Department of Computer Science

Domain Based Testing : Increasing Test Case Reuse

Anneliese von Mayrhauser, Richard Mraz,
Jeff Walls, and Pete Ocken

Technical Report CS-93-131

December 10, 1993

Colorado State University

Domain Based Testing : Increasing Test Case Reuse

Anneliese von Mayrhauser, Richard Mraz, Jeff Walls, and Pete Ocken

Abstract

Domain Based Testing (DBT) uses Domain Analysis and a Domain Model to automate test generation for command-based systems. Instead of using a Domain Model for reuse, we use it as a structure to generate test cases. Part of the Domain Model for DBT includes the syntax and semantics of the command language. Domain Based Testing separates command language syntax from command language semantics. The test generation process is divided into three phases, (1) Scripting, (2) Command Template Generation, and (3) Parameter Value Selection. DBT handles the complexity of the semantic rules by distributing them across all three phases. Because it is based on ideas from software reuse, DBT also provides a good structure for *test case reuse*. Our DBT tool, *Sleuth*, archives test cases at each phase of test generation. Production use of *Sleuth* to generate system tests for an automated robot tape library confirms that Domain Based Testing provides a wide variety of test case reuse scenarios.

Key Words: Domain Analysis, Domain Models, Automated Test Generation.

[1]Address correspondence to A. von Mayrhauser Department of Computer Science, Colorado State University, Fort Collins, CO 80523. (303)491-7016. Fax: (303) 491-6639, Email: avm@cs.colostate.edu

Research partially supported by the Colorado Advanced Software Institute (CASI), **StorageTek**, and the Air Force Institute of Technology. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

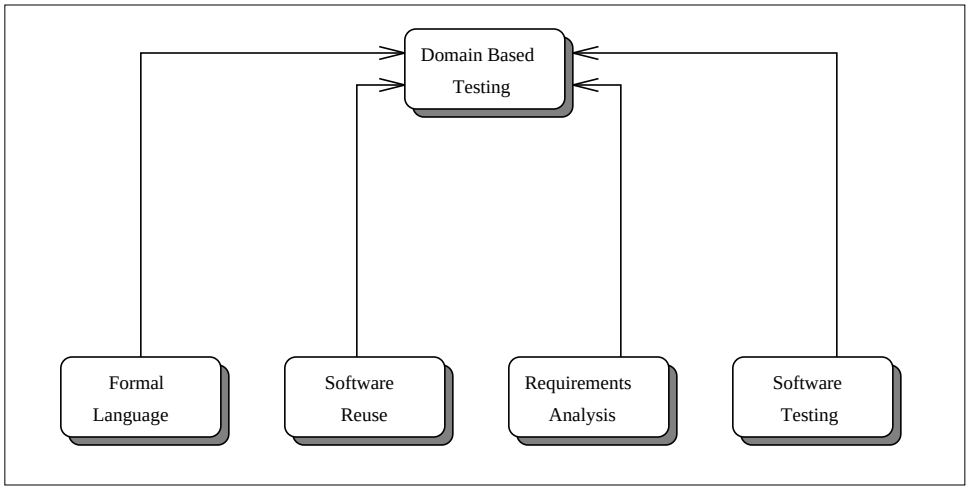

Figure 1: Foundations for Domain Based Testing

1 Introduction

Domain Analysis supports development and use of reusable components [4] [8]. Domain Analysis provides an organized model of a family of applications. Therefore, such a model, if defined properly should also be useful to develop a structured set of test cases for testing software in such domains. We call testing that is based on a domain analysis geared toward test generation *Domain Based Testing* (DBT) [2]. Its Domain Analysis and Domain Model facilitate reuse of test cases at several levels of abstraction. DBT has tool support (*Sleuth* version 1.1 is in production use by a system testing group at **StorageTek**), and experience has been very positive. Domain Based Testing combines domain analysis and automated test suite generation using a variety of sentence generation mechanisms to generate large sets of test cases that are syntactically and semantically correct.

We do not, however, rely on traditional use of grammars to generate test cases [5] [6] [7]. The main reason is that for practical problems with significant amounts of semantic information, the underlying grammars [3] quickly become unmanageable and difficult to maintain [4]. Further, possible reuse and incremental change or temporary disabling of selected semantic rules is more difficult with techniques that combine all aspects of test generation in a single mechanism. Instead, we assumed a three step process to test data generation:

(1) **Scripting**: Dynamic behavior as sequences of actions on logical or physical *objects* of a system.

(2) **Individual Manipulation of Object(s)**: In Command Languages these would be single command templates.

(3) **Objective Attribute Value Selection**: In Command Languages this refers to parameter value selection.

[2] This should not be confused with *Domain Testing* as described in [3].

[3] VanWijngaarden grammars, Attribute grammars [1].

[4] This has been confirmed in private conversation with J. Hutchison and J. Schlauer, as well as our own experience.

The advantages of this approach are:

1. Separation of concerns simplifies test suite generation at each step
2. Reusability becomes possible at all three levels of refinement. We will explore this further in Section 3

At this time, feasibility and practical usefulness of this approach have been demonstrated for command-based software systems that possess the following properties of the command language:

- Parameters in particular must have a high semantic content.

  This ensures that the Test Domain Model is rich enough to enable semantically meaningful test suite generation. Thus, a command generator for a compiler is inappropriate at the compiler invocation level (e.g. `cc example.c`), since all relevant knowledge is in the *content* of the file `example.c` rather than at the compiler invocation level (i.e. the command `cc` itself). This serves to identify the proper level of analysis rather than constrain the applicability of Domain Based Testing.

- Command Language parameters and command types must map to physical or logical objects of the system under test.

  This ensures that we can actually build a domain model that relates back to the commands of the system under test. In our analysis, all command languages exhibited this property well enough to be worthwhile candidates for Domain Based Testing.

Domain Analysis for testing is described in detail in Section 2. This section also shows how to structure the resulting objects, relationships and rules about dynamic behavior to facilitate test case reuse. Section 3 describes the automated test generation engine *Sleuth*. *Sleuth* supports Domain Based Testing. Its domain configuration phase allows descriptions of domain analysis results. Test engineers "customize" *Sleuth* for a particular testing domain. *Sleuth* then generate suites of test cases using the three step approach described above to facilitate test case reuse. Section 4 reports experiences with Domain Based Testing of an automated robot tape library system, particularly its usefulness in reuse of test cases at various levels of abstraction. Such reuse improves regression testing, incremental testing, and testing a variety of system configurations. Section 5 provides our conclusions.

2 Domain Analysis for Testing

Early reuse was confined to shared libraries, reusable programs, and reusable software components. Recently, software reuse concepts have been applied throughout the software life cycle. In general, reuse looks beyond single projects or systems where information common to a set of similar systems is exploited. 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

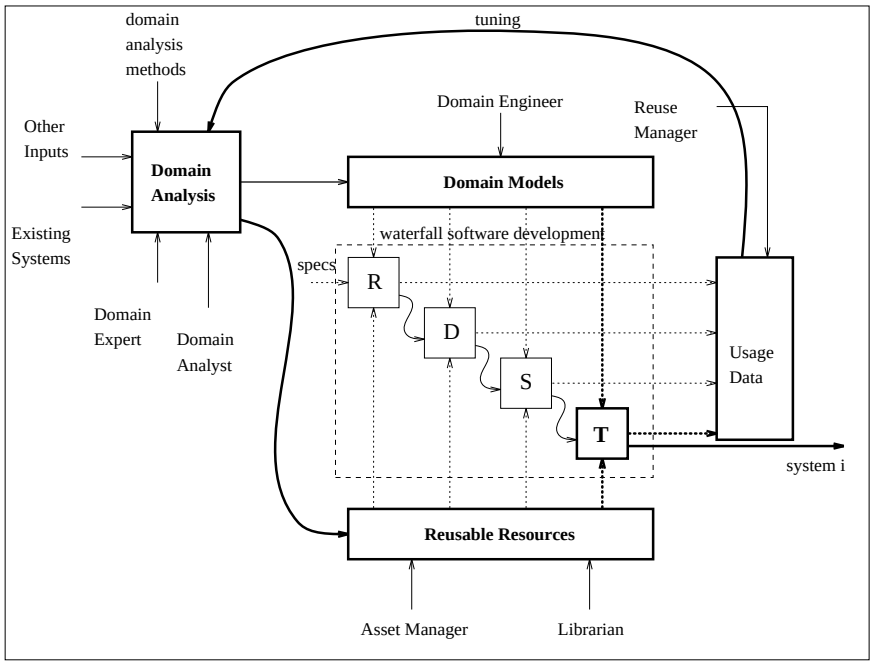

Figure 2: A Reuse Infrastructure [9]

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" [9]. The result of a Domain Analysis is called a *Domain Model*. Domain Models represent the reuse problem domain and serve as a mechanism to create instances of reusable components when building new software for the same domain.

Software reuse can be exploited at every phase of the software life cycle. Prieto-Diaz suggests a "reuse infrastructure" for waterfall software development (see Figure 2) [9]. Domain Models are used at each phase of software development. Feedback loops are provided to update the Domain Models. We explored the possibilities of a Domain Model for testing and automated test generation. In Figure 2, those boxes highlighted with **BOLD** lines denote the focus. From this point of view, Domain Based Testing can be used as a test generation tool, a "reusable" test case generator, or to identify regression test suites. For the remainder of the paper, we will show how to use Domain Based Testing as a "reusable" test case generator.

2.1 Objects, Relationships, and Semantic Rules

Domain Based Testing is based on a specialized Domain Analysis. The steps for the Domain Analysis are listed in Table 1. For command-based systems, we can easily identify parameters of the command language. We can also identify whether they are part of or describe a property of a physical or logical object. This analysis gives us a first cut of the *objects* and their particular *object elements*. Objects denote physical or logical entities in the problem domain. *Object elements* are similar to the concept of "object attributes" in Object-Oriented Analysis/Object-Oriented Design

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 1: Domain Analysis Steps for Domain Based Testing

(OOA/OOD). An attribute in OOA/OOD defines qualities and properties of the object [10]. Rarely do OOA/OOD methods refine the concept of an object's attributes. For Domain Based Testing, we found that classifying attributes more precisely simplifies test generation. Table 3 shows our detailed classification for object elements.

Object elements when derived from the command language can be attributes, modes, or states. *Attributes* describe parts of an object like name, physical or logical part, etc. *Modes* relate to modes of use, display, etc. For example, a command parameter may specify whether to run a software system in a novice mode or expert mode. In this case, the object element is a parameter of type *mode* and associated with the highest level (i.e. system) object. Similarly, it may be possible to influence the state of objects in the system via parameters (e.g. `ON/OFF`). Figure 3 also shows

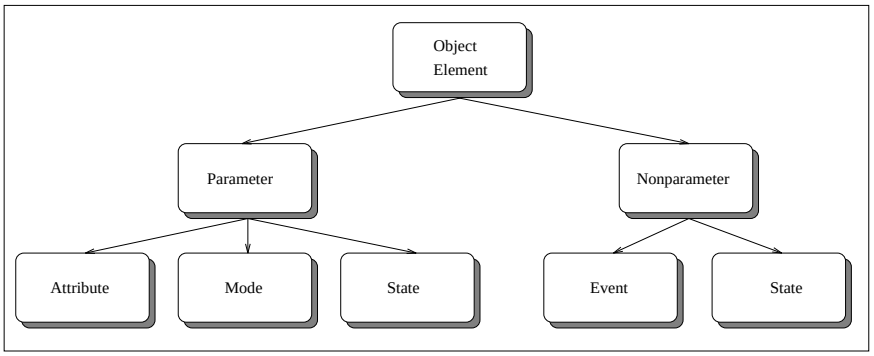

Figure 3: Object Element Classification

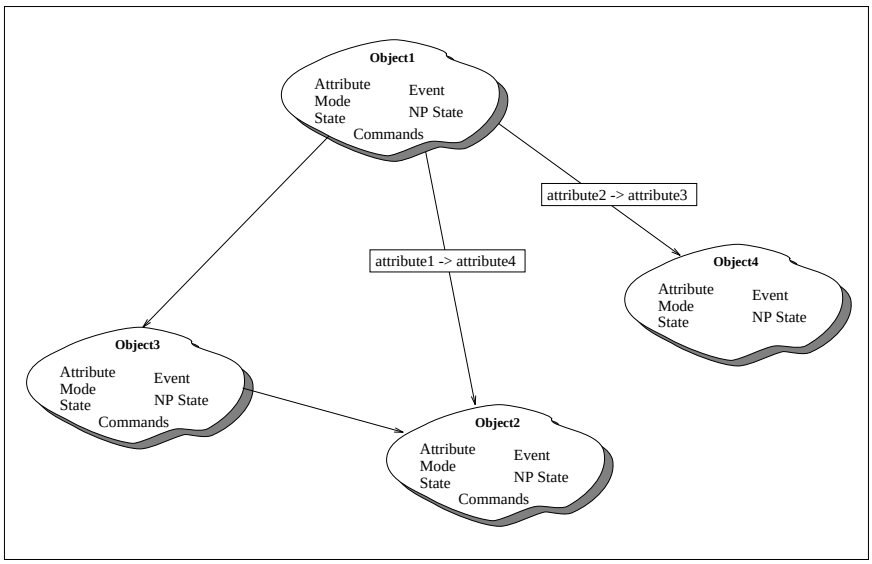

Figure 4: Generic Object Model

nonparameter elements. These represent *events* or *states* that occur as an implicit consequence of the dynamics of system operation but are not directly set via parameter values in commands. They are,however, important for test data generation to provide semantically correct command sequences as events and states may cause certain commands or parameter values to be no longer valid for semantically correct commands (e.g. if a tape library is full, I cannot enter another tape into it, yet I cannot set a library to "full" with a command).

Once the objects and their elements are defined, we need to show object relationships. In Domain Based Testing, relationships between objects define semantic rules about parameter values. These relationships are captured in an *object hierarchy*. Figure 4 shows a generic object hierarchy with four objects. The existence of a relationship between objects is shown by an arrow from one object to another. The arcs are annotated with detailed information about the relationship or parameter value constraint. In Figure 4 the annotation between **Object 1** and **Object 2** indicates a relationship between **attribute 1** of **Object 1** and **attribute 4** of **Object 2**. For "small" objects, rules and attributes can be listed directly in the object model figure. For larger objects, a table for detailed information is more appropriate to prevent clutter and unreadable graphics. Table 2 shows a generic attribute table for a "large" object. In particular, the attribute definition table lists all possible values for an attribute. The relationship definition table describes relationships between object attributes in more detail. In our domain analysis for testing, all rules so far have been *parameter inheritance rules*, i.e. the value of one attribute constrains the value of another. Mode, state, and event object elements on the other hand appear in rules for parameter selection at the command level and constrain which commands are semantically valid for a script (dynamic behavior).

Next, command syntax and semantic rules are defined for each command. These are used later to generate command templates. The semantic rules ensure the test is meaningful. Three types of semantic rules exist for commands: *Preconditions*, *postconditions*, and *intracommand rules*. *Preconditions* identify the conditions that must hold **before** the command can execute. *Postconditions* list the conditions that are true **after** the command executes. Preconditions ensure proper state or mode for a command. They also may further constrain valid parameter values. Postconditions state

Rule Type	Rule Specifics
O1.attribute1 → O2.attribute4	(O1.attribute1 = val1) → O2.attribute4 ϵ subset(O2.attribute4)
O1.attribute2 → O3.attribute3	(O1.attribute2 = val1) → O3.attribute2 ϵ subset(O3.attribute3)

Table 2: Generic Attribute Table for a "Large" Object

effects on object elements and influence future command sequences or parameter value selection. *Intracommand rules* identify constraints placed on parameter value selection within a command. Intracommand rules thus handle semantics of object manipulation via commands. An example of such a rule is "you must not copy a file onto itself." This leads to a rule `filename1` $\neq$ `filename2` in a copy command template such as:

```
copy <filename1> <filename2>
```

Such rules are only active during the execution of the command with which they are associated.

At this point of analysis we have a static model of our domain, its objects, commands that manipulate objects, and the rules that govern such manipulation. We have not yet analyzed dynamic behavior. The dynamic behavior model of our domain describes rules for possible consequences of actions (in our case, command sequences). It corresponds to the scripting phase during test generation. Sequencing information is necessary because arbitrarily ordering a list of commands rarely produces semantically correct test cases [5]. Besides capturing dynamic system behavior, scripting allows the test engineer to develop meaningful test cases at a high level of abstraction. Two types of semantic rules are associated with the scripting phase:

1. Command Sequencing Rules

   In some systems, commands must be issued in a particular order. It may not make sense to issue some commands before others. For example, is does not make sense to "dismount" a tape from a tape drive unless one has previously been "mounted." Semantic rules like this form "bracketing" information that must be obeyed for meaningful test generation.

2. Script Parameter Selection Rules

   This type of script rule defines how to choose parameters for script sequencing. Table 3 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. Script rules are valid for the duration of a command sequencing rule. To illustrate, the `MOUNT - DISMOUNT` sequence is annotated with script parameter selection rules.

[5] We experimented with a prototype [2] that used no sequencing rules and found that only about 30%-40% of the test cases were meaningful without sequencing rules.

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: Script Rule: Parameter Value Selection

`MOUNT` *tape-id*`*`
<A list of commands>
`RELEASE` *tape-id*

This rule states the *tape-id* parameter can be selected from any valid choice for the `MOUNT` command. The `DISMOUNT` command must use the previously bound value for the *tape-id*. Simply stated, the tape that is mounted should be the one that is dismounted.

These three aspects of domain description form a domain model that is sufficient to generate meaningful test cases for command-based software. As we will see in the next section, it is also an excellent basis for automated test suite generation.

3 Automation and Experience

Sleuth is an automated test generation tool developed at Colorado State University. *Sleuth* supports Domain Based Testing by providing tools and utilities to automatically generate tests. The first step for any new problem domain is to capture the testing domain. *Sleuth* does this during a *System Specification* phase in which the user defines a domain model in terms of command language syntax, script, command, and parameter inheritance rules. Using the specification, test engineers are given a "work space" to configure the system for specific tests. Specifically, they can turn commands and sets of commands `on/off`, they can also turn scripting or command rules `on/off`. They can use or decide to switch off object inheritance rules. Sets of possible parameter values can be edited. While in the scripting phase, testers can define their own "megascripts" using classes of commands or importing existing tests (e.g. "bugfinders" that have worked well in the past). This turned out to have great practical value for our test engineers as they could reuse and leverage existing test cases while at the same time enjoying the benefits of *Sleuth*'s new technology.

Figure 5 shows the main window of the tool. Test generation follows the three phase approach outlined above. First, the scripting phase generates a list of command names. Second, a command template is created for each command by taking a random-walk through the command's syntax diagram [6]. Third, parameter values are selected based on the current system configuration. At each phase only those rules are applied that are associated with a particular phase, i.e. scripting uses

[6] *Sleuth* allows command syntax to be represented and edited as syntax diagrams or BNF. Syntax diagrams were vastly more popular since they appear to be easier to understand and work with.

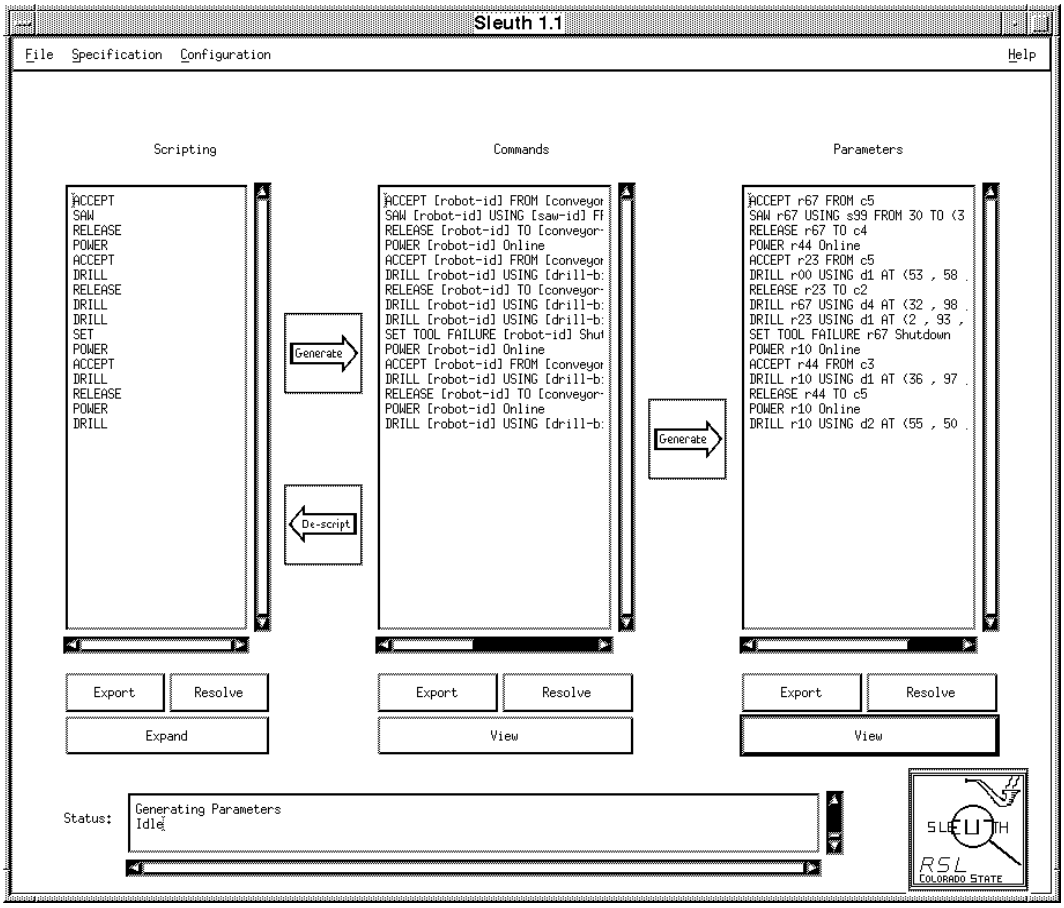

Figure 5: Window Based Test Generation Tool

command sequencing rules, command template generation applies command rules, and parameter selection utilizes parameter (inheritance) rules. This greatly simplifies the test suite generation process and reduces the number of active rules at any one point. It avoids some of the complexity problems of other approaches while making it possible to turn various rules `on/off` at will. This latter capability was of particular interest to our test engineers.

Test engineers can archive tests at each phase of test generation using the `EXPORT` button. By saving tests at each phase, we can reuse tests in a variety of ways. Figure 6 illustrates the potential of this three phase approach for test case reuse. From a single script, we can generate several different command template sequences. Each command template would be specific to a software release, domain configuration, or semantic rule set up. For each command template, *Sleuth* can instantiate different parameter values and thus generate test cases using different system configurations, parameter values, or parameter inheritance rules. Through the `DESCRIPT` button this mechanism also works in reverse, i.e. a command can be turned into a template, a set of command templates can be turned into a sequence of command values. These can then be "re-generated" using different system configuration, rules, etc. These mechanisms provide the infrastructure for test case reuse. We explore its practical implications in the next section. Through test case reuse, test engineers become more productive, they can re-run tests, and they can recall and modify test cases easily. Combining test cases also provides a mechanism to create stress tests for the system. Table 4 lists how one could generate a reusable script, command template, or test case.

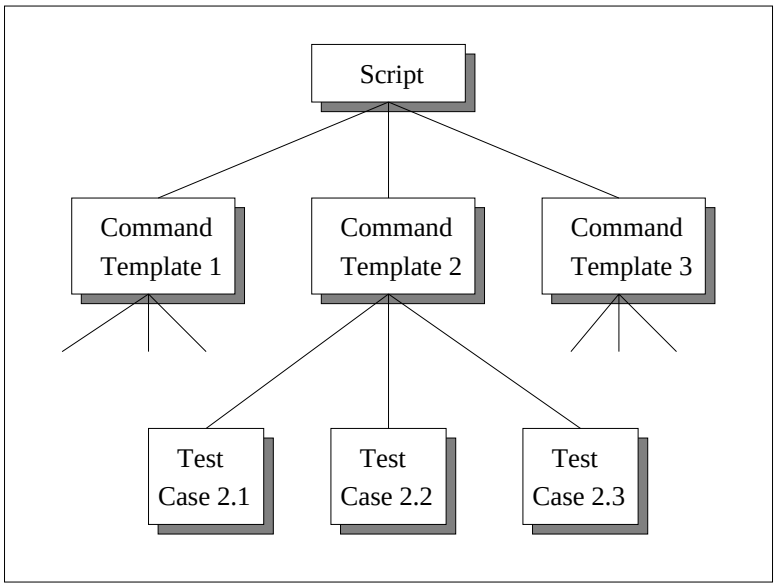

Figure 6: Test Case Reuse Concept

	Steps in *Sleuth*
Reuse Script	1. Generate Command List 2. Export Command List
Reuse Command Template	1. Generate Command List 2. Generate Command Template 3. Export Command Template
Reuse Test Case	1. Generate Command List 2. Generate Command Template 3. Generate Parameter Values 4. Export Test Case

Table 4: How to Create Reusable Tests in *Sleuth*

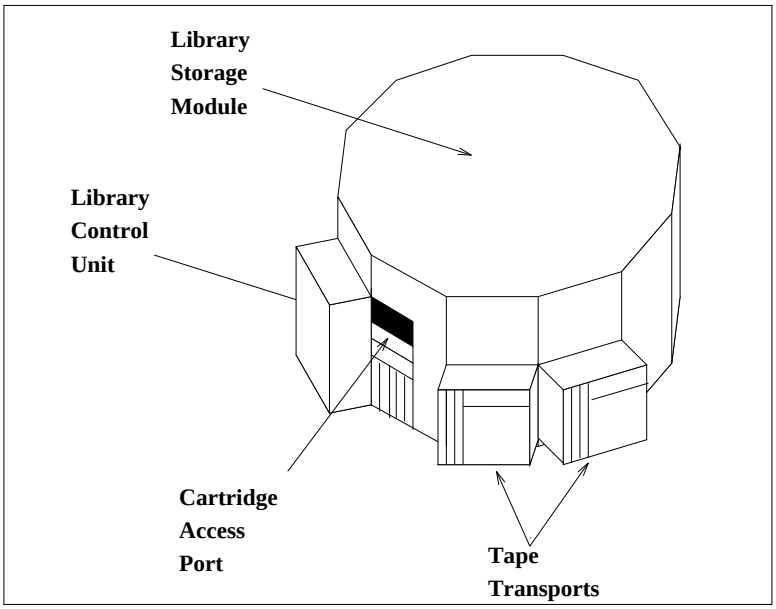

Figure 7: ACS Hardware [11]

4 Example Problem Domain - Robot Tape Library

StorageTek Corporation produces an Automated Cartridge System (ACS) that stores and retrieves cartridge tapes [11]. The system maintains cartridge tapes in a 12-sided storage device called a Library Storage Module (LSM). Tapes are placed in storage cells in the outer and inner panels. New tapes can be entered into an LSM through a special door called a Cartridge Access Port (CAP). Figure 7 shows a single LSM with tape drives, access panel, and control unit. The robot inside the LSM identifies tapes using an optical scanner. Once a tape is identified, the robot can move the tape to a cell, mount the tape in a tape drive, dismount tapes, or eject tapes through a CAP. An ACS can support up to sixteen LSMs. Figure 8 shows a "top-down" look at three LSMs and the associated hardware. Tapes can move between LSMs through special doors called "pass-through-ports."

The ACS and its components are controlled through a command language interface called the *Host Software Component* (HSC). The commands manipulate cartridges, set the status of various components in the system, and display status information to the operator's console. The command language consists of 30 commands and 45 parameters. The object hierarchy for the **StorageTek** problem domain is shown in Figure 9. The hierarchy also identifies the *parameter inheritance rule classes* for the attributes of the objects. On some of the arcs, specific constraint relations are annotated. For example, the choice of a particular ASC (*acs-id*) restricts the choices for the LSM (*lsm-id*). An annotated attribute table would list specific values. Note that we find all three types of object elements, e.g. the LSM object has attributes (*lsm-id*), modes (*lsm-subpool-threshold, lsm-scr-threshold*), and state (*lsm-status*) elements.

Table 5 shows a list of possible test case reuse scenarios at all three levels of test case generation.

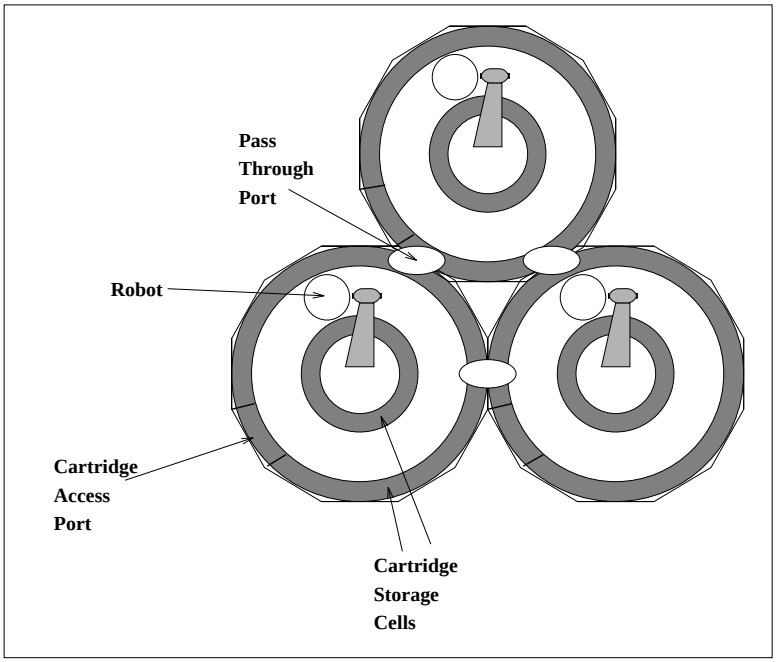

Figure 8: Library Storage Module - Top View [11]

Many of these applications were suggested by the test engineers at **StorageTek**. We find three classes of reuse scenarios: (1) Varying Software Releases, (2) Varying System Configurations, (3) Varying Test Case Construction. The testers at **StorageTek** are using *Sleuth* with all three. The following sections provide a short overview of each class and show how the test engineers at **StorageTek** employ *Sleuth* to reuse archived tests. Even though the examples presented here are specific to the robot tape library, it is reasonable to assume that other systems will have similar reuse needs.

4.1 Scenario #1 - Software Releases

Between releases, the command language for command-based systems may change. Typically, this is represented as a new version of the command language or as a new release of the system. Usually, a new software release is similar to the previous one. New commands may be added, obsolete commands are deleted, or command syntax is modified. In some cases, a new release may influence semantic rules like command pre/post conditions or parameter inheritance rules.

At **StorageTek**, test engineers use HSC commands to generate tests for the robot tape library. These tests can be archived at all three phases of test case generation. When a new release of the software is issued, archived test cases can be recalled at the *Scripting* level for reuse. Once the script is recalled, a command template can be generated using the updated command syntax and semantic rules. Each command template can be used to generate tests for a wide variety of hardware configurations. **StorageTek** has a requirement that software should be upwardly compatible from release to release. Therefore, test cases that ran without incident on one release should run without incident on the new release. *Sleuth* provides utilities to recall test cases for this simple form of

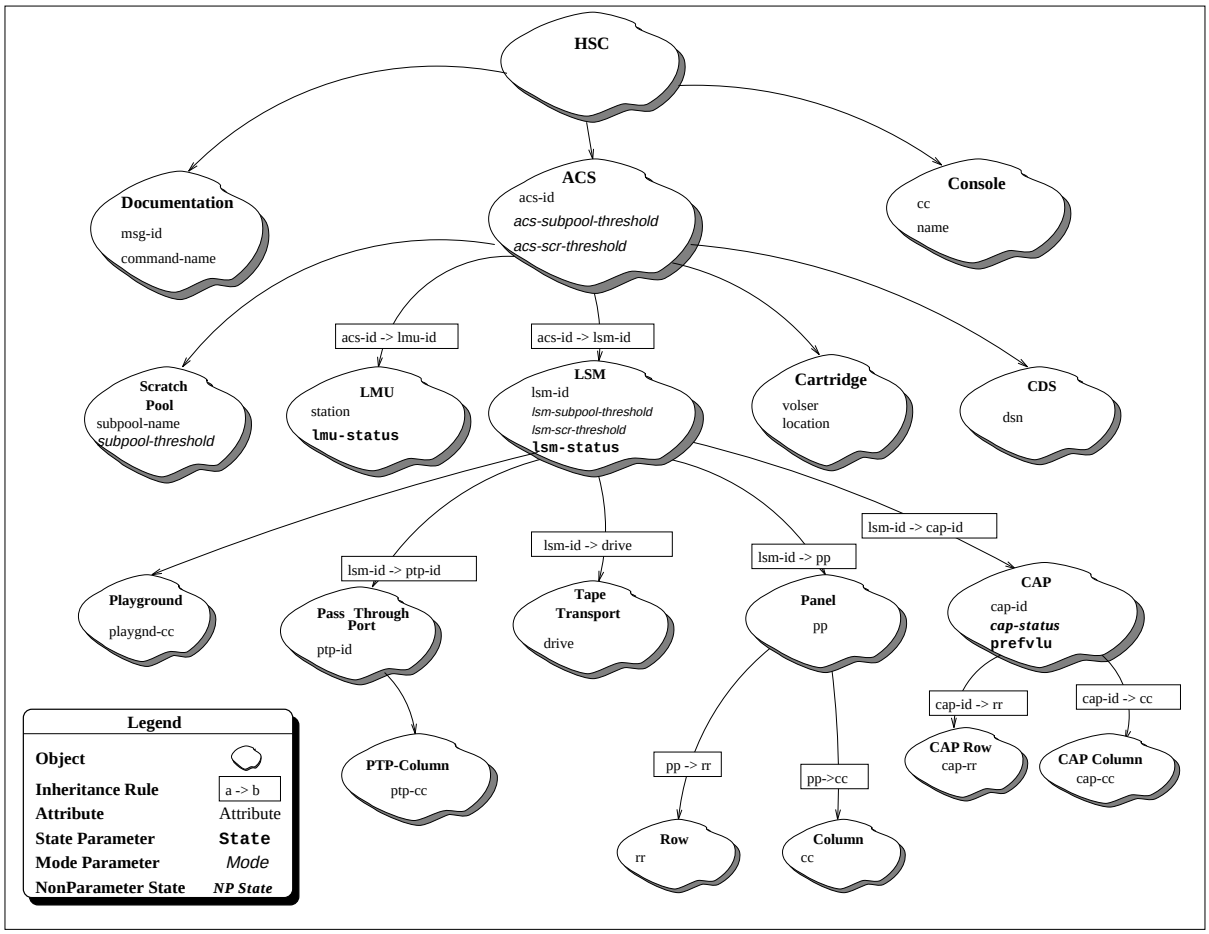

Figure 9: StorageTek Object Hierarchy

	Reuse Possibility
Reuse Script	Regression Testing Command Syntax Change New Software Release Stress Test Creating new test scripts Operating System Version
Reuse Command Template	Regression Testing Domain Specification Change Domain Configuration Change Hardware Configuration Change Stress Test Creating new command templates Parameter Value Change
Reuse Test Case	Regression Testing Re-run Test Case Creating new test cases Stress Test

Table 5: Domain Based Testing - Reuse Scenarios

regression testing.

This same reuse scenario can be used for operating system versions. Many times command-based systems are used across different operating systems. There may be many commands in common, but some slight differences in the command language. For example, some commands may be needed for one OS and not another. Using this same scenario, test cases can be recalled at the *Scripting* level, and new tests can be generated for various OS versions. Since all test suites are identical at the Script level, we build uniform, comparable test suites for a variety of releases and platforms.

4.2 Scenario #2 - System Configurations

In general, test cases for command-based systems can be changed by altering the parameter values that can be selected. The set of parameter values represents a configuration of the problem domain. Sometimes it represents a configuration of a logical object while other times it may represent a configuration of physical devices.

The **StorageTek** robot tape library hardware can be configured in many ways. To test these configurations, testers can execute tests on a Library Management Unit (LMU) simulator or on actual ACS hardware. The simulator is convenient for testing because it can model a customer's installation or it can be set up to model any valid ACS configuration. ACS hardware can be configured in many ways. Tests run on the actual hardware is typically used to test timing problems or real-time processing faults. Each ACS configuration needs a separate set of parameter value files

and parameter inheritance rules.

For this scenario, tests will be reused at the *Command Template* level. Upon recalling a command template, test cases can be regenerated based on the new configuration. Hence, a single command template can generate a test case for each parameter configuration. This saves generation time, but more importantly make test cases comparable because at the scripting and command template level they test "the same thing."

4.3 Scenario #3 - Test Case Construction

Often testers find a particular list of commands is good at detecting an error. Test engineers may also have a set of commands that puts the system in a particular state before running a test case. It should be easy to recall such test cases and to include them in a new test suite. In *Sleuth*, this reuse scenario spans all three phases of test generation. For instance, a new test case can be constructed from a set of "building blocks." One script is included to put the system in a particular state, another presents a workload to the system, and then a third tests for a particular fault. Command templates can be included to create a larger test sequence. Finally, archived test cases can be included to form a stress test for the system. To support test case construction, *Sleuth* provides a feature to *include* archived scripts, command templates, and test cases.

In a similar scenario, testers should be able to merge several tests. Merging test cases takes several archived files and "shuffles" them into a single test suite. This scenario is important when testing shared devices. It provides a simple way to interleave commands from several users. Merging tests can be applied across all three phases. *Sleuth* provides a feature to *merge* scripts, command templates, or test cases.

5 Summary

Domain Based Testing currently is a test method for command-based systems. Using a Domain Analysis and the resulting Domain Model, DBT provides a structure for generating test cases and for test case reuse. In this paper, we described the general ideas for test case reuse and listed three reuse scenarios. The concepts of Domain Based Testing have been implemented in a test generation tool called *Sleuth*. *Sleuth* is currently in use at **StorageTek**. The following benefits from Domain Based Testing have been reported:

- It is easily to test a product enhancement
- It allows test engineers to "sleuth" for bugs
- It supports stress testing
- It identifies real-time bugs and hardware timing errors
- It generates tests against requirements specifications while allowing testers to follow their natural inclinations, i.e. thinking in terms of *what* they want to test (namely *objects*)
- It allows operational profile testing (*Sleuth*'s test case construction capabilities)

- It tests new hardware configurations
- It increases tester productivity
- It provides test case reusability and uniform testing across releases and configurations

Our efforts have shown that domain based testing is useful for system test engineers at **Storage Technology** for an automated tape library. We now need to evaluate how well the test generation method works in other industrial settings, and we need to apply it to other problem domains. Feedback from different domains will either reinforce our DBT method, or it will give us insight about how we can improve it. At the same time we are also investigating how to build domain models for graphical user interfaces (GUI) from which we can generate sequences of test instructions in a language that captures GUI-based system use for replay (e.g. visual basic, Test System Language (TSL), and Test Command Language (TCL)).

References

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

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

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

[4] Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, Volume I : Concepts and Models, Frontier Series, ACM Press, 1989.

[5] 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*, 1980, vol. 10, pp. 897-918.

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

[7] 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, June 1991, pp. 591-603.

[8] Charles Krueger, Software Reuse, *ACM Computing Surveys*, (24)2, pp.131-183, June 1992.

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

[10] Kenneth S. Rubin and Adele Goldberg, "Object Behavior Analysis", *Communications of the ACM*, 35(9), pp.48-62, September 1992.

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