax	`SAW-CMD ::= SAW` <*robot-id*> `USING` <*saw-id*> `FROM` <*coord1*> `TO` <*coord2*>
Description	Instruct a robot make a saw cut between to points with a specific saw blade
Preconditions	
State	robot-status(<robot-id>) = Online
NP State	robot-state(<robot-id>) = AVAILABLE saw-state(<saw-id>) = AVAILABLE
Postconditions	
State	
NP State	robot-state(<robot-id>) = Busy saw-state(<saw-id>) = Busy
Intracommand	<coord1> ≠ <coord2>

Table 3.9: Command Glossary Entry for the `SAW` Command

Initial State	Transition	Final State
Online	None	Online
Online	POWER <robot-id> Offline	Offline
Offline	POWER <robot-id> Online	Online
Offline	None	Offline

Table 3.10: State Transition Table for the *robot-status* Precondition

One way to represent the transitions needed to put the system in the proper state is to use a state transition table. Entries in the table define the current state, the transition (command) needed to change the state, and the value of the resulting state. An example of the state transition table for the *robot-status* precondition is shown in Table 3.10.

Nonparameter State elements represent object state information that results from side-effects of executing a command or a sequence of commands. It may not be possible to change the value of a nonparameter state element on demand. Therefore, Nonparameter States are used to adjust the set of choices for parameter values. For instance, the `SAW` command requires the robot and the saw to be `AVAILABLE`. Using this information, we can reduce the number of choices for the *robot-id* parameter to those robots that are `AVAILABLE` during parameter value selection, and we can eliminate the *saw-id*'s that are not `AVAILABLE`.

Postconditions identify the state of the system after the command executes. This state information may be important for future command sequences or for parameter value selection. The postconditions for the `SAW` command show that the *robot-state* and the *saw-state* are `BUSY`. Associating this information with the proper object instances will make sure the remainder of the test generation for the script is meaningful.

The last step in Command Language Definition is to define Intracommand Rules. These rules identify constraints placed on parameter value selection within the command. They apply only while generating parameters for a single command. Intracommand rules handle special parameter

Intracommand Rule	Description
P1 *OP* P2	The relationship between P1 and P2 must hold where, $OP = \{=, \neq, <, >, \leq, \geq\}$
if P1 *OP* P2 then OMIT(P3)	If the relationship between P1 and P1 holds then Eliminate the value of P3 as a valid choice where, $OP = \{=, \neq, <, >, \leq, \geq\}$

Table 3.11: Intracommand Rules

generation constraints that cannot be encoded into the Object Hierarchy or pre/post conditions. From our experience, the object hierarchy and the pre/post conditions handle most of the parameter constraints. Because of this, few Intracommand Rules seem to exist. An example of an Intracommand Rule is shown in the Command Glossary for the `SAW` command (see Table 3.9). This rule states that *coord1* cannot equal *coord2*. Other Intracommand Rules have been analyzed, and they are summarized in Table 3.11.

3.5 Script Definition (Command Sequencing)

The fourth step in the Domain Analysis for Domain Based Testing is called Scripting. Scripts encode dynamic system behavior by capturing rules for sequencing commands. Sequencing information is necessary because arbitrarily ordering a list of commands rarely produces semantically correct test cases. In fact, results from an early prototype of Domain Based Testing suggests that without scripting less than 50% of the commands in the test case are meaningful [Cra93]. Besides capturing dynamic system behavior, scripting allows the test engineer to develop test cases at a high level of abstraction. They don't have to worry about parameter value selection, command syntax, or pre/post conditions. Instead, they can focus on high level test scenarios.

Scripts are visualized as state transition diagrams (see Figure 3.11). As the figure shows, the script traverses various states based on the value of the current state and the choices for the next transition(s). Arcs are labelled with the names of specific commands, script classes, or named scripts. By restricting the commands that can be executed on each arc, we can create proper command sequences.

While the state transition diagram is a good way to conceptualize scripting, scripts are easier to represent as stored command sequences. We call these archived scripts *Megascripts*. For example, "Robot.script" in Table 3.12 shows a megascript for the robot manufacturing system. The script turns three robots on, generates up to 100 `DRILL` commands, generates up to 50 commands from the "Any" Class, includes another script called "BugBuster," and it merges two scripts.

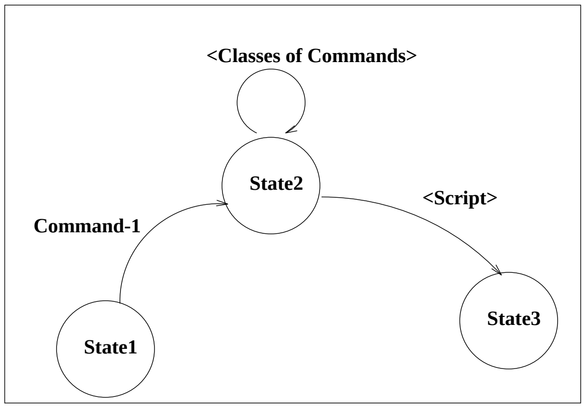


Figure 3.11: State Transition Diagram for a Generic Script

```
POWER r23 Online
POWER r24 Online
POWER r25 Online
@100/DRILL
@50/Any
@include BugBuster.script
@merge Saw.script Router.script
```

Table 3.12: Robot.script

3.5.1 Script Classes

Scripting classes create a set of “building blocks” from which more complex scripts can be built. Table 3.13 suggests ideas for script classes, where one column lists examples from the robot manufacturing system. The number of scripting classes and their content is problem dependent. Some software products can be tested with a small number of scripts while others may need an elaborate hierarchy of script classes. Script classes list a set of commands that perform a specific task. For example, the *Setup* class lists commands that perform system set up tasks. During test generation, test engineers should be able to configure script classes. For example, one may want to turn off the `ACCEPT` and `RELEASE` commands from the *Action* Class. By doing this, the set of Action commands becomes {`SAW, DRILL`}.

Megascript is the name given to an archived (stored) sequence of commands. Megascripts can be used to create complex test scenarios. For instance, a megascript can perform a system setup, present a workload to the system, and then test some fault. Megascripts can also be used to exercise particular objects, commands, or to stress test the system. Megascripts can be *included* into another script and two or more megascripts can be *merged*. Including a megascript into a new script is a good way to create test cases from smaller sequences of commands. It is also a good way

Class	Description	Robot Example
Setup	Commands that perform system Set Up	`POWER`
Mode	Commands that set *Parameter Modes*	`SET TOOL FAILURE`

Action	Commands that exercise the software system	`SAW`
		`DRILL`
		`ACCEPT`
		`RELEASE`

Any	Setup $\cup$ Mode $\cup$ Action	`SET TOOL FAILURE`
		`POWER`
		`SAW`
		`DRILL`
		`ACCEPT`
		`RELEASE`

Megascript	An archived script	Robot.script

Table 3.13: Script Classes

to reuse test cases. Merging takes two or more megascripts and "shuffles" them together. This may be useful when testing shared devices or when building more complicated test cases.

3.5.2 Script Rules

The last step in the Domain Analysis for Domain Based Testing is to define semantic rules associated with script generation. From our preliminary work, two types of scripting rules have been identified (1) Command Sequencing and (2) Script Parameter Selection. In some systems, commands must be issued in a particular order. Consider the robot manufacturing system. It does not make sense to `RELEASE` a job from a robot unless one has been `ACCEPT`ed. The sequence below shows one way to represent this rule.

```
ACCEPT <25/Any> RELEASE
```

The rule states that an `ACCEPT` command can be followed by up to 25 commands from the "Any" class and the sequence is terminated by a `RELEASE` command. The first and the last command in this rule form "bracketing" information that must be obeyed for meaningful test generation. For example, suppose `RELEASE` is selected as the next command in the megascript. The scripting rule above states that an `ACCEPT` command must occur earlier in the command sequence. The application of this rule will generate the `ACCEPT <25/Any> RELEASE` command sequence. For flexible test generation, the test engineer should be able to modify the rule. One should be able to change the upper limit on the number of commands to be generated between "brackets" and the tester should be able to change the class from which commands are chosen. Testers should also be able to turn this rule on and off to provide as much flexibility as possible for test case generation.

Notation	Description
p*	Choose any valid value for *p*
p	Choose a previously bound value for *p*
p-	Choose any except a previously bound value for *p*

Table 3.14: Script Rule: Parameter Value Selection

The second script rule defines how to choose parameters for script sequencing. Table 3.14 shows three script parameter selection rules. The first rule, *p**, states that parameter *p* can be selected from any valid choice according to object inheritance constraints. The second rule, *p*, restricts the value of parameter *p* to a previously bound value. The third rule, *p-*, denotes that parameter *p* can be selected from any valid choice except for the currently bound value of *p*. The `ACCEPT - RELEASE` sequencing rules can be annotated with script parameter selection rules. For instance,

```
ACCEPT robot-id* conveyor-id *
<25/Any>
RELEASE robot-id conveyor-id -
```

This rule states that the *robot-id* and *conveyor-id* parameters can be selected from any valid choice for the `ACCEPT` command. The `RELEASE` command must use the previously bound value for the *robot-id* and it can not use the same value for the *conveyor-id.* This rule should make sense. The robot that accepted a job should be the same one to release it, but you don't want to put the job on the same conveyor belt from which the job was received.

3.5.3 Summary

Domain Based Testing is a software test method based on Domain Analysis and Domain Modeling. In this chapter, we presented steps for a Domain Analysis for DBT. The analysis is based on three components (1) Object Model Definition, (2) Command Language Definition, and (3) Scripting Definition. We defined Domain Based Testing, what must be analyzed when building a Domain Based Testing system, and what functions are needed for automated test generation. All static and dynamic behaviors of the software product were considered. The steps were presented sequentially, but we recommend an iterative approach for the analysis. The results of the specification is a Domain Model from which test cases can be automatically generated.

Chapter 4

TEST GENERATION PROCESS MODEL

4.1 Introduction

The Test Generation Process Model defines the sequence of steps to create test cases. In the previous chapter, we concentrated on analyzing the software system under test, and we defined *what* needs to be represented for test generation. In this chapter, we discuss implementation issues needed to build an automated test generation tool. Figure 4.1 shows the components of the Test Generation Process. First, one defines a *System Specification* of the system under test. Next, test engineers create *System Configurations* for specific test scenarios. Along with the configuration, the tester may need to consider various *Test Criteria.* Finally, the *Test Generation* component creates the test case. In the next four sections, we describe the functions of each block in the figure.

4.2 System Specification

The *System Specification* sets up a new problem domain for automated test generation. This step is performed for every new software system and it may be needed after new releases of the software. If the new release changes the command language significantly, then the specification should be redefined. The *System Specification* must represent all of the information from the Domain Model. One can think of the *System Specification* as the default configuration for test data generation. Table 4.1 lists the operations available to the test engineer. For instance, testers must be able to define *scripting classes*, and utilities to enter command syntax, define pre/post conditions, and intracommand rules are needed. To complete the *System Specification*, we need editors to define default parameter values and parameter inheritance rules.

4.3 System Configuration

System Configuration allows test engineers to hand-craft scenarios for test generation. Test scenarios may concentrate on particular commands or command sequences. A set of test cases

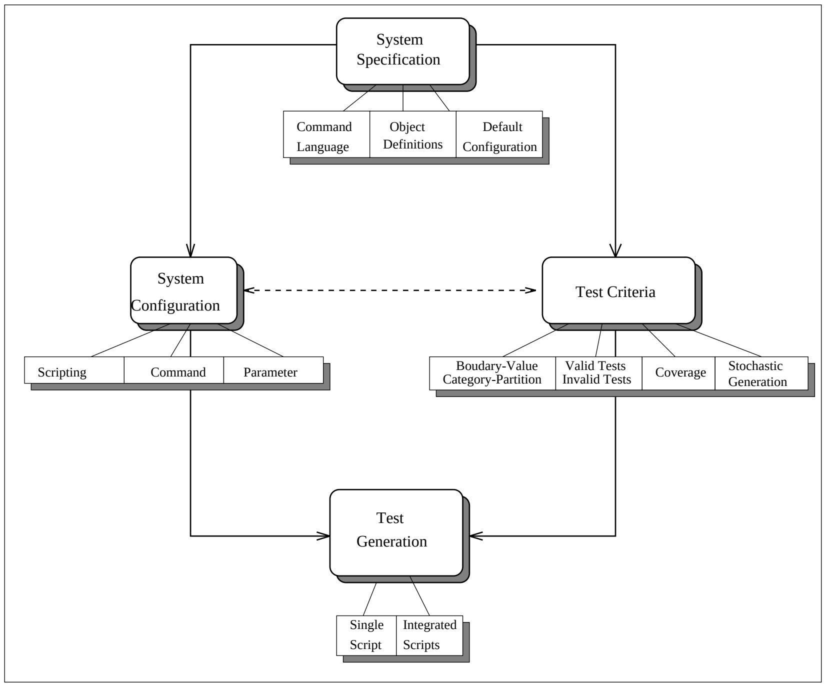


Figure 4.1: Test Generation Process Model

Component	Operation
System Specification	Define Script Classes Define Script Rules Define Command Syntax Define Pre/Post Conditions Define Intracommand Rules Define Default Parameter Values Define Parameter Inheritance Rules

Table 4.1: Operations for System Specification

Component	Operation
System Configuration	Modify Script Classes
	Modify Script Rules
	Define New Scripts
	Turn Script Rules `on/off`
	Modify Command Syntax
	Modify Intracommand Rules
	Turn Pre/Post Conditions `on/off`
	Turn Intracommand Rules `on/off`
	Modify Parameter Values
	Modify Parameter Inheritance Rules
	Turn Parameter Inheritance Rules `on/off`
	Reset Configuration
	Load Configuration
	Save Configuration

Table 4.2: Operations for System Configuration

may be needed to check a special configuration of the system. Sometimes we may need tests with some of the semantic rules turned off. To achieve these objectives, a "work space" is created for each tester. This prevents corruption of the original specification and it allows multiple test engineers to use the system simultaneously. Table 4.2 lists the operations supported by the *System Configuration* phase. Most of the functions allow the test engineer to modify the specification. The last three functions allow configurations to be restored from the original specification, saved for future use, and recalled at a later time.

4.4 Test Criteria

The *Test Criteria* component is used with the *System Configuration* to develop test case scenarios based on Black-Box Testing strategies. Black-Box testing identifies the conditions where a program does not behave according to its specification. Domain Based Testing should support strategies such as *boundary-value*, *valid test generation*, and *invalid test generation*. Most of the time, these strategies involve modifying the set of parameter values from which test cases can be generated. The *Test Criteria* component should provide utilities to adjust the frequencies of command generation and functions to calculate Domain Based Coverage Measures. Sometimes the test engineer knows that certain commands are generated more frequently than others, and sometimes the tester has operation profile data that provides a real-world test scenario. Therefore, a utility should be provided to set command generation frequencies. Another function is needed to measure Domain Based Coverage. We need to define and measure coverage with respect to the Domain Model. With this information we may be able to improve test case generation or we

Component	Operation
Test Criteria	Modify Parameter Values Modify Command Syntax Turn Script Rules `on/off` Turn Pre/Post Conditions `on/off` Turn Intracommand Rules `on/off` Modify Parameter Values Modify Parameter Inheritance Rules Turn Parameter Inheritance Rules `on/off` Measure Domain Based Coverage

Table 4.3: Operations for Test Criteria

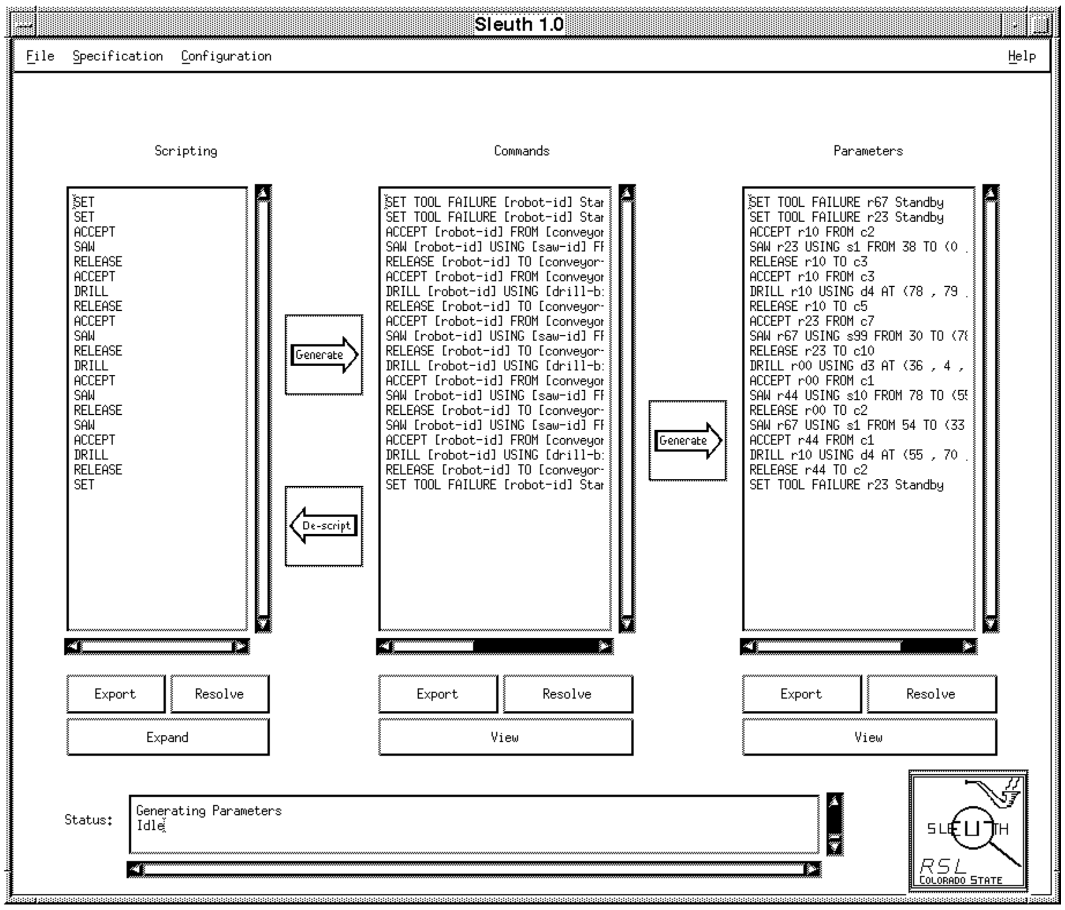
may be able to define better coverage measures. Table 4.3 summarizes the operations for the *Test Criteria* component.

4.5 Test Generation

The *Test Generation* component creates test cases using information from *Test Configuration* and *Test Criteria*. Test generation follows three steps (1) Script Expansion, (2) Command Template Generation, and (3) Parameter Value Selection. Script expansion allows a tester to type command sequences, it applies script sequencing rules, and it includes or merges saved scripts. The results from the scripting phase is a list of command names. The second phase takes each command name and it generates a template by "walking" through the command syntax. The last phase takes the list of command templates and it selects parameter values. Besides generating test cases, the *Test Generation* phase should provide functions to save the tests at all three stages. We called the archived tests *megascripts*, *megacommands*, and *megaparameters*, respectively. Once archived, the test engineer can recall the test case or include it in a new test. The operations for *Test Generation* are listed in Table 4.4. Domain Based Testing and the Testing Generation Process Model have been implemented using the X Windows System. Figure 4.2 shows the main window from the system [Wal93]. As shown in the figure, pull down menus and buttons control the test generation process. We will use this tool to assess the quality of Domain Based Testing.

Component	Operation
Test Generation	Generate Scripts Generate Command Templates Generate Parameter Values Save Scripts Save Command Templates Save Test Cases Include Scripts Include Command Templates Include Test Cases Merge Scripts

Table 4.4: Operations for Test Generation



```
SET
SET
ACCEPT
SAW
RELEASE
ACCEPT
DRILL
RELEASE
ACCEPT
SAW
RELEASE
DRILL
ACCEPT
SAW
RELEASE
SAW
ACCEPT
DRILL
RELEASE
SET
```

```
SET TOOL FAILURE [robot-id] Star
SET TOOL FAILURE [robot-id] Star
ACCEPT [robot-id] FROM [conveyor
SAW [robot-id] USING [saw-id] FF
RELEASE [robot-id] TO [conveyor-
ACCEPT [robot-id] FROM [conveyor
DRILL [robot-id] USING [drill-b:
RELEASE [robot-id] TO [conveyor-
ACCEPT [robot-id] FROM [conveyor
SAW [robot-id] USING [saw-id] FF
RELEASE [robot-id] TO [conveyor-
DRILL [robot-id] USING [drill-b:
ACCEPT [robot-id] FROM [conveyor
SAW [robot-id] USING [saw-id] FF
RELEASE [robot-id] TO [conveyor-
SAW [robot-id] USING [saw-id] FF
ACCEPT [robot-id] FROM [conveyor
DRILL [robot-id] USING [drill-b:
RELEASE [robot-id] TO [conveyor-
SET TOOL FAILURE [robot-id] Star
```

```
SET TOOL FAILURE r67 Standby
SET TOOL FAILURE r23 Standby
ACCEPT r10 FROM c2
SAW r23 USING s1 FROM 38 TO (0 .
RELEASE r10 TO c3
ACCEPT r10 FROM c3
DRILL r10 USING d4 AT (78 , 79 .
RELEASE r10 TO c5
ACCEPT r23 FROM c7
SAW r67 USING s99 FROM 30 TO (7
RELEASE r23 TO c10
DRILL r00 USING d3 AT (36 , 4 ,
ACCEPT r00 FROM c1
SAW r44 USING s10 FROM 78 TO (5
RELEASE r00 TO c2
SAW r67 USING s1 FROM 54 TO (33
ACCEPT r44 FROM c1
DRILL r10 USING d4 AT (55 , 70 .
RELEASE r44 TO c2
SET TOOL FAILURE r23 Standby
```

```
Generating Parameters
Idle
```

Figure 4.2: Window Based Test Generation Tool

Chapter 5

RESEARCH PLAN

5.1 Introduction

In this chapter, we present the research plan for Domain Based Testing. Preliminary work for DBT has been finished. We developed specification procedures for the Domain Analysis and a representation for the Domain Model. We started to consider design and implementation issues with the Test Generation Process Model. We also have experimental installation of a Domain Based Testing tool called *Sleuth*. Yet, we still have work remaining. First, we need to completely analyze design and implementation issues for DBT and the Testing Process Model. Second, we need to assess the quality of DBT through an experimental evaluation. Third, we need to develop Domain Based Regression Testing rules. Each of these is discussed in more detail in the remaining sections. We also provide a schedule for completing the research and a summary about the contributions of this research.

5.2 Domain Based Testing : Design Issues

We propose to evaluate design and implementation issues for Domain Based Testing and the Test Generation Process Model. To help with this evaluation, we will compare and contrast two DBT implementations. The first one is a *prototype* based on W-grammars where Scripting, Command Template Generation, and Parameter Value Selection are encoded into grammar productions [vMCH93]. The second implementation is an interactive tool called *Sleuth*. *Sleuth* separates syntax and semantic rules, and it separates the three phases of test generation. Using the *prototype* and *Sleuth*, we can show how design decisions influence one another. We can demonstrate shortcomings of particular designs, and we can point out good alternatives.

Table 5.1 summarizes the issues that will be included in the analysis. Design issues will be separated into two categories, High Level Design and Low Level Design. The first issues to resolve will be the interfaces between Scripting, Command Template Generation, and Parameter Value

Component	Design Issue
High Level Design Test Generation Process	 Script to Command Generation Interface Command Generation to Script Interface Command Generation to Parameter Value Selection Interface Script Rule Resolution Command Rule Resolution Parameter Rule Resolution
Low Level Design Scripting	 Script Representation Script Rule Representation Script Rule `on/off`
Low Level Design Command Generation	 Command Syntax Representation Command Pre/Post Condition Representation Intracommand Rule Representation Turning Command rules `on/off`
Low Level Design Parameter Value Selection	 Parameter Value Representation Parameter Inheritance Rule Representation Turning Rules `on/off`

Table 5.1: Design Issues to Investigate

Selection. We need a clear idea about what information is passed between each phase. With well-defined interfaces, each phase becomes a self-contained component. With separate components, each phase can be designed independently and implemented separately. Maintenance should be easier, too. As we refine our ideas about Domain Based Testing, we can replace one phase without disrupting the others. Furthermore, the three phased approach provides a good foundation for experiment. Suppose we want to try a new Script Generation procedure. We could install the experimental Script phase without changing the Command Template Generation, or Parameter Value Selection.

Low level design will examine each phase of the test generation process. In the Scripting phase, we must consider alternatives for script representation, script rule representation, and script rule modification. The Command Generation phase must evaluate command syntax, pre/post condition, and intracommand rule representations. In the last phase, we address parameter value and parameter inheritance representation.

Besides representation issues, we must also think about where to resolve the semantic rules. Table 5.2 shows phases where each type of rule could be resolved. The `X`'s in the table show all possible phases where a rule type could be resolved. `P`'s denote where the *prototype* resolves the rule, and the `S`'s show where *Sleuth* resolves the rule. One approach would be to resolve each semantic

Semantic Rule	Scripting Phase	Command Phase	Parameter Phase
Script			
Command Sequencing	X S	X	X P
Parameter Selection	X	X	X S P
Command			
Template Generation		X S	X
Precondition	X S	X	X P
Postcondition		X	X
Intracommand		X	X S P
Parameter			
Parameter Inheritance			X S P

X = Possible Alternative
P = Prototype
S = Sleuth

Table 5.2: Design Alternatives : Semantic Rule Resolution

rule in its own test generation phase. For example, Command Sequencing and Script Parameter Selection could be resolved in the Scripting phase. In practice, there are implementation trade-offs that show that this may not be a good approach. Therefore, we need to address rule resolution in more detail and we need to show both the benefits and shortcomings of each alternative.

5.3 Domain Based Testing : Experimental Evaluations

In the second part of the research, we propose to evaluate the quality of Domain Based Testing and the Testing Process Model. Domain Based Testing seems sound, but we plan three evaluations to support our ideas (1) Experimental Evaluation, (2) Test Case Reuse, and (3) Domain Based Coverage Measures. First, empirical results from commercial software testing would provide feedback about DBT and its usefulness. We currently have a Domain Model for a robot tape library system [1]. We can generate test cases for the tape library using the *prototype* and *Sleuth*. Our experimental evaluation can proceed in two directions. First, we can evaluate the **StorageTek** problem domain in detail. This would give use detailed information from one problem domain. Second, we can apply DBT to other problem domains. This would give us feedback about other command-based systems. In either case, we will summarize the DBT features that make the testing easier, which features testers don't like, and which features aren't needed.

For the second evaluation, we propose to assess test case reuse. Because Domain Based Testing is based on ideas from the software reuse community, we should be able to reuse test cases in each phase of test case generation. Figure 5.1 shows the combinatorial potential of the three phase approach to test case reuse. From a single script, we can generate several command template sequences. Each command template would be specific to a software release, domain configuration,

[1]The tape library is a product of the **StorageTek** Corporation.

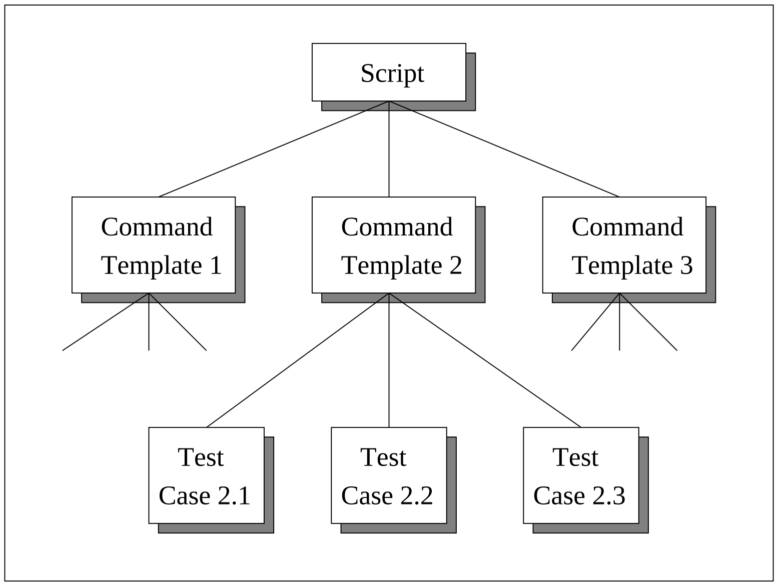


Figure 5.1: Test Case Reuse Concept

Domain Based Coverage Measure
Object Coverage
Path Coverage (w.r.t. the Object Hierarchy)
Command Coverage
Command Rule Coverage
Script Coverage
Script Rule Coverage

Table 5.3: Domain Based Coverage Measures

or semantic rule set up. Each command template can generate test cases using different system configurations, parameter values, or parameter inheritance rules. Through test case reuse, test engineers become more productive, they can re-run tests, and they can recall and modify test cases easily. Even though Domain Based Testing creates a good structure to reuse test cases, we need empirical results to support our claims.

Finally, we propose to evaluate Domain Based Coverage Measures (see Table 5.3). All of the coverage measures are defined with respect to the Domain Model. For example, *command coverage* measures the number of commands used in a particular test case. Many coverage measure come to mind, and some of them may be a better at identifying errors than others. Intuitively, we think that many of the coverage measure will be *problem dependent.* Nevertheless, we will calculate Domain Based Coverage, and we will correlate the results with error detection. From the correlation, we should be able to identify worthwhile coverage measures, or we may be able to define better Domain Based Coverage Measures.

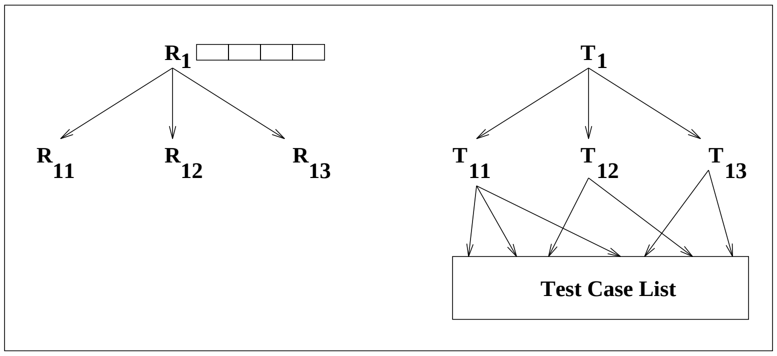


Figure 5.2: Requirements and Test Suite Hierarchies [vMO93]

5.4 Domain Based Regression Testing

The third part of our proposed research is called Domain Based Regression Testing. Regression Testing is a testing strategy associated with evolutionary software development. New features may be added, old options may be deleted, and bugs may be fixed. As the software evolves, the tester must make sure the old features still work and the "fixes" don't cause new problems. Most of the time, it is economically infeasible to re-run all of the test cases from the original system. Therefore, one must choose a subset of test cases that have a high potential to detect errors. Leveraging an idea from Von Mayrhauser and Olender, we can use the structure of the Domain Model to identify rules for regression test suites [vMO93]. In their paper, requirements are represented hierarchically and test suites are associated with each requirement. Figure 5.2 shows a example of this structure. We propose to substitute the *object hierarchy* from our Domain Model for the requirements hierarchy. Figure 5.3 shows the concept using the robot manufacturing example. The Domain Model provides a unique structure from which regression tests can be selected. Table 5.4 lists the regression testing rules that will be included in the research. The columns `Add, Modify,` and `Delete` refer to the operation of adding a new feature, modifying an existing feature, or deleting a feature. Consider the table entry for *command syntax*. We will investigate test suite selection rules for *Adding a New Command*, *Modifying Command Syntax*, and *Deleting a Command*. Along with these rules, we will also examine both aggressive and conservative test suite selection. Suppose an object is modified. One regression test suite may focus on tests that only include that object. Alternatively, we could define a regression test suite with test cases that include the modified object **and** objects with a relationship to the modified object.

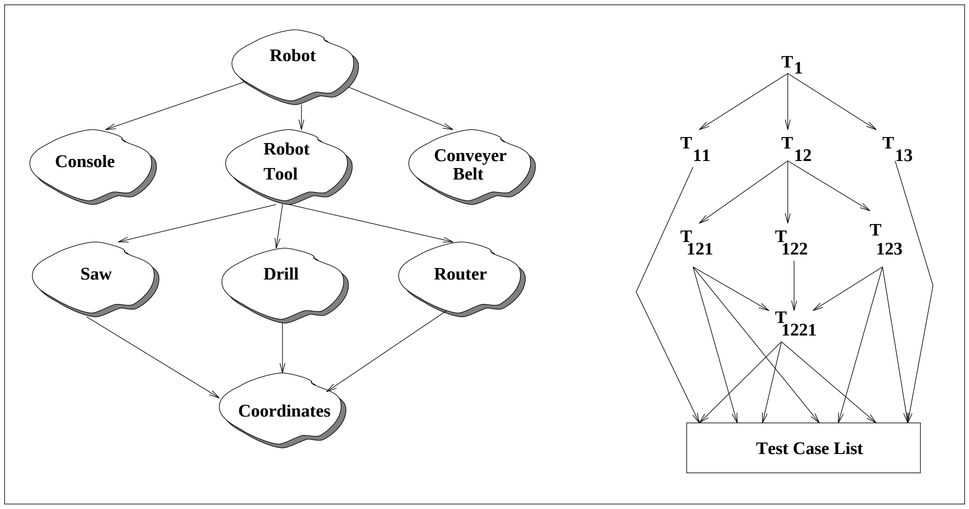


Figure 5.3: DBT Regression and Test Suite Hierarchies

	Add	Modify	Delete
Script			
Megascript	X	X	X
Script Rule	X	X	X
Script Class	X	X	X
Command			
Syntax	X	X	X
Pre/Post Condition	X	X	X
Intracommand Rule	X	X	X
Megacommand	X	X	X
Parameter			
Object	X	X	X
Object Element	X	X	X
Object Relationship	X	X	X
Megaparameter	X	X	X

Table 5.4: Domain Based Regression Testing Rule Matrix

5.5 Research Schedule

Table 5.5 shows the schedule for completing this research. Our goal is to address the remaining issues within the next year. We also have ideas for at least four research papers. The first will describe Domain Based Testing, DBT Domain Analysis, and the Domain Model. The second will address our Test Generation Process Model. We are considering a paper that addresses test case reuse, and finally, we will publish our results from Domain Based Regression Testing.

Start Date	End Date	Research Topic
Jan 94	May 94	Domain Based Regression Testing • Analysis • Design • Implementation
April 94	June 94	Design Issues Evaluate New Domains Evaluate **StorageTek** Domain
June 94	Aug 94	Domain Based Coverage Analyze Test Case Reuse
Sept 94	Nov 94	Write Dissertation
Dec 94		Defend Dissertation

Table 5.5: Domain Based Testing Research Schedule

5.6 Contributions of this Work

Domain Based Testing is a general approach to automated test generation for command-based systems. It is the first technique to use a Domain Analysis and a Domain Model for software testing. Domain Analysis shows how to specify a problem for DBT, and the Domain Model provides a structure from which test cases can be generated. For efficiency, DBT separates syntax and semantic issues, and it divides test generation into three phases. Handling semantic information has been a recurring problem for automatic test generators. By spreading the semantic rules across three phases, DBT is able to address the complexity of the command language semantics. The Domain Model for Domain Based Testing is not only useful for test generation it is also good for test case reuse and regression test suite selection. Reusing test cases at three levels of abstraction improves tester productivity. It also provides a mechanism where combinatorial potential of test cases may be beneficial. Domain Based Testing also provides a structure for generating regression test suites. Choosing a good test suite will reduce the number of test cases to execute, and automating some of the test suite selection will increase the productivity of the test engineer.

REFERENCES

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *Computing Surveys*, 14(2):159–192, June 1982.

[AI90] Kent C. Archie and Robert E. McLear III. Environments for Testing Software Systems. *AT&T Technical Journal*, pages 65–75, March/April 1990.

[And86] Stephen J. Andriole, editor. *Software Validation, Verification, Testing and Documentation*. Petrocelli, 1986.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bas90] Victor R. Basili. Viewing Maintenance as Reuse-Oriented Software Development. *IEEE Software*, pages 19–25, January 1990.

[BE85] John G. Brautigam and Marianne E. Erdos. Description and Application of a Software Testing Methodology. In *Proceedings of the Conference on Software Tools*, pages 184–189, 1985.

[Bei90] Boris Beizer. *Software Testing Techniques*. VanNostrand Reinhold, second edition, 1990.

[BF79] Jonathan A. Bauer and Alan B. Finger. Test Plan Generation Using Formal Grammars. In *Proceedings of the Fourth International Conference on Software Engineering*, pages 425–432, 1979.

[BF82] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 2. HeurisTech Press, 1982.

[Big92] Ted J. Biggerstaff. *Advances in Computers*, chapter An Assessment and Analysis of Software Reuse. Academic Press, 1992.

[Boo83] Grady Booch. *Software Engineering with Ada*. Benjamin/Cummings, 1983.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[BP89] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability - Concepts and Models*, volume I of *Frontier Series*. ACM Press, 1989.

[BS82] Franco Bazzichi and Ippolito Spadafora. An Automatic Generator for Compiler Testing. *IEEE Transactions on Software Engineering*, 8(4):343–353, 1982.

[CB92] Chye-Lin Chee and Jit Biswas. Experience with Integrating Operating Systems. In Peter A. Ng C.V. Ramamoorthy Laurence C. Seifert and Raymond T. Yeh, editors, *Proceedings of the Second International Conference on Systems Integration*, page 593, 1992.

[CDK+89] B.J. Choi, R.A. DeMillo, E.W. Krauser, R.J. Martin, A.P. Mathur, A.J. Offutt, H.Pan, and E.H. Spafford. The Mothra Tool Set. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume II, pages 275–284, 1989.

[CF82] Paul R. Cohen and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 3. HeurisTech Press, 1982.

[CFR90] James Collofello, Terry Fisher, and Mary Rees. A Testing Methodology Framework. In George J. Knafl, editor, *Proceedings of the IEEE 14th Annual International Software and Applications Conference*, pages 577–586, 1990.

[Che92] Jingwen Cheng. Parameterization Specifications for Software Reuse. *ACM SIGSOFT Sofware Engineering Notes*, 17(4):53–59, October 1992.

[Cho77] Tsum S. Chow. Testing Software Design Modeled by Finite State Machines. In *Proceedings of the First COMPSAC*, pages 58–64, 1977.

[CKO92] Bill Curtis, Marc I. Kellner, and Jim Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[Con89] Jeff Conklin. Design Rationale and Maintainability. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume 2, pages 533–539, 1989.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[Cra93] Stewart Crawford. Prototype for Storage Tek Automated Test Generator. Internal Test Results Report, 1993.

[CRV+80] A. Celentano, S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, G. Gramata, and F. Savoretti. Compiler Testing using a Sentence Generator. *Software-Practice and Experience*, 10:897–918, 1980.

[CU84] J. Craig Cleaveland and Robert C. Uzgalis. *Grammars for Programming Languages.* Programming Language Series. Elsevier, 1984.

[CvM93a] Stewart Crawford and Anneliese von Mayrhauser. A Program for Executable Command Generation from BNF Grammars. Technical report, Colorado State University, Computer Science Department, February 1993.

[CvM93b] Ronald T. Crocker and Anneliese von Mayrhauser. Maintenance Support Needs for Object-Oriented Software. 1993.

[DBCI91] William H. Deason, David B. Brown, Kai-Hsiung Chang, and James H. Cross II. A Rule-Based Software Test Data Generator. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):108–117, March 1991.

[DH81] A.G. Duncan and J.S. Hutchison. Using Attributed Grammars to Test Designs and Implementations. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 170–177, 1981.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[FF89] Anthony Finkelstein and Hugo Fuks. A Cooperative Framework for Software Engineering. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume 2, pages 189–199, 1989.

[Fis77] Kurt F. Fisher. A Test Case Selection Method for the Validation of Sofware Maintenance Modifications. In *Proceedings of the International Computer Software and Application Conference*, pages 421–426, 1977.

[FL88] Fisher and LeBlanc. *Crafting a Compiler*, chapter 14, pages 507–531. Benjamin-Cummings, 1988.

[FS86] Craig D. Fuget and Barbara J. Scott. Tools for Automating Software Test Package Execution. *Hewlett-Packard Journal*, pages 24–28, March 1986.

[FvBK$^+$91] Susumu Fujiwara, Gregor v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[GH88] David Gelperin and Bill Hetzel. The Growth of Software Testing. *Communications of the ACM*, 31(6):687–695, June 1988.

[Gom91] Hassan Gomaa. An Object-Oriented Domain Analysis and Modeling Method for Software Reuse. In V. Milutinovic B.D. Shriver J.F. Nunamaker Jr. and R.H. Spague Jr., editors, *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 2, pages 46–56, 1991.

[Gru82] Dick Grune. *On the Design of ALEPH*, chapter 1, pages 3–41. Mathematisch Centrum, 1982.

[Ham88] Richard Hamlet. Special Section on Software Testing. *Communications of the ACM*, 31(6):662–667, June 1988.

[HC91] James W. Hooper and Rowena O. Chester. *Software Reuse - Guidelines and Methods.* Plenum Press, 1991.

[Het84] William Hetzel. *The Complete Guide to Software Testing.* QED Information Sciences, 1984.

[How85] William E. Howden. The Theory and Practice of Functional Testing. *IEEE Software*, pages 6–17, September 1985.

[How86] William E. Howden. A Functional Approach to Program Testing and Analysis. *IEEE Transactions on Software Engineering*, 12(10):997–1005, October 1986.

[How87] William E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, 1987.

[How89] William E. Howden. Validating Programs without Specifications. In Richard A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 2–8, 1989.

[HT90] Dick Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[HTD90] James Hendler, Austin Tate, and Mark Drummond. AI Planning: Systems and Techniques. *AAAI*, pages 61–77, Summer 1990.

[Hul84] Christer Hulter. Simple Dynamic Assertions for Interactive Program Validation. In *Proceedings of the National Computer Conference (AFIPS84)*, pages 405–421, 1984.

[HVY84] J.C. Huang, Peter Valdes, and Raymond T. Yeh. A tool-based approach for software testing and validation. In *Proceedings of the National Computer Conference AFIPS*, pages 411–421, 1984.

[ICCHM] James H. Cross II, Elliot J. Chikofsky, and Jr. Charles H. May. *Advances in Computers*, volume 35, chapter Reverse Engineering.

[JCR90] T.N. Coomer Jr., J.R. Comer, and D.J. Rodjak. Developing Reusable Software For Military Systems - Why It Isn't Working. *ACM SIGSOFT - Software Engineering Notes*, 15(3):33–38, July 1990.

[Jeo92] Taewoong Jeon. *A Knowledge-Based System for Regression Testing.* PhD thesis, Illinois Institute of Technology, May 1992.

[JR92] Paul Johnson and Ceri Rees. Reusability Through Fine-gain Inheritance. *Software-Practice and Experience*, 22(12):1049–1068, 1992.

[Kor90] Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[Kru92] Charles Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[LG89] J.R. Lyle and K.B. Gallagher. A Program Decomposition Scheme with Applications to Software Modification and Testing. In *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume 2, pages 479–485, 1989.

[LvK92] Haikuan Li and Jan van Katwijk. Issues Concerning Software Reuse-in-the-Large. In Peter A. Ng C.V. Ramamoorthy Laurence C. Seifert and Raymond T. Yeh, editors, *Proceedings of the Second International Conference on Systems Integration*, pages 66–75, 1992.

[LW91] Hareton K.N. Leung and Lee White. A Cost Model to Compare Regression Test Strategies. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 201–208, 1991.

[Man85] Robert Mandl. Orthogonal Latin Squares : An Application of Experiment Design to Compiler Testing. *Communications of the ACM*, 28(10):1054–1058, October 1985.

[Mar92] David M. Marks. *Testing Very Big Systems.* McGraw-Hill, 1992.

[Mul89] Mark Mullin. *Object Oriented Program Design with Examples in C++.* Addison-Wesley, 1989.

[Mun88] Carlos Urias Munoz. An Approach to Software Product Testing. *IEEE Transactions on Software Engineering*, 14(11):1589–1596, November 1988.

[Mye76] Glenford J. Myers. *Software Reliability : Principles and Practices.* John Wiley and Sons, 1976.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, 1979.

[New88] D.J. Newman. A Test Harness for Maintaining Unfamiliar Software. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 409–416, 1988.

[Nta84a] Simeon C. Ntafos. An Evaluation of Required Element Testing Strategies. In *Proceedings of the 7th Conference on Software Engineering*, pages 250–256, 1984.

[Nta84b] Simeon C. Ntafos. On Required Element Testing. *IEEE Transactions on Software Engineering*, 10(6):795–803, November 1984.

[Nta88] Simeon C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6):868, June 1988.

[OB88] Thomas J. Ostrand and Marc J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[OC92] Leon Osterweil and Lori A. Clarke. A Proposed Testing and Research Initiative. *IEEE Software*, pages 89–96, September 1992.

[OM91] A.A. Omar and F.A. Mohammed. A Survey of Software Functional Testing Methods. *ACM SIGSOFT Sofware Engineering Notes*, pages 75–82, April 1991.

[OU88] Martyn A. Ould and Charles Unwin. *Testing in Software Development*. Cambridge University Press, 1988.

[Pay78] A.J. Payne. A Formalized Technique for Expressing Compiler Exercises. *ACM SIGPLAN Notices*, 13:59–69, 1978.

[PD93] Rubén Prieto-Díaz. Status Report: Software Reusability. *IEEE Software*, pages 61–66, May 1993.

[Per86] William E. Perry. *How to Test Software Packages*. John Wiley and Sons, 1986.

[Pet85] Nathan H. Petschenik. Practical Priorities in System Testing. *IEEE Software*, pages 18–23, September 1985.

[Pur72] Paul Purdom. A Sentence Generator for Testing Parsers. *BIT*, 12(3):366–375, 1972.

[RAO89] Debra J. Richardson, Stephanie Leif Aha, and Leon J. Osterweil. Integrating Testing Techniques Through Process Programming. In *Proceedings of the ACM SIGSOFT89 - Symposium on Software Testing, Analysis, and Verification*, December 1989.

[RC81] Debra J. Richardson and Lori A. Clarke. A Partition Analysis Method to Increase Program Reliability. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 244–245, 1981.

[RG92] Kenneth S. Rubin and Adele Goldberg. Object Behavior Analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[RK91] Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, second edition, 1991.

[Ros85] A. Frederick Rosene. A Software Development Environment Called STEP. *Proceedings of Conference on Software Tools*, pages 154–160, April 1985.

[Roy93] Thomas C. Royer. *Software Testing Management - Life on the Critical Path*. Prentice Hall, 1993.

[Sne93] Harry M. Sneed. Regression Testing in Reengineering Projects. In *International Conference on Program Cognition*, June 1993.

[SSSW86] Ben Shneiderman, Philip Shafer, Roland Simon, and Linda Weldon. Display Strategies for Program Browsing: Concepts and Experiment. *IEEE Software*, pages 7–15, May 1986.

[TCY93] Will Tracz, Lou Coglianese, and Patrick Young. A Domain-Specific Software Architecture Engineering Process Outline. *ACM SIGSOFT Sofware Engineering Notes*, 18(2):40–49, April 1993.

[Tek92] Storage Tek. *StorageTek 4400 Operator's Guide. Host Software Component (VM) Rel 1.2.0.* StorageTek, 1992.

[Ter86] Patrick D. Terry. *Programming Language Translation*. Addison-Wesley, 1986.

[Tra92] Will Tracz. Domain Analysis Working Group - First International Workshop on Software Reusability. *ACM SIGSOFT Sofware Engineering Notes*, 17(3):27–34, July 1992.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing*. McGraw-Hill, 1985.

[vM90] Anneliese von Mayrhauser. *Software Engineering - Methods and Management*. Academic Press, 1990.

[vM93a] Anneliese von Mayrhauser. Computer science 514 - software engineering topics notes, August 1993.

[vM93b] Anneliese von Mayrhauser. Maintenance and Evolution of Software Products. Technical report, Colorado State University, Computer Science Department, 1993.

[vM93c] Anneliese von Mayrhauser. Telephone conversation with j.s. hutchison. Telecon, September 1993.

[vMC93a] Anneliese von Mayrhauser and Stewart Crawford. Automated Test Support based on Domain-Specific Information. Technical report, Colorado State University, Computer Science Department, March 1993.

[vMC93b] Anneliese von Mayrhauser and Stewart Crawford. Formal Language Evaluation for 4400 ACS Operator's Commands. Technical report, Colorado State University, Computer Science Department, March 1993.

[vMC93c] Anneliese von Mayrhauser and Stewart Crawford. Preliminary Domain Information Classification. Technical report, Colorado State University, Computer Science Department, July 1993.

[vMCH93] Anneliese von Mayrhauser and Steward Crawford-Hines. Automated Testing Support for a Robot Tape Library. *Proceedings of the Fourth International Software Reliability Engineering Conference*, pages 6–14, November 1993.

[vMJ93] Anneliese von Mayrhauser and Taewoong Jeon. CASE Tool Architecture for Knowledge-Based Regression Testing. In *Tri-Ada93*, 1993.

[vMO93] Anneliese von Mayrhauser and Kurt Olender. Efficient Testing of Software Modifications. In *International Conference on Testing*, 1993.

[Wal93] Jeff Walls. Sleuth : An automated test generation tool, Nov 1993.

[WF89] Dolores R. Wallace and Roger U. Fujii. Software Verification and Validation: An Overview. *IEEE Software*, pages 10–17, May 1989.

[WG84] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, 1984.

[Wil88] David E. Wilkins. *Practical Planning : Extending the Classical AI Planning Paradigm*. Morgan-Kaufmann, 1988.

[WJ91] Elaine J. Weyuker and Bingchiang Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[WL92] Lee J. White and Hareton K.N. Leung. A Firewall Concept for both Control-Flow and Data-flow in Regression Integration Testing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 262–271, 1992.