Using Domain Models for System Testing

A. von Mayrhauser
Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523

R. Mraz
Dept. of Computer Science
U. S. Air Force Academy
Colorado Springs, CO 80841

Abstract

Domain Models [8, 9, 25] have long been used as a basis for software development and reuse. We present a specialized, simplified domain model that has been used for system testing in industry as the framework for a system testing approach we call Application Domain Based Testing. Sleuth*, a test suite generation tool, is based on this concept. We report on the domain analysis, the domain model components, and industrial experience of reusing domain models and tests generated with* Sleuth.

Key words: Domain models, reuse, system testing

1 Introduction

System test is typically the last step before release or beta-test of a software product. System tests examine a software product against requirements and specifications [7, 20]. Many testers also derive tests primarily from user manuals. When software development is based on a domain model, the software could be tested using the domain model specification and the specifics of the user interface [3, 4]. To analyze, design, and implement a system, developers need detailed domain model specifications, well defined abstractions, and rules for extending the domain model into new problem domains. In contrast, system testers may not need all features of the domain model and therefore could rely on a subset, especially since system testing is usually black box. In addition, when the objective is to merely generate system level tests rather than automate both test generation as well as validation of executed tests, the domain model can be further simplified, as it will not need information for test validation.

This chapter explores the use of a simplified domain analysis technique and domain model as a framework for system testing. The domain model describes semantics of the application that are relevant for testing. It also captures user interface specifics, such as command language syntax or a formal, syntactic model of the GUI [35]. We propose a domain analysis technique and an associated domain model for testing software and software/hardware systems with a command language interface. The technique was also applied to a transaction processing system [36] to show that it is suitable for testing telecommunication systems and databases, as well. The purpose of the specialized domain model is to generate tests for a domain while accommodating a flexible set of test objectives. Test objectives are taken into account by building test subdomains and by adding test criteria rules to the test case generation method.

Application Domain Based Testing addresses the need of software testers for a tool that supports their thought processes. Test generation addresses three levels of abstraction: the process level (i.e., how the target software commands are put together into scripts to achieve high level tasks), the command level (i.e., which specific commands are included in the scripts), and the parameter level (i.e., particular parameter values used in command templates).

The next section describes approaches used in software testing. Section 3 introduces a mass storage device which was used in an extensive empirical study of application domain based testing. Section 4 presents an explanation of the domain analysis for testing purposes, the components of the resulting domain model, and a model of the domain capture process. The application domain model (including the syntax) is the information the test generation tool uses to generate tests. Section 5 explains the test generation process. Section 6 presents a brief look at the test generation tool, *Sleuth* [39], and the methods it employs [41]. One of the purposes of domain models is their reuse potential. This is no different when a domain model is used for testing. Section 7 explores the reuse potential of this approach to

system testing and relates case study data from an industrial application at Storage Technology Corporation (StorageTek). Reuse occurred at two levels, the domain level and the test suite level. Conclusions summarize our experiences with this system testing approach to date and point to open research questions in the area.

2 System Testing Approaches

For systems with a command language interface, system tests consist of sequences of commands to test the system for correct behavior. Similarly, transaction based or request oriented systems can be tested by generating transactions or requests. Traditionally, test automation for both command language and transaction based systems is based on a variety of grammars or state machine representations.

2.1 Grammar-based Test Generation

Grammar based test generation represents each command using a grammar, generates sentences (commands) from the grammar, and runs the list of commands as the test case (for early work see [32, 5]). When generating a test case from a context free grammar, the generator has to decide which grammar productions to use and which choices within a production to select in deriving terminal symbols (the test case).

2.1.1 Generation Assuring Production Coverage

Purdom [32] resolves this through test criteria rules: Each production in the grammar is used at least once. In addition, the algorithm prefers short sentences when there is choice. In generating sentences, the algorithm uses two types of information,

- the production rule to use next on any of the nonterminals so that the shortest terminal string can be derived; and
- the production rule to use next to introduce a nonterminal that also ensures that the shortest sentence is derived which uses that nonterminal in its derivation.

Grammar	Example sentences
1. $S \rightarrow < E >$	`i+i`
2. $< E > \rightarrow < E > + < T >$	`(i+i)` $\uparrow$ `i`
3. $< E > \rightarrow < T >$	`i` $\uparrow$ `i`
4. $< T > \rightarrow < P > \uparrow < T >$	`(i)` $\uparrow$ `i+i`
5. $< T > \rightarrow < P >$	
6. $< P > \rightarrow$ `(` $< E >$ `)`	
7. $< P > \rightarrow$ `i`	

Table 1: Context free grammar and sample derivations[32]

The left column of table 1 shows an example of a simple grammar for arithmetic expressions [13]. S is the start symbol, $< E >$, $< T >$, and $< P >$ are nonterminals, `i`, `+`, $\uparrow$, `(`, `)` are terminal symbols. The right column shows possible derivations. The last is the one that would be derived with Purdom's algorithm [32]. It is also the only sentence needed to fulfill his testing criterion, as it uses all productions of the grammar. Purdom used this method to test parsers.

2.1.2 Attribute Grammars

When using grammars for test case generation, we also need to address command language semantics ([6, 11, 15, 23]. One way to do this is to use attribute grammars for test case generation ([15], [40]). The syntax and semantics of the command language are encoded as grammar productions. Test case generation is a single stage algorithm.

Duncan and Hutchison [15] used an attributed, context-free grammar to formally describe classes of inputs and outputs based on a design description for the purpose of generating large classes of test inputs and expected outputs. Test cases should be semantically meaningful. As in other grammar-based approaches, the basic syntax is given in the form of an EBNF (e. g. table 1). To this basic grammar, Duncan and Hutchison [15] add additional information to capture sematic rules and expected output. Attributes incorporate context-sensitive information, such as relationships between parameters of a command language

(non-terminals of the grammar). When processing attributes, designators assign values to attributes. They are expressed as individual values, ranges, or boolean conditions. Guards are boolean expressions used in attribute processing. They can involve inherited attributes of a left-hand side nonterminal, synthesized attributes of symbols to the left of the predicate in a production rule, or the value of the current terminal and any of its synthesized attributes.

Also associated with the right-hand side of grammar productions are action routines. They produce the expected output associated with the generated test input (the sentence derived using the grammar). This is called grammar output value. In effect, the attributed grammar provides a partial specification of the system under test. This specification can get quite complex. Not surprisingly, this approach had scaling problems and thus could not be used for large systems [22]. As systems to be tested become more complex, the grammars do, too. This limits performance, but also means that for the average system tester these grammars are difficult to write and maintain and that the generation process does not follow the test engineers' thought processes, particularly in terms of testing goals and refinement of these goals at successive levels of abstraction.

In addition, heuristics need to be defined to guide the generation process during sentence derivation in the choice of productions and attributes. Duncan and Hutchison [15] consider this an open problem related to grammar based test generation.

```
1. testcase → [#N in 0..MAX_N] @init sort_input(N)
2. sort_input(N) → {N: [#j in 1..MAXBUCKETS] ``element''(J)
        @put_terminal(J)} ``end_input'' @put_end
```

Table 2: Attributed test grammar for sort program [15]

Table 2 shows an example of an attributed grammar to test a sort routine [15]. In the first production, `# N` is a designator (used for choosing the length of the vector to be selected, the `in` clause specifies the choices for this selection. `@init` is the action routine that initializes the expected output (zero for all possible value buckets). `sort_input` is

the nonterminal derived in this production. It is associated with the attribute `(N)` bound through the designator. The second production transforms this attributed nonterminal into the unsorted list by generating `N` elements in the range of `1..MAXBUCKETS`. As each value is generated, the action routine `@put_terminal` puts it into the appropriate value bucket. Then the terminal string `end_input` is generated and the action routine `@put_end` uses the knowledge of how many elements are in each value bucket to generate the sorted list (the expected output).

2.2 State Machine Based Test Generation

Transaction based systems and state transition aspects of some other systems have been tested using state machine representations [12, 16]. Test generation based on a finite state machine representation includes Transition tour [26], W-method [12], and partial W-method [16], the Distinguishing sequence method [18], and the Unique-input-output method [34]. Their objective is to detect output errors based on state transitions driven by inputs. Not all methods can guarantee finding all transfer errors (i. e. when an incorrect state is reached). Finite State machine based test generation has been used to test a variety of applications, including lexical analyzers, real-time process control, protocols, data processing, and telephony.

We use the W-method [12] to illustrate this approach to test generation. The W-method assumes a minimal finite state machine with a finite set of states, all of which are reachable, and a fixed initial state. Test sequences consist of two parts $p \in P$ and $z \in Z$. P is the set of all partial paths through the finite State machine (including the empty path). P is determined by constructing a test tree. The distinguishing set Z is based on the characterization set W of the design underlying the Finite State machine. W consists of input sequences that distinguish between every pair of states in the minimal automaton. Sequences in Z consist of a sequence of characters from the input alphabet of the Finite State machine that can be as long as the difference in states between the design and the minimal Finite State machine.

They end with a character from W. The resulting test sequences (and the associated outputs according to the Finite State machine) are executed and the Finite State machine output is compared to the actual output. Test sequences can detect output or transfer errors, given some restrictions on the number of states in the implementation. The partial W-method [16] improves over the W-method by reducing the length of the test suite. It does so by reducing the characterization set W to subsets W_i for specific states i.

State machine representations work well for generating sensible sequences of command types, but become cumbersome for generation of both sequencing as well as command details of systems with large and intricate command languages.

2.3 Frequency-based Test Generation

Automatic generation, whether based on grammars or state machines, requires making choices during the traversal of the representations. The choices are due to ambiguities as well as the purposeful inclusion of options in the representation. Choice is directed by incorporating selection rules of various types. As mentioned earlier, Purdom [32] integrates "coverage rules" for grammar productions to reduce choice. Maurer [27, 28] uses probabilistic context free grammars that are enhanced by selection rules including permutations, combinations, dynamic probabilities, and Poisson distribution. The following is an example of a production that includes frequency rules for value selection:

`bnumber` → `%32bit`
`bit`: 31:1, 1:0

This production states that when deriving from the nonterminal `bnumber` the nonterminal `bit` is generated 32 times. When choosing a terminal value for each nonterminal `bit`, the value 1 is to be chosen 31 times more often than the value 0. Like [15], Maurer [27, 28] also allows action routines to compute expected output of the test.

In all these methods, selection is based on making choices relating to the representation of the command language or state machine.

2.4 Partition Testing

Alternatively, one can argue that choices should be made depending on the functional characteristics of the system. Functional testing according to [30] uses heuristic criteria related to the requirements. Each requirement is identified, related to a (set of) commands, and valid, boundary, and invalid cases are tested. So are heuristically determined combinations of commands. The method is manual.

2.4.1 Goodenough and Gerhart [17]

Goodenough and Gerhart [17] suggest partitioning the input domain into equivalence classes and selecting test data from each class. In this approach, we divide possible input values into equivalence classes based on common effect. It is enough to test one input from each class. Classes can be subdivided further by including other criteria, such as executing the same portion of the code, or being part of the same algorithm, although this is no longer black-box testing as it would include knowledge of how functions are designed and implemented.

2.4.2 Category-Partition Testing

Category-partition testing [2, 31] accomplishes this by analyzing the specification, identifying separately testable functional units, categorizing each function's input, and finally partitioning categories into equivalence classes. Category partitioning starts with the functional specification, then goes through a series of decompositions of items and inputs related to the specifications to the level of individual subprograms to be tested. The first level of decomposition identifies functional units (e. g. a top level user command). The next step identifies parameters and environment conditions that affect functional behavior. Next, careful reading of the specification identifies categories of information that characterize each parameter and environment condition. Each category is partitioned into distinct choices (possible set of similarly treated values). These choices form the partition classes. Choices can be annotated with constraints to model relationships between categories. While the analysis is

manual, a generator tool produces test cases from these (constrained) partitions. To avoid too many error test cases, choices can be marked `[error]`, in which case they do not get selected, unless error recovery testing is a test objective. Others can be marked `[singl]` to avoid combining them with choices from other categories.

Category	Choice
pattern size	many characters
quoting	pattern is quoted
embedded blanks	several
embedded quotes	one
file name	good file name
no. of occurrences in file	exactly one
pattern occurences on target line	one

Table 3: Example test categories and choices [31]

Table 3 [31] shows an example of a test case with choices identified for a command that looks for patterns in a file. The left column identifies the category, the right identifies the choice selected for the category. Substituting values for each choice produces the test input.

Richardson et al. [33] consider these approaches manual, leaving test case selection completely to the tester through document reading activities. Further, partition-testing as a testing criterion does not guarantee that tests will actually uncover faults [19, 38, 45]. From a practical standpoint, a better approach is to combine different test generation methods with a variety of testing criteria. Examples are to combine exhaustive generation of some commands or parameter values with probabilistic or combinatorial criteria for others, which requires flexible command generation methods.

2.5 AI Methods in Test Generation

2.5.1 Planner-based Test Generation

So far, few approaches to system testing use artificial intelligence methods. The most relevant is Howe et al. [21] which uses an AI Planning system (UCPOP) to generate system

tests (black box testing).

The planner generates test cases in three steps: generate a problem description, create a plan to solve the problem, and translate the plan into test case notation. These three steps correspond to three modules: preprocessor, planner and postprocessor.

The preprocessor develops a problem description based on user directions. The problem description consists of a problem name, domain knowledge, an initial state and a goal state. The problem name is generated automatically. The domain knowledge describes the command language semantics. The initial and goal states define the specific needs of a particular test case.

The preprocessor incorporates knowledge about how command language operations relate to changes in the state of the system. The user indicates how many of the different types of operations should be included in the plan. Based on knowledge of the test domain, the preprocessor creates an initial state and goal state description that would require using the indicated commands. The initial state also includes information about system configuration and initial state information. The configuration information is taken directly from the knowledge base and the initial status information is randomly generated from the problem constraints.

The planner constructs a plan to transform the initial state into the goal state. If a plan cannot be found within a set amount of time, the planner fails. In this case, the preprocessor generates different initial and goal states that satisfy the user's requirements. The postprocessor translates from plan representation to command language syntax. The transformation is purely syntactic and straightforward.

Table 4 summarizes representation of Domain Knowledge in the AI Planner.

The results are encouraging, in that the planner was able to come up with novel tests. Generation speed and with it scale up are an issue. Possible further work to remedy that include investigation of hierarchical planning methods or subsumption as in CLASP [14].

Domain Knowledge	Planner Representation
Command Sequencing Rules	Operator preconditions
Parameter Binding	Operator effects
Command Language Syntax	Operators : One operator for each “path” in a command. Postprocessor : Translates Planner output into Command Language Syntax.
Command Preconditions	Operator preconditions
Command Postconditions	Operator effects
Parameter Constraint Rules (within same command)	Preprocessor : Initial State Generator Preprocessor : Goal Generator Preprocessor: supporting data structures Operator preconditions

Table 4: Domain Knowledge and AI Planner Representation [21]

2.5.2 CLASP[14]

CLASP uses terminological plan-based reasoning to generate test scenarios and test scripts. This approach combines plan based reasoning with regular expression processing. Subsumption helps to manage large collections of plans. CLASP implements a general approach for both plan synthesis and plan recognition, testing is only one of its possible applications. Consequently, it does not include any guidelines or process how to derive a CLASP description for testing purposes.

2.5.3 KITTS

KITSS [24] is a knowledge-based translation system for converting informal English scenario descriptions of desired system behavior (informal test descriptions) into formal executable test scripts. Its premise is that language used in informal test scenarios is not really standard English, but a stylized technical subdialect and thus amenable to automated or at least machine assisted natural language processing. In doing so, the translation system must also bridge abstraction gaps between the goal-oriented intent of the testers when writing informal test scenarios, and the specifics required in executable scripts. In addi-

tion, informal scenarios can have other problems such as missing steps, hidden or implied effects, underspecification, missing initialization and boundary conditions, and erroneous statements. KITSS includes a knowledge base describing telephony testing. Converting the English scenario into intermediate form (WIL) is accomplished with an adaptive statistical parser and a rule-based phrase converter. The statistical chart parser computes probabilities to rank alternatives due to grammatical ambiguity. User input adapts the parser to learn and define previously unknown language constructs.

Scenario understanding and analysis is done on the intermediate representation (WIL) with the aid of a coarse black-box simulation of a telephone switch, a telephone domain theory, and a library of typical phone usage patterns. Table 5 shows an example of a test scenario as written by the tester and the corresponding WIL language statement (Kelly and Jones [24] do not provide the corresponding executable test script).

scenario	place calls to station B3 and D1 and make them busy
WIL	Busy-out station B3 Busy-out station D1

Table 5: Behavior scenario and corresponding WIL statement

This is a viable technique for translating some English documents into an intermediate language and from there into executable test scripts. This technique requires that relevant information in English is available. It also requires building a domain theory of the application that is fairly sophisticated. It does not handle graphical descriptions like syntax diagrams.

Following a similar philosophy, Zeil and Wild [46] describe a method for refining test case descriptions into actual test cases by imposing additional constraints and using a knowledge base to describe entities, their refinements, and relationships between them. This is considered useful for test criteria that yield a set of test case descriptors which require further processing before usable test data can be achieved.

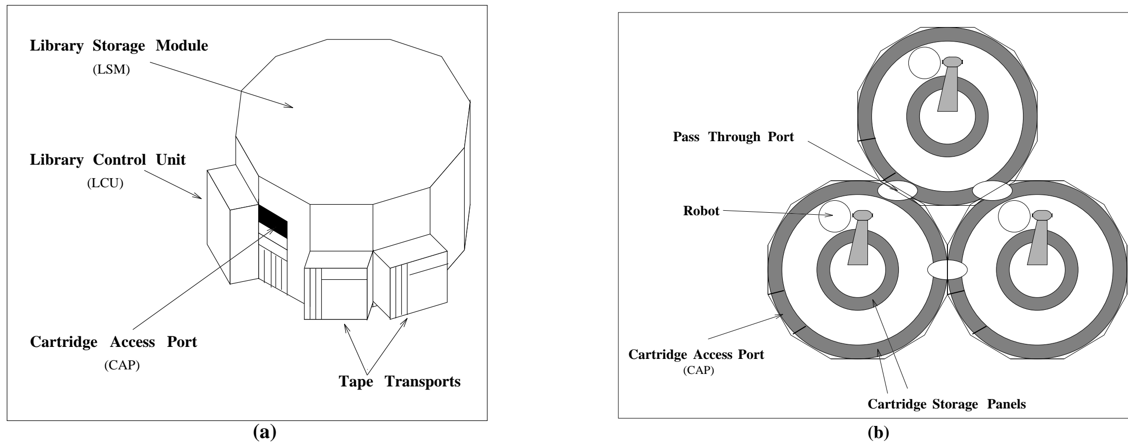

Figure 1: Automated Cartridge System with Three LSMs [37]

2.5.4 Neural Nets

Automated test case generation can easily generate tens of thousands of tests, particularly when random or grammar based methods are chosen. Running them takes time. To avoid running test cases that are not likely to reveal faults, a neural net has been successfully trained as a filter ([1]). After a subset of generated tests has been run, results indicate whether or not the test revealed a fault. The neural net is trained on test case measurements as inputs and test results (severity of failure) as output. This is then used to filter out test cases that are not likely to find problems. The results are encouraging for guiding testing.

3 Industrial Application - Robot Tape Library

System testing based on a domain model has many applications. Experience with this testing approach spans a wide range of application domains. They include: mass storage devices at StorageTek (HSC, IXFP, Snapshot), a database query/update system for cable television and generation of SQL code, USWEST's personal voice messaging system [36] and a spacecraft command and data handling application [43].

StorageTek produces several mass storage devices for mainframes and distributed workstation environments. Access to the mass storage devices is through a command language. System administrators use commands to configure the mass store, set up volume libraries,

and control device access. End-users access data, set up personal volume libraries, and configure their storage through command line requests. The Automated Cartridge System (ACS), a mass store for tape cartridges, and its Host Software Component (HSC) serve as the primary example [37].

The ACS maintains magnetic tape cartridges in multi-sided "silos" called Library Storage Modules (LSM). Each tape holds 350 megabytes. The ACS product line offers several LSM sizes ranging from small (6 feet tall, 8 feet diameter) to large (7 feet tall, 12 feet diameter) models. Inside the largest LSM, storage panels on inner and outer walls contain up to 6000 cells for tape cartridges. System operators enter and eject tapes through a Cartridge Access Port (CAP). A vision-assisted robot moves tapes inside the LSM. The robot effector houses a bar code reader, an optical vision system, and gripper sensors. All three sensors are used to identify tapes, pick up tapes, place them in cells, mount them in tape transports (tape drives), dismount tapes, and move them to/from the CAP. Control hardware for the LSM and robot and tape transports are housed outside the LSM. Tape transports are high speed tape drives that transfer data to/from the tapes. Figure 1(a) shows a single LSM with tape drives, access port, and control unit.

One Automated Cartridge System supports from one to sixteen silos. Figure 1(b) presents a top-down look at an ACS with three LSMs. Tapes move between LSMs through pass-through-ports. The silos in an ACS can be physically arranged in a variety of ways. Figure 2 shows some of the possible configurations of five LSMs. The ACS and its components are controlled through a command language interface called the *Host Software Component* (HSC). Each HSC supports from one to sixteen ACS systems. HSC 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. Typical customers for mass storage systems like the ACS include financial institutions, Internal Revenue Service, Social Security Administration, spacecraft ground station operators, and weather researchers.

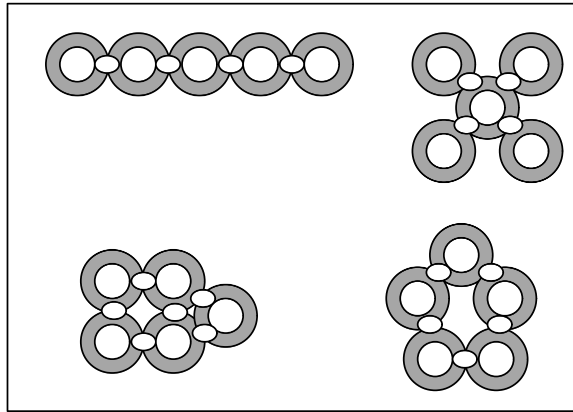

Figure 2: Possible Physical Configurations of Five LSMs

4 Building the Domain Model

Depending on whether the software to be tested already exists or is still under development, two possible ways exist to analyze a domain for system test: A Priori Domain Analysis and Reverse Engineering Domain Analysis. A Priori Domain Analysis develops the domain model as part of the functional specification phase of software development. Reverse Engineering Domain Analysis on the other hand, assumes that the software and its user manuals or functional specifications exist and develops the domain model based on that information.

A Priori Domain Analysis starts with domain objects, object attributes, their relationships, their actions, and rules for proper action sequencing (cf. Table 6). The last step is to represent the command language syntax. Syntax and some of the system specific values are the only parts of the application description that change between systems. Table 6 indicates this by separating command language syntax from the remainder of the model with a double bar. Reverse Engineering Domain Analysis develops the domain model starting with the syntax, adding semantic elements at each step, abstracting the objects and their behavior. Since many testers develop their tests after the software exists, rather than with the functional specifications, the latter is used more often.

These domain models for testing are simplified versions of a complete domain model that could be used for software development. When syntax information is included, one

Table 6: Steps in A Priori Domain Analysis

Analysis Step	**Model Component,** *data fields*
Identify objects	object list for object glossary *name, description*
Characterize object elements	Object element glossary *name, definition, values, representation*
Associate object elements with objects	Object glossary entries for associated object elements by type *parameter: attribute, mode state* *nonparameter: event, state* Object element glossary entry for associated object
Determine actions and action classes on objects and object elements	Action table *command name, object elements necessary* Object glossary *list of commands using an object* Script class table *class name, list of command names*
Identify relationships between objects	Object hierarchy
Determine constraint rules for object element values	Object element inheritance rules Intracommand rules
Identify pre/post-conditions for actions	Scripting rules Object element value binding rules
Command language syntax	EBNF or syntax diagrams

could also consider them very rudimentary functional specifications with just enough information to generate semantically meaningful tests. In and by themselves, these domain models do not prescribe any specific testing criteria. This is intentional as system testers use a variety of strategies during a testing cycle. Thus a domain model for testing should be flexible and adaptable to a variety of testing objectives, from functional to load testing. Separating the domain model from how it is used during test generation provides this flexibility. The domain model provides the framework for what is to be tested, test criteria applied during test generation determine how testing is to be done.

4.1 A Priori Domain Analysis

Table 6 shows the steps in the a priori domain analysis. The first step identifies physical and logical objects in the problem domain. Following Object Oriented Analysis/Design (OOA/OOD), analysts apply a variety of methods to identify objects [10]. The analyst can use any of the existing OOA approaches to identify objects. The difference between OOA/OOD and this approach is in the level of detail and the type of operation. Application domain models for testing do not need as much information as is necessary for implementation.

Table 7: Objects in the Robot Tape Library Domain

Object	Abbreviation	Description
Host Software Component	HSC	Operating system software used to control the robot tape library.
Documentation		On line documentation.
Console		Operator's console.
Automated Cartridge System	ACS	A collection of one or more LSMs.
Scratch Pool		Set of scratch cartridges.
Library Management Unit	LMU	Commands robot.
Library Storage Module	LSM	A single "silo" where cartridges are stored.
Cartridge		Storage medium.
Control Data Set	CDS	Contains volume information about all cartridges.
Playground		Reserved area for cartridges during LSM initialization.
Pass Through Port	PTP	Access door between LSMs.
Tape Transport		Tape drive that reads/writes cartridges.
Panel		Racks located inside an LSM. Used for cartridge storage.
Cartridge Access Port	CAP	A special door to enter and retrieve cartridges.
Pass Through Port Column		A column of cartridge locations on the PTP.
Row		A row of cartridge locations on a panel.
Column		A column of cartridge locations on a panel.
CAP Row		A row of cartridge locations on a CAP.
CAP Column		A column of cartridge locations on a CAP.

This specialized domain analysis for testing focuses on requirements documentation and its further analysis for object identification. [1] Consider the StorageTek HSC-ACS robot tape library. Cartridge tapes, tape drives, and tape silos are objects germane to this application domain. Objects included in the model control or manipulate the application, or are obvious parts of the system. Table 7 shows a list of all objects for the HSC-ACS domain.

Next, *object elements* define qualities and properties of the object. Object elements are similar to the concept of *object attributes* in OOA/OOD [10]. Object attributes and object elements differ in the amount of information that is described, because test generation does not need as much information about an object when compared to the amount of information needed for implementation. Object elements often identify an object, its state or operating mode, or its value. These attributes may be user controllable or not. In the first case, they will eventually become parameters in the command language; in the latter, they describe the effect of system operation on objects (events, states). This gives rise to the concept of

[1] When functional specifications are available, the command language has already been specified and the reverse engineering domain analysis described below is appropriate.

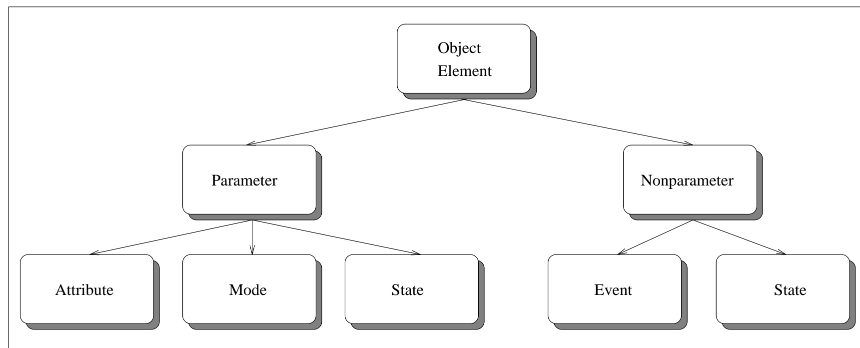

Figure 3: Object Element Classification

object element types. Application domain analysis classifies object elements into one of five mutually exclusive categories as shown by the leaf nodes in Figure 3.

The first level of classification determines whether the object element is controllable by the user or needed to submit requests to the system. These object elements will eventually be parameters of the application's command language. Object elements related to command language parameters can be *parameter attributes*, *mode parameters*, or *state parameters*. A parameter attribute uniquely identifies an object. Mode parameters set operating modes for the system under test. State parameters hold state information for the object. For example, an operating mode may be "verbose" or "quiet". A parameter state may be "tape drive A and B operational, use of C and D reserved".

Object elements which, while crucial for describing system operation, are neither controllable nor immediately visible to the user, are classified as *nonparameters*. They can be important for test case generation. A nonparameter event is an event caused by the dynamics and consequences of issuing a sequence of commands. A nonparameter state is state information that cannot be controlled through the command language. For example, a tape silo may become full as a consequence of system operation. We neither set a tape silo to `full` with a command, nor do we need to use the state of the tape silo in a command. At the same time, one cannot load any more tapes into a full tape silo. This affects test generation. For example, for testing regular system operation, generating further tape loads after

Table 8: Parameter Attributes in the Robot Tape Library Domain

Object	Attribute-id	Explanation
Host Software Component	host-id	Host identifier
Documentation	msg-id	Message identifier
Console	console-id	Console identifier
Automated Cartridge System	acs-id	ACS name
Scratch Pool	subpool-name	name for set of scratch tapes
Library Management Unit	station	LMU name
Library Storage Module	lsm-id	LSM name
Cartridge	volser	cartridge name
Control Data Set	dsn	data set name
Playground	playgnd-cc	name of playground column
Pass Through Port	ptp-id	Pass Through Port name
Tape Transport	drive	tape drive identifier
Panel	pp	panel number
Cartridge Access Port	cap-id	CAP name
Pass Through Port Column	ptp-cc	identifier for pass through port column
Row	rr	row number
Column	cc	column number
CAP Row	cap-rr	CAP row number
CAP Column	cap-cc	CAP column number

the silo is full does not serve a purpose. On the other hand, when testing error recovery, one would want to generate further loads into a full silo.

Table 8 shows all object elements of the parameter attribute type for each object in the

Table 9: Mode Parameters in the Robot Tape Library Domain

Object	Mode	Explanation
Host Software Component	baltol	scratch redistribution level
	comp-name	HSC component for which tracing is enabled/disabled
	deferred	deferred mount processing
	dismount	on tape dismount: automatic or manual deletion of volume from control data set
	entdup	automatic or manual deletion of duplicate volume
	float	new home cell possible on tape pass through or not
	full-journal	operating mode when journal becomes full
	inittime	time interval between checks of number of scratch cartridges
	initwarn	threshold for warning when scratch pool becomes low
	maxclean	maximum number of cleans allowed
	mount-msg	whether to scroll messages on operator screen
	output	upper or lower case
	scratch	automatic or manual selection of scratch volume
	sectime	second time interval for checking when scratch pool is low
	secwarn	second warning level for low scratch pool
	viewtime	time to focus camera
	vol-watch	whether to give warning message when mounting library volume on non-library device
Automated Cartridge System	acs-scr-threshold	set threshold for scratch pool on an acs
	acs-subpool-threshold	set threshold for subpool on an acs
Scratch Pool	subpool-threshold	set threshold for tapes in scratch pool
Library Storage Module	lsm-scr-threshold	set threshold for scratch tapes in LSM
	lsm-subpool-threshold	set threshold for tapes in subpool on LSM

Table 10: State Parameters in the Robot Tape Library Domain

Object	State	Explanation
Host Software Component	autoclean	HSC automatic tape transport cleaning on or off
	gdg-sep	Unit affinity separation for GDG chains
	separation	Unit affinity separation
	service-level	basic or full
	specvol	whether transports are available when no non-library drives exist
	zeroscr	whether device selection is restricted
Cartridge Access Port	prefvlu	preference for CAP
Library Management Unit	lmu-status	up or down
Library Storage Module	lsm-status	online or offline

Table 11: Object Glossary Entry for the **LSM** Object

Object	**LSM**
Description	Library Storage Module - A single tape "silo"
Commands	`DISPLAY MODify MOVE VIew Warn`
Parameter Attribute	*lsm-id*
Mode Parameter	*lsm-subpool-threshold* *lsm-scr-threshold*
State Parameter	*lsm-status*
Nonparameter Event	*lsm-full*
Nonparameter State	

HSC-ACS. Table 9 gives a subset of possible user controllable operating modes. Table 10 lists possible types of new states for objects of the HSC-ACS that can be set by the user. Finally, Table 11 lists an object glossary entry for the LSM object with all object element entries relevant for this object.

The next step associates object elements with possible values in the domain. These values may have to be restricted further for particular system configurations and setups for the software under test. A glossary stores detailed information about each object element. An automated test generation tool must know the range of values for each element, the representation of each object element, and the default set of values for each object element. This information is needed for parameter value selection during test case generation. Table 12 shows representative entries from the Object Element Glossary for the StorageTek HSC command language. For a complete object element glossary refer to [29].

The next step in the domain analysis is to show relationships between the objects. These

Table 12: Entries from the HSC-ACS Object Element Glossary

Element Name		
lsm-id		
	Full Name	Library Storage Module (LSM) Identifier
	Definition	Names an Instance of an LSM within an ACS
	Type	parameter attribute
	Values	000. . .FFF
	Object	LSM
	Representation	Range
maxclean		
	Definition	Number of times a cleaning cartridge is used before ejecting
	Type	mode parameter
	Values	10. . .100
	Object	HSC
	Representation	Range
lmu-status		
	Definition	Status of the Library Management Unit (LMU)
	Type	state parameter
	Values	UP | DOWN
	Object	LMU
	Representation	Enumeration
journal-full		
	Definition	A dynamic event that results when the system journals become full
	Type	nonparameter event
	Values	NOT-FULL | FULL
	Object	HSC
	Representation	Enumeration
drive-status		
	Definition	Status of a tape transport (tape drive)
	Type	nonparameter state
	Values	BUSY | AVAILABLE
	Object	Tape Transport
	Representation	Enumeration

relationships are captured in an *object hierarchy*. The relationships take the form of a structural, or “part-of”, hierarchy because the structural relationships indicate parts of the objects [10]. For test generation, the main interest in object relationships is in how related objects affect possible values for their object elements. This is captured with rules about object element values. The rules take the form of object element constraint rules where the choice of one object element value constrains the choices for another. The StorageTek HSC-ACS domain provides a good example for constructing an object hierarchy (Figure 4 shows the complete Object Hierarchy for this application with objects (bubbles), object elements (inside the bubble), relationships and constraints (arrows)). [2] Consider the ACS object. Each

[2]The figure lists object elements for all but the highest level HSC object because these did not fit nicely into the dia-

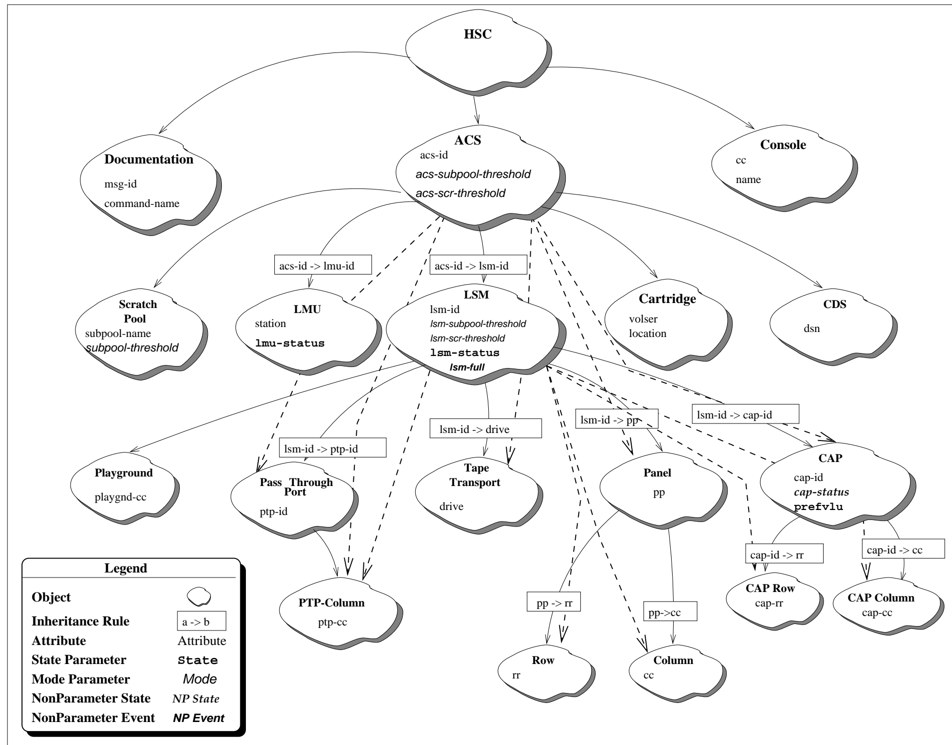

Figure 4: StorageTek Object Hierarchy

ACS supports up to sixteen LSMs, and this *structural* relationship is shown in the figure as an arrow from the ACS object to the LSM object. Each LSM contains panels, tape drives, cartridge access ports, etc. Arrows from the LSM to each object denote this structure. Annotations on the arcs denote parameter constraint rules. For instance, the choice of an ACS (i.e., a specific *acs-id* value) constrains choices for the LSM (i.e., possible *lsm-id* values). The specific choices depend on the physical configuration (e. g. the number of LSMs in an ACS and how they are interconnected). Figure 4 shows that this domain has 10 such inheritance rules. The exact specification of the rule can depend on the specific system configuration and parameter value syntax.

Table 13 contrasts the general rules of figure 4 with the specific rules used for an actual physical configuration of a robot tape library used by the system testing group at StorageTek to test HSC version 1.2. Figure 5 shows the hardware configuration of the HSC described in Table 13 (except for panels, rows, and columns to avoid clutter in the figure).

gram. HSC has 26 object elements. The majority are *mode parameters*. HSC *modes* set various operating modes of the HSC software.

Table 13: Example Parameter Inheritance Rules for Robot Tape Library

General rule	Domain Values	Configuration-specific rule	Configuration-specific values
acs-id → lmu-id	ACS={00..FF} LMU={000..FFF}	None	acs-id∈{00,01} lmu-id∈{0D0,0D1,0CC, 0CD,0CE,0CF}
acs-id → lsm-id	LSM={000..FFF}	acs-id=00 → lsm-id ∈ {000,001} acs-id=01 → lsm-id ∈ {010}	lsm-id∈{000,001,010}
lsm-id → ptp-id	PTP={0..7}	lsm-id=000 → ptp-id ∈ {0} lsm-id=001 → ptp-id ∈ {0}	ptp-id∈{0}
lsm-id → drive	DRIVE={000..FFF}	lsm-id=000→drive∈ {A10,A14,A20,A25,A29,A2C} lsm-id=001→drive∈ {A10,A14,A20,A25,A29,A2C} lsm-id=010→drive∈ {A17,A32,A36,A2F}	drive∈{A10,A14,A17,A20,A25, A29,A2C,A2F,A32,A36}
lsm-id → pp	PP={00..19}	lsm-id=000 → pp ∈ {04,07,10,14,18} lsm-id=001 → pp ∈ {00,05,10,15,19} lsm-id=010 → pp ∈ {02,08,10,12,15}	pp∈{00..19}
lsm-id → cap-id	CAP={000..FFF}	lsm-id=000 → cap-id ∈ {000} lsm-id=001 → cap-id ∈ {001} lsm-id=010 → cap-id ∈ {010}	capid∈{000,001,010}
pp → rr	RR={00..14}	pp∈{00..04} → rr ∈ {00..04} pp∈{05..10} → rr ∈ {05..09} pp∈{11..19} → rr ∈ {10..14}	rr={00..14}
pp → cc	CC={00..19}	pp∈{00..04} → cc ∈ {00..04} pp∈{05..10} → cc ∈ {05..09} pp∈{11..19} → cc ∈ {10..14}	cc={00..14}
cap-id → cap-rr	CAP-RR={00,01}	None	cap-rr∈{00,01}
cap-id → cap-cc	CAP-CC={00..06}	None	cap-cc∈{00..06}

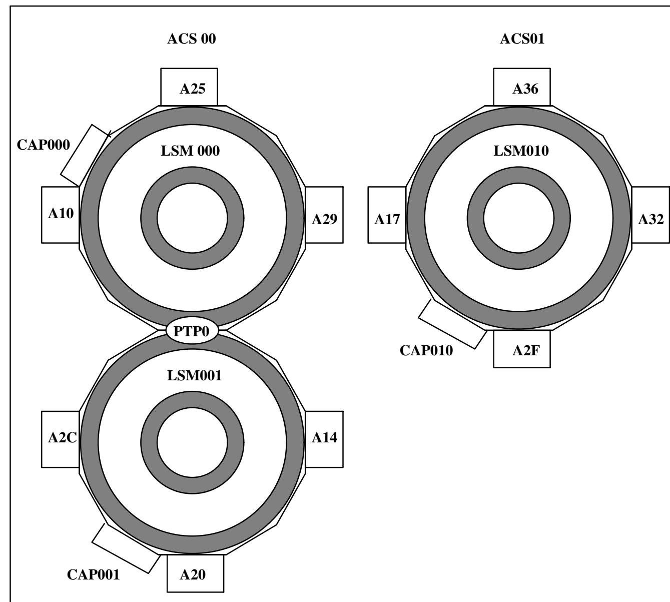

Figure 5: Hardware Configuration for the HSC described in Table 13

In some cases no rule was necessary, because the physical configuration allowed all values, or the architecture was the same (e. g. same layout of cartridge access port columns and rows for both cartridge access ports). The rightmost column gives the values used for

the constrained object element of each rule. These values reflect subsets of all potential values by taking into account the set-up of the test environment (in this case the system was tested using actual hardware rather than a simulator and the values reflect this installation).

Next, domain analysis identifies actions on objects. These actions usually manifest themselves in commands of a command language for the application. We usually start by asking *What can one do with this object? Which other objects are involved? What are desirable actions for this application domain?* For example, one would want to *mount* `tapes` into a `tape drive` of a specific `silo`, *move* `tapes` between `silos`, etc. This provides a list of possible actions and the objects involved in them. Table 14 provides such a list for the HSC-ACS domain. For use in subsequent references, the left column gives the name of the action as it is specified in the HSC release 1.2. command language. [3].

An action may only need some of the object elements associated with an object, so it is important to identify which of these object elements are necessary. Table 15 gives an example of the object elements associated with some of the HSC-ACS actions. This step provides object glossaries and a set of actions with object elements needed to perform the action. Table 11 shows an example of an object glossary entry for the LSM object. Table 16 illustrates an action table entry using the `CLean` action. Similar actions can be grouped and collectively referred to as *action classes*. For example, Table 17 groups all commands that perform setup operations for the HSC-ACS into a `Set-Up` class, all commands that inform of status and display information into a `Display` class. Actions that perform useful work are members of the `Action` class. Classes need not be disjoint. The `Any` class contains all commands.

The next step is to look at each individual action and determine whether the object elements associated with it have to satisfy value constraints for sensible operation. An example of such a rule is "don't copy a file to itself" which requires the names of the files

[3] The upper case letters in a command name indicate required syntax, the lower case letters are optional; e. g. `CL` or `CLEAN` denote the same command

Table 14: HSC Release 1.2 Command Descriptions

Command Name	Description
ALLOC	Changes the Host Software Component (HSC) allocation options.
CAPPref	Assigns a preference value to one or more cartridge access ports (CAPs)
CDs	Enable / Disable copies of the control data set
CLean	Schedules the cleaning cartridge to be mounted on a library controlled transport
DISMount	Directs the Library Storage Module (LSM) to dismount a cartridge
DRAin	Terminates and ENter command
EJect	Directs the robot to take cartridges from a Library Storage Module (LSM) and places them into a cartridge access port (CAP) where they can be removed by an operator
ENter	Used to place cartridges into a Library Storage Module (LSM) through a cartridge access port (CAP) while operating in automatic model
Journal	Used to establish the action taken by the Host Software Component (HSC) if both journals fill to capacity before a control data set backup or a journal off-load is executed
LOad	Used to query the status of the current tape transport activity
MNTD	Set options on how the Host Software Component (HSC) processes the mounting and dismounting of library volumes
MODify	Places a Library Storage Module (LSM) online or offline to all hosts
MONITOR	Initiates monitoring of cartridge move requests from the programmatic interface
Mount	Directs the robot to mount a volume onto a specified library controlled transport
MOVe	Directs the robot to move cartridges to selected destinations within the same Library Storage Module (LSM) or to any LSM within an Automated Cartridge System (ACS)
OPTion	Used to set or change general purpose options of the HSC
RECover	Allows the operator to recover the resources owned by a host that becomes inoperable
RELease	Used to free an allocated cartridge access port (CAP)
RETry	Applies only to the JES3 environment. It enables the user to restart HSC/JES3 initialization without restarting the HSC address space component
SCRparm	Dynamically modifies the scratch warning thresholds and interval values for the host on which the command is issued
SENter	Used to schedule the enter of a single cartridge using a cartridge access port (CAP) that is currently allocated for ejecting cartridges
SET	Used to activate / deactivate various functions within the HSC
SRVlev	Used to specify the service level at which the Host Software Component (HSC) operates
STOPMN	Terminates the monitoring of cartridge move requests received from the programmatic interface
SWitch	Used in dual Library Management Unit (LMU) configuration to reverse the roles of the master and standby LMUs
TRace	Enables / Disables tracing of events for selected Host Software Components (HSCs)
UEXIT	Permits you to invoke your own processing routines at particular points during HSC processing
Vary	Places physical Library Management Unit (LMU) stations online, offline, or standby
VIew	If video monitors are attached to the LSM, the VIew command enables the operator to visually inspect internal components of the LSM using the robot's cameras
Warn	Used to establish the scratch warning threshold values

in the copy command to be different. These types of rules are called *intracommand rules*. An intracommand rule specifies constraints between object element values that hold during the execution of an individual action. Constraints that are currently possible have the form `precondition` → `constraint` or (no precondition) `constraint`. A `precondition` can be an equality or inequality constraint of two or more object elements associated with an action. If the precondition is true, the constraint must hold. A `constraint` states value constraints for the affected object elements in terms of equalities or inequalities. While a

Table 15: Object elements associated with selected commands

Command Name	Object Elements
ALLOC	acs-id
CAPPref	Host-id, acs-id
CDS	dsn
CLean	drive-id, host-id
Dismount	volser, drive-id, host-id
Display	acs-id, command-name, host-id, lsm-id, console-name, subpool-name, volser
Drain	cap-id
Eject	volser, cap-id, acs-id, subpool-name, vol-count
Enter	acs-id, cap-id
Modify	lsm-id
monitor	console-name
Mount	volser, drive-id, host-id, subpoolname
Move	lsm-id, pp, rr, cc, volser
Recover	host-id
Release	cap-id
Scrparm	initwarn, inittime, secwarn, sectime, baltol
Senter	cap-id
Vary	lmu-id

Table 16: Action Table Entry for CLEAN

Command name	CLEAN
Objects	`drive, host`
Object elements	*drive-id* *drive-range* *drive-list* *host-id*
Intra-command rule	none

Table 17: Script Classes for the **StorageTek** HSC Domain

Script Class	Commands					
Any	Alloc	Commpath	Eject	Mntd	Move	Retry	Srvlev
	Cappref	Dismount	Enter	Modify	Option	Scrparm	Switch
	Cds	Display	Journal	Monitor	Recover	Senter	Trace
	Clean	Drain	Load	Mount	Release	Set	Uexit
Mode	Cappref	Clean	Mntd	Option	Set	Trace	Warn
	Cds	Journal	Monitor	Scrparm	Stopmn	Uexit	
Set-Up	Alloc	Journal	Option	Srvlev	Trace	Cappref	Mntd
	Scrparm	Stopmn	Uexit	Commpath	Modify	Set	Switch
	Vary						
Action	Alloc	Display	Enter	Move	Retry	Commpath	Drain
	Load	Recover	Senter	Dismount	Eject	Mount	Release
	View						

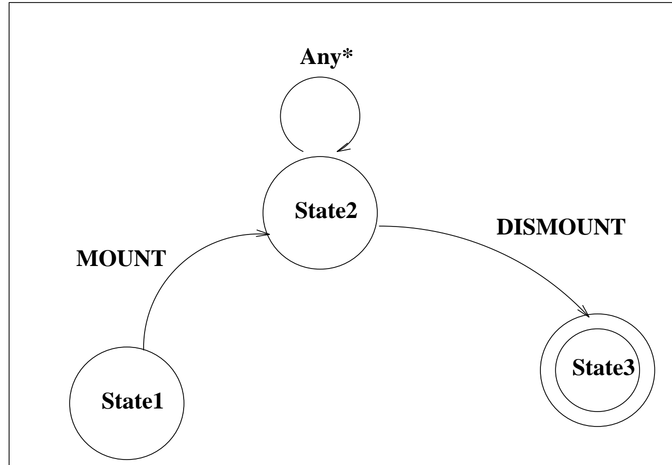

Figure 6: State Transition Diagram: MOUNT-DISMOUNT Script Rule

richer set of constraint operators is theoretically possible (relational operators to describe possible relationships between object element values), we have yet to encounter a need for them in practice.

The HSC-ACS domain only has one intra-command rule (for the `MOVE` command). It prohibits moving a series of tapes within the same panel. It is written as `(Flsm=Tlsm)` $\rightarrow$ `Fpanel` $\neq$ `Tpanel`, meaning if source LSM (`Flsm`) and target LSM (`TLSM`) are the same, then the source panel (`Fpanel`) and target panel (`Tpanel`) must be different.

The next element of analysis concerns dynamic behavior rules for sensible application behavior in the domain. *Preconditions* identify the conditions that must hold before a command representing an action can execute. *Postconditions* list the conditions that are true after it executes. Preconditions define required state or mode for an action to be meaningful. For example, a specific tape must have been entered into a tape silo before it can be moved around or removed. They also may further constrain valid parameter values through parameter binding. Postconditions state effects on object elements and influence future action sequences or parameter value selection. We represent these conditions in the form of script or behavior rules and parameter binding rules.

Scripts can be visualized as state transition diagrams (Figure 6) or regular expressions. For example, the script rule in figure 6 could be represented as `[MOUNT Any* DISMOUNT]`.

Table 18: Script Rule: Parameter Value Selection

Rule	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 19: Example Script Rules with Parameter Binding

Command Name	Script Rule
Mount/Dismount	MOUNT [tape-id*] [drive-id*] <n/any> DISMOUNT [tape-id] [drive-id]
Enter/Drain	ENTER [cap-id*] <n/any> DRAIN [cap-id]

Once action sequencing has been defined, the sequence can be annotated with value selection rules for parameters as shown in Table 18. The first rule, p*, states that the value for parameter p can be selected from any valid choice as long as it fulfills inheritance constraint rules. 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. To illustrate, the MOUNT - DISMOUNT sequence is annotated with script parameter selection rules.

```
MOUNT tape-id* drive-id*
Any*
DISMOUNT tape-id drive-id
```

This rule states that the *tape-id* and *drive-id* values can be selected from any valid choice for the MOUNT action while the DISMOUNT action must use the previously bound value for the *tape-id* and the *drive-id* parameters. (The tape mounted in a drive must be dismounted from the same drive).

Table 19 shows script rules for the HSC-ACS domain with parameter binding. [4] At this point in the analysis, we still have not defined any syntax. Except for parameter value restriction in inheritance rules, all information is independent of the actual command language chosen. This makes it possible to associate different related or competing products with the

[4] For practical reasons we let testers set n so as to give them more control in how many commands can be generated as a maximum between the two required commands.

current domain models by mapping object elements to command language parameters, actions to commands, and object element value sets to parameter value sets, and to identify to which degree the domain rules for object element values, action rules, and behavior rules manifest themselves in the command language. For test generation, we use the syntax of the command language as well as the domain model that represents the semantics of the application. The advantages of building such a domain model early in the development are: (1) The model can be reviewed early in the life cycle; (2) The model can guide command language development; and (3) One can complete the model further so that it will be useful for development purposes. Thus testers and developers will know early on what will be tested.

4.2 Reverse Engineering a Domain Model

The use of a domain model for testing is not restricted to software that was built using domain analysis or reusable components. Frequently, system testers need to test a system against a user manual. In this case the command syntax is given and they reverse engineer the application domain model for testing purposes. Figure 7 shows the process by which a domain is captured. This process is represented in the navigator utility of the domain management system component of *Sleuth* ([41]), a tool that supports application domain based testing. The following explanation refers to the navigator utilities in Figure 6 in parentheses to explain the relationship between domain capture and the navigator tool.

We start with the syntax of the language (SDE), then extract parameters (OEE) and group them into objects (OTE). Each parameter in the command language is categorized according to the object it influences. This classification provides a first cut of the objects and their properties. To illustrate this process, consider Figure 8. Two HSC commands from the robot tape library have parameters that relate to three domain objects **Cartridge**, **Tape Transport**, and **HSC**. This groups parameters by related object. For a complete grouping see

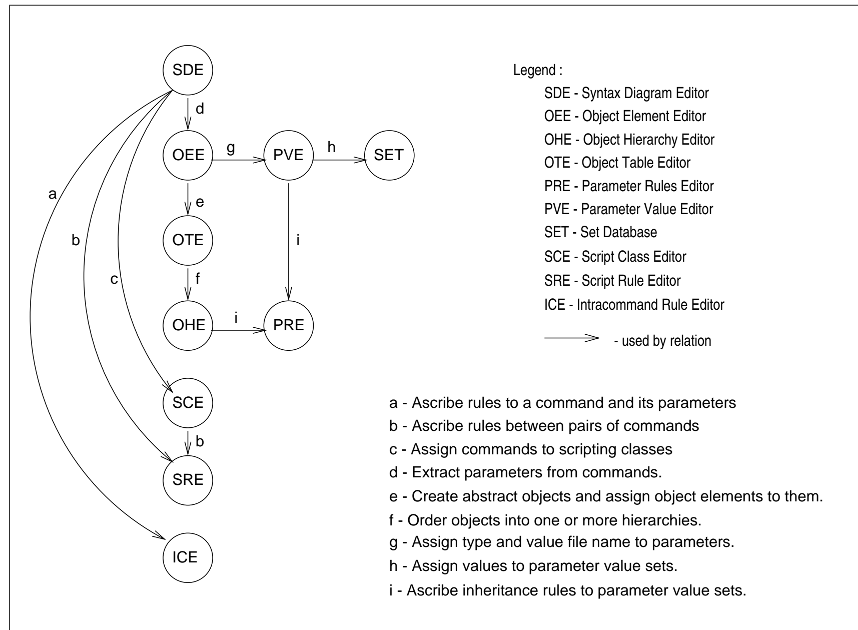

Figure 7: Process for Reverse Engineering a Domain Model

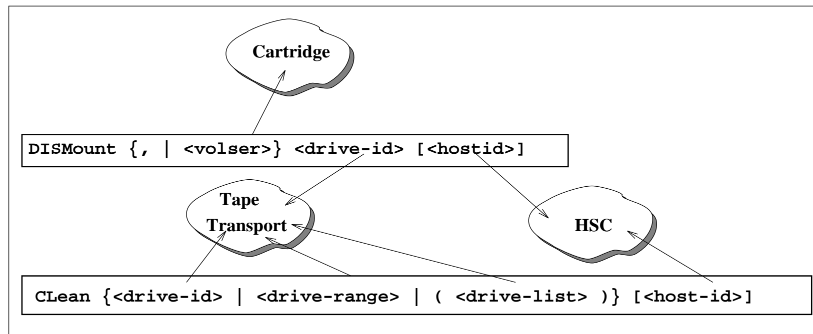

Figure 8: HSC Object and Object Element Analysis

Figure 4 in which each bubble represents an object and the parameters in the bubble its object elements. The object elements for the HSC object are mostly mode parameters and are listed separately in Table 9.

Next, each object element is classified by defining its *object element type*. We also have to identify possible parameter values for each object element (PVE). In *Sleuth*, objects consist of object elements and relate to each other. Relationships between objects produce the object hierarchy (OHE) (cf. Figure 4). Arcs in the hierarchy are annotated with parameter constraint rules (PRE). Next we identify whether parameters of single commands constrain

each other (ICE) (refer back to the rule for the `MOVE` command discussed earlier). Last, we determine scripting rules and parameter binding rules (SRE). They are equivalent to those found in the apriori domain analysis (cf. Table 19).

Sleuth also allows sets of commands to be grouped together (SCE) as action classes (e. g. Action versus Display types of commands). Scripting rules may involve such classes of commands. The steps are the same as in the a priori analysis. Results for the HSC-ACS are in Table 17. This completes the domain model. Again, we have a language dependent and a language independent part of the model.

5 Test Generation Process

The domain model of Section 4 serves as an abstract representation of the system under test. To make the domain model useful requires a process to automatically generate test cases based on that abstraction. The process must also consider testing strategies and test criteria used during system test, and couple domain analysis and the domain model with test generation. We base this test generation process on the results of observing a team of system testers in industry. We investigated (1) How they tested their products, (2) The steps in their testing process, (3) Needs for test automation, and (4) Desirable features of an automated test generator.

Figure 9 shows the input and output for each step in the test generation process. The domain model D_0^v captures the syntax and semantics of version v of the system under test. The zero subscript identifies the domain model as the starting point from which all other models are derived (for example those representing a competing software product by a different manufacturer). For instance, $D_0^{HSC1.2}$ denotes the StorageTek HSC Release 1.2 domain model.

The domain model is a persistent description of the software. It represents the default description from which test suites are generated. A domain model is needed for each new domain and every time a domain changes significantly. All testers share the domain model

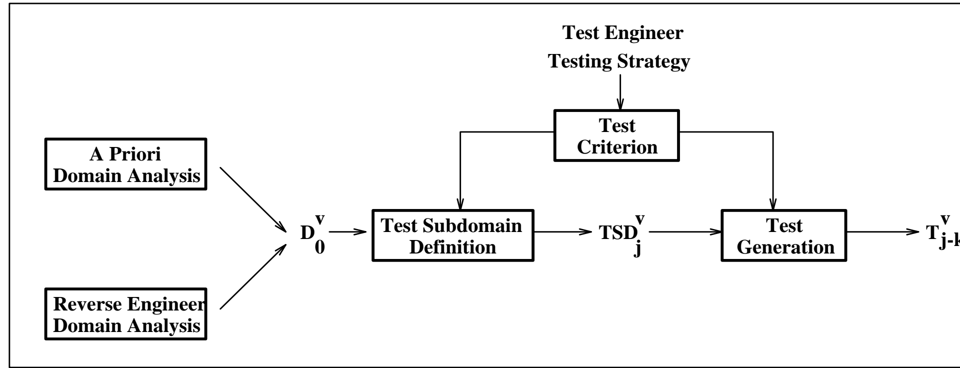

Figure 9: Test Generation Process Model

to provide a consistent view of the system under test. Sometimes test objectives call for test cases generated directly from D_0^v. Such tests represent "valid" sequences of commands that follow all syntax and rules defined in D_0^v. Currently, *Sleuth* captures the domain model using a reverse engineering process. Figure 7 shows the utilities which correspond to the domain capture activities.

Often, testers want to test a particular system configuration or a particular feature. To achieve this, they build a *test subdomain* TSD_j^v. The subscript j identifies the specific subdomain created, and the superscript identifies the version of the system under test. Test subdomains may be a subset or a superset of the original domain model. A subset restricts the parameters and commands generated in a test case. For example, $TSD_{CAP}^{HSC1.2}$ is the Cartridge Access Port (CAP) Test Subdomain for HSC Release 1.2. In this test subdomain, only the following commands are turned on (activated and thus can be generated): `Action` class: `Drain, Eject, Enter, Release, Senter`. In addition, the set-up commands `Vary` and `SRVLEV` must be turned on to ensure that the LMU's are on-line, and the service level is FULL (a precondition to the *Action* class commands of this test subdomain).

A superset of the domain model allows greater freedom in test generation by turning rules off (script rules, intracommand rules, or parameter inheritance rules). This is important for testing error checking and recovery features of applications. For example,should one want to test erroneous use of the CAP, one can turn off all rules for *Action* and *Set-Up*

commands. Test suites generated for this test subdomain would try to eject tapes that have never been entered into the silo, and use devices that are either not on-line or not at the required service level. Test criteria also drive the creation of test subdomains. To illustrate, after changes to a system, regression testing rules [42] prescribe how to build a regression testing subdomain. In this case, the test criteria define what parts of the (modified) system must be regression tested, leading to the formation of a regression test subdomain.

Sleuth provides a set of utilities to customize domain models into test subdomains. Table 20 summarizes them.

Domain Model Component	Utility	Purpose
commands	activate, deactivate	define scope of test
script rules	activate, deactivate, edit	test valid, erroneous sequences
intracommand rules	activate, deactivate, edit	test valid, invalid single commands
syntax	edit syntax	model syntactically incorrect commands
	edit branch frequencies	control test emphasis
parameter values	edit	reduce set, add new (invalid) values
parameter inheritance rules	activate, deactivate, edit	test valid versus invalid operation

Table 20: Utilities to Build Test Subdomain

Test Generation takes information from the test subdomain and directives from the tester to generate test suites, $T^{v}_{j-k}(k = 1, 2, 3, \ldots,)$. For instance, $T^{HSC1.2}_{CAP-10}$ denotes test #10 generated from the CAP subdomain for HSC Release 1.2. Test directives include

- generate `n` commands of class or type `c`: This is the basic test generation directive. `n` is the number of commands to generate, `c` is either the name of a command (like `Move` or `Drain`), or the name of a script class (like `Action` in Table 17).
- recall archived test suites: This enables use of legacy test suites, or tests generated by other means. It makes using *Sleuth* flexible, particularly with regards to other tools.
- merge several test suites: This is used to simulate parallel requests from various sources where appropriate.

A test suite is the result of the test generation. It contains test scripts, test templates, and test cases. Test scripts are lists of command names. Test templates are lists of commands

with place holders for parameters. A test case is a list of executable commands. Test scripts, test templates and test cases are generated in three phases. This decision was made to allow reuse of tests at various levels of specificity. In the first phase, test directives are interpreted and the test script is generated taking scripting rules into consideration. The second stage creates a command template by selecting a specific instance of each command in the script. Parameters remain as place holders. The third phase uses parameter binding rules, intra-command rules, and parameter inheritance constraints to create a fully parameterized list of executable commands.

In the development of the test generation module of *Sleuth*, we experimented with a variety of generation approaches, including context free grammars [32], attribute grammars [15, 40], probabilistic grammars [27, 28], and AI Planning Systems [21]. All of these have their advantages and disadvantages. Our selection criteria were that the methods selected should

- show adequate performance for industrial use (scale up). This disqualified the AI technique for the moment. We are, however, working on improving its performance, since a pilot study [21] showed that the Planner generated some innovative tests to achieve test goals.
- make changes to the domain model easy to deal with when testing subdomains are formed. Unlike some other sentence generation problems, our test generation problem encounters frequently changing grammars, if grammars are used. Those changes should not require the user to manipulate the grammar rather, the test tool should perform this. The initial version of *Sleuth* [40] was based on attribute grammars. Empirical evidence suggested that while automated algorithms to percolate domain model changes into the grammar are, of course feasible, they were needlessly complex. The same was true for probabilistic grammars, and, we suspect, for some of the other approaches discussed in section 2.

As a result, we settled on a combination approach with different generation mechanisms for the scripting, template, and parameter value phases of test generation:

- Phase 1: Commands are randomly selected from the set of currently allowable commands (those that are part of the current testing subdomain and possible with respect to scripting rules). The precise distribution can be influenced by the user to model operational profiles. Scripting rules are activated when a command is selected for generation that is part of a scripting rule.
- Phase 2: This phase generates a command template for each command chosen in phase 1. Command templates can be thought of as one path through a command's syntax diagram. For every branch point in the syntax diagram, the generator makes a choice dictated by user definable frequencies (default: uniform).
- Phase 3: This stage selects the actual parameter values for each command. The test generator must fulfill all constraints: parameter binding from the scripting phase, intracommand rules, and parameter inheritance rules. The generator uses set operations (on parameter values) to compute the possible values at a particular point in the generation and then selects one of the values. If the set is empty, the generator chooses from a user defined alternate set. Users often choose this set to contain a single value of '?' to alert them that their domain model has a fault, because the constraint was not satisfiable. A generation log provides information on how a particular set of values was chosen.

6 *SLEUTH* Test Generation Example

Sleuth supports all stages of application domain based testing. It provides utilities to capture the domain model through the reverse engineering process described in section 3.2. Figure 7 identifies the utilities for the domain capture activities. The *Specification* pull-down menu

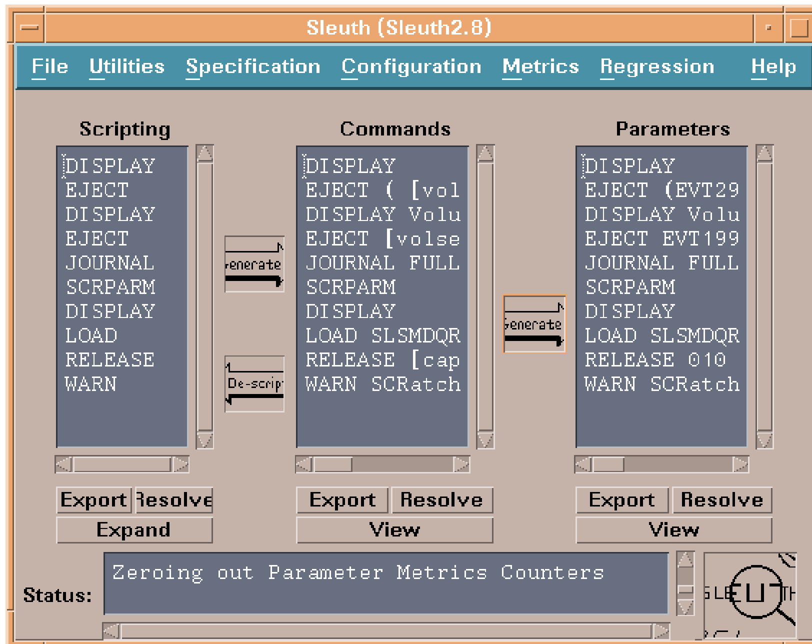

Figure 10: Sleuth Main Window

in Figure 10 provides access to these utilities. The *Configuration* pull-down menu accesses the utilities of Table 20 to build test subdomains.

Sleuth acts as a test tool generation engine in that it uses the domain model to build a customized test generator for that domain. Every domain model thus results in a different test generator, making *Sleuth* capable of handling a wide range of application domains.

Test generation uses the current test subdomain (the default is the full domain model) as the basis for generating tests. Test generation follows the three stage approach described earlier. Figure 10 represents these three stages in the three panels of the main window labeled “scripting”, “commands”, and “parameters”. The scripting stage generates a list of command names. The user gives *Sleuth* high level test generation directives such as “generate 100 MOVE commands” (`@100\MOVE`). The scripting rules cause *Sleuth* to generate the necessary additional commands to make the 100 MOVE commands meaningful (*e. g.*, system setup, entering of enough tapes). The tester does not have to worry about this since the domain model ensures generation of command sequences that represent valid system operation. The tester saves the list of commands (the script) using the export button. The

generate button starts the second phase of test generation.

Table 21 shows results of generating 20 commands in the first stage using the StorageTek $D_0^{HSC1.2}$ domain model. Command #1, SRVLEV, ensures the correct service level before the test begins. ENTER is used to insert tapes into the ACS through the CAP door. This ensures a known set of tapes. In the first stage, *Sleuth* applied all scripting rules. For instance, the ENTER command requires a corresponding DRAIN command. Command #2 issues an ENTER command and *Sleuth* generated an appropriate DRAIN (command #20). Similarly, tape mounts and dismounts must be sequenced because tapes must be mounted before dismounting. *Sleuth* generated two MOUNT-DISMOUNT sequences in commands #6-#15 and #9-#18.

Table 21: *Sleuth* Stage 1 Test Generation

Line Num	Command Name	Description
1	SRVLEV	System Set Up
2	ENTER	Enter Tapes for the Test
3	RECOVER	
4	MONITOR	
5	JOURNAL	
6	MOUNT	Mount First Tape
7	CLEAN	
8	CAPPREF	
9	MOUNT	Mount Second Tape
10	UEXIT	
11	MOVE	Move tapes inside the ACS
12	SWITCH	
13	JOURNAL	
14	SCRPARM	
15	DISMOUNT	Dismount Second Tape
16	DISPLAY	
17	LOAD	
18	DISMOUNT	Dismount First Tape
19	EJECT	
20	DRAIN	

The command template stage takes the list of commands and creates a command template for each command by selecting a path through the command's syntax diagram.[5] To archive the suite of command templates, the tester clicks on the export button underneath

[5] *Sleuth* allows command syntax to be represented and edited as syntax diagrams. Branches are associated with selection frequencies

Table 22: *Sleuth* Stage 2 Test Generation

Line Num	Command Template
1	SRVLEV FULL
2	ENTER [cap-id] SCRatch
3	RECOVER [host-id]
4	MONITOR PGMI ,L= [console-name]
5	JOURNAL FULL = ABEND
6	MOUNT [drive] SUBpool ([subpool-id])
7	CLEAN [drive-id] [host-id]
8	CAPPREF [prefvlue] 001
9	MOUNT [volser] [drive]
10	UEXIT [nn-id] Load =LSLUX [uexit-id] ,Enabled
11	MOVE Flsm([lsm-id]) Panel([pp]) Row([rr-ls]) TLsm([lsm-ls]) TPanel([pp])
12	SWITCH Acs [acs-id]
13	JOURNAL FULL = Continue
14	SCRPARM
15	DISMOUNT [volser-id] [drive-id] [host-id]
16	DISPLAY COMMPath
17	LOAD SLSMDQR
18	DISMOUNT [drive-id] [host-id]
19	EJECT VOLSER ([volser-ls])
20	DRAIN [cap-id]

the command panel. Clicking on the generate button between the command and parameter panel (cf. Figure 10) starts the last phase of test generation. Table 22 shows the results of generating command templates for the commands in Table 21. Parameters are shown as place holders using square brackets.

In the third phase, parameter values are selected based on the current system configuration taking into account parameter inheritance rules, intracommand rules, and parameter binding. The tester saves the command suite by clicking on the export button. Table 23 shows the final test case for our example. Note that CAP `000` used to enter tapes in command #2 is released by `DRAIN` in command #20. Likewise, the `MOUNT-DISMOUNT` command sequences of commands #6-#15 and #9-#18 select the appropriate tape drives and tape id. For command #11, *Sleuth* applied the intra-command rule for the `MOVE` command mentioned earlier.

Sleuth can be used for functional testing of specific commands (the test subdomain includes the specific command to be tested and possibly other commands required due to scripting rules). Frequencies ensure that all syntactic options in a command are exercized,

Table 23: *Sleuth* Stage 3 Test Generation

Line Num	Commands
1	SRVLEV FULL
2	ENTER 000 SCRatch
3	RECOVER MVSH
4	MONITOR PGMI ,L=MVSH
5	JOURNAL FULL=ABEND
6	MOUNT A2F SUBpool (EVT180)
7	CLEAN A10 MVSE
8	CAPPREF 9 001
9	MOUNT EVT280 A14
10	UEXIT 10 Load=LSLUX 11 ,Enabled
11	MOVE Flsm(001) Panel(00) Row(02,03,04) TLsm(001) TPanel(15)
12	SWITCH Acs 00
13	JOURNAL FULL=Continue
14	SCRPARM
15	DISMOUNT EVT180 A2F MVSE
16	DISPLAY COMMPath
17	LOAD SLSMDQR
18	DISMOUNT A14 MVSE
19	EJECT VOLSER (EVT180,EVT280)
20	DRAIN 000

provided that enough instances of the command are generated. The user can also define frequencies such that test generation is biased towards specific options in a command (e. g. it may be desirable during regression testing to test modified parts of a command more often than unchanged parts).

The metrics pull-down menu provides access to domain model usage information, i. e. how often which part of the domain model was used during test generation. This information is analogous to a coverage analyzer for white-box testing. The measures report on how often commands, rules, branches, and values have been used during test generation.

System testing is supported by using the complete domain model and instructing *Sleuth* to generate a certain number of commands (we have seen system testers generate hundreds to thousands of commands on Monday morning and execute and evaluate the results for the remainder of the week). The domain model makes sure that dynamic behavior gets tested realistically. User controlled frequencies force a desired operational profile. Error recovery in the midst of regular operation can be tested by defining a testing subdomain through intentionally breaking scripting rules, intracommand rules, or parameter inheritance rules,

or by defining invalid parameter values, thus forcing erroneous behavior.

In our empirical observations of system testers over the last four years, we have seen that testers restrict the domain model to a testing subdomain for a variety of reasons: test focus (e. g. testing a new type of tape drive controlled by the HSC), exclusion of commands that are undesirable for some reason (e. g. when a tester wants to run tests over night, excluding commands that would require (human) operator intervention is desirable), testing a specific type of workload, etc.

7 Reuse Scenarios and Experiences

Testing based on an application domain model offers two types of reuse capabilities: Domain Reuse and Test Case Reuse. Domain Reuse refers to the amount of domain reuse possible when testing different systems in closely related or overlapping application domains. Test case reuse relates to the amount of reuse for a test case under various circumstances.

7.1 Domain Reuse

Complete Domain Reuse occurs when testers use an existing domain model with no change or modification to generate test cases for a new system or release. For example, IBM released MVS 5.1.1, a new version of the MVS operating system. StorageTek used the $D_0^{HSC2.0.1}$ domain model and its associated test subdomains to test HSC's operation with the new release. No changes were made to the domain model. Testers saved significant time by recalling and generating test cases very quickly.

Partial Domain Reuse can be classified as to which portions of the domain model have to change and as to the extent of such change. Reuse is greater when fewer parts of the model require changes and the extent of changes is small. Syntax and parameter value sets are most prone to change. They are the least general to a domain and depend on an actual application. Partial domain reuse was found in three test situations: (1) Testing Competing Systems, (2) Test Varying System Configurations, and (3) Testing Successive System Releases.

Testing competing systems can reuse all parts of a domain model except for syntax and possibly parameter value files. An example of such a situation are two competing tape storage systems, StorageTek's HSC and IBM's 3594 and 3495 tape library systems. Testers used the $D_0^{HSC2.0.1}$ domain model as a basis for creating the $D_0^{IBM3594}$ domain model. Since their physical configurations vary, most changes to the domain model occurred in the parameter value definitions and parameter inheritance rules. On the other hand, the command syntax of the two systems had a large intersection. Therefore, the testers were able to reuse many of the command syntax descriptions from the HSC domain.

Second, we built a domain model for testing ACSLS (the UNIX environment version of HSC) running on competing workstations, Sun Microsystem Sparc and Hewlett Packard 9000. The ACSLS software was derived from the HSC system. Therefore, the $D_0^{HSC2.X}$ domain model was reused to create the $D_0^{ACSLS-X}$ domain models. The functionality of the IBM and the Unix systems are the same, but the command language and some of the parameter value files are different due to the variations in naming conventions on those platforms. In this example, the changes to the parameter definitions and the command syntax are minimal so domain models $D_0^{ACSLS-SunOS}$ and $D_0^{ACSLS-HPUX}$ are almost identical.

One can also use the domain model to identify to which degree competing systems have the same features (which portions of the domain model are needed in both systems; we compare objects, object elements and actions).

Testing Varying System Configurations of the same version of a system is a frequent testing task. At the same time, it only requires modifying the parameter value files and some parameter inheritance constraint rules if they relate to configuration–relevant values. The HSC provides an example for such a situation. A specific tape silo configuration (ACS) can connect one to sixteen silos. These configurations influence the LSM, tape transport, pass through port, panel, cell row/column, and cartridge access port objects. Specifically, they affect which values are valid for LSM identifiers. Parameter inheritance constraints also change, because the specific connections between silos affect what panels, rows and

columns are available for tape storage. To test the wide variety of system configurations, testers would load the appropriate domain model and modify it for each configuration. For example, $TSD_{Single}^{HSC2.0.1}$ defines a test subdomain for a single silo configuration for the HSC 2.0.1 domain. $TSD_{Three}^{HSC2.0.1}$ defines a three LSM configuration.

Testing Successive System Releases provides another opportunity for domain model reuse. The domain model for the previous release is used as a basis for the new domain model. We reused domain models for successive releases of HSC 1.2, 2.0, and 2.0.1. The first step compared command syntax diagrams for HSC 1.2 and HSC 2.0. Twenty-six commands required modifications to the syntax diagrams in the form of new keywords and new branches or command options. Similarly, the tester identified and updated the commands that changed between HSC 2.0 and 2.0.1. Five commands required keyword and option updates. It was also necessary to update the parameter value files for the hardware configurations. The tester changed the parameter files for the following parameters (cf. Table 8): *lsm-id, acs-id, drive, dsn, volser, station, cap-id, host-id* and *subpool-name*.

To illustrate the nature of changes to a domain model between versions in more detail, consider one of the key enhancements between HSC1.2 and HSC2.0. StorageTek modified the ACS hardware to support multiple Cartridge Access Ports (CAP). This hardware modification initiated changes to the command line interface, HSC2.0. Specifically, it modified all commands in the `Action` script class that included the `cap-id` parameter (See Section 4). Modifications included adding an option to the syntax diagrams to allow either a `cap-id` or a list of `cap-id`'s.

At the parameter value level, there are changes in the `cap-id` format. The format provides backward compatibility to old CAP configurations and new functionality to the new CAP doors. HSC2.0 appends a `'':00''` or `'':01''` to each `cap-id` ({`000...FFF`}). The first extension denotes an "old" CAP while the second designates the new CAP hardware.

Table 24: Domain Reuse Example

Num	HSC 1.2	HSC 2.0
1	SRVLEV FULL	SRVLEV FULL
2		OPTION REPATH (Yes)
3		OPTION EJLimit (9999)
4	ENTER 000 SCRatch	ENTER 001:00 SCRatch
5	RECOVER MVSH	RECOVER MVSH
6	MONITOR PGMI ,L=MVSH	MONITOR PGMI ,L=MVSH
7	JOURNAL FULL=ABEND	JOURNAL FULL=ABEND
8	MOUNT A2F SUBpool (EVT180)	MOUNT A36 SUBpool (EVT2)
9	CLEAN A10 MVSE	CLEAN A2F MVSE
10	CAPPREF 9 001	CAPPREF 4 000:00
11	MOUNT EVT280 A14	MOUNT EVT297 A29
12	UEXIT 10 Load=LSLUX 11 ,Enabled	UEXIT 04 Load=LSLUX 02 ,Enabled
13	MOVE Flsm(001) Panel(00) Row(02,03,04) TLsm(001) TPanel(15)	MOVE Flsm(000) Panel(18) Row(10) TLsm(000) TPanel(14)
14	SWITCH Acs 00	SWITCH Acs 00
15	JOURNAL FULL=Continue	JOURNAL FULL=Continue
16	SCRPARM	SCRPARM
17	DISMOUNT EVT180 A2F MVSE	DISMOUNT EVT297 A29 MVSH
18	DISPLAY COMMPath	DISPLAY COMMPath
19	LOAD SLSMDQR	LOAD SLSMDQR
20	DISMOUNT A14 MVSE	DISMOUNT A36 MVSE
21	EJECT VOLSER (EVT180,EVT280)	EJECT VOLSER (EVT297)
22	DRAIN 000	DRAIN 001:00
23		EJECT SCRTCH (010:01) VOLCNT (99)

In addition, the new CAP doors can eject up to 9999 tapes in one `EJECT` command. Previously, HSC1.2 allowed a maximum of 100 tapes per `EJECT` command. This was a minor change to a parameter value file. HSC2.0 also introduced a new system set up command: `OPTION`. This command specifies `cap-id`'s to eject tapes when a CAP is unavailable. It also sets limits on how many tapes to eject. We included this command in the `SETUP` scripting class.

Table 24 compares HSC1.2 test generation with HSC2.0. We added the new command `OPTION` and set frequencies to ensure that *Sleuth* generated the updated command syntax and parameter value choices. The new `OPTION` commands in lines #2 and #3 show how new features can be added to an existing domain model when changes call for new scripting rules - in this case the new commands are part of the SETUP class. Command #3 also reflects the change in the maximum number of tapes ejected. In several commands, the new `cap-id` syntax is evident (e.g., 4,10,22, and 23).

Reuse Stage	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 25: Domain Based Testing - Reuse Applications

7.2 Test Case Reuse

Test Case Reuse is the process of recalling and using a previously archived test case. The three stage test generation offers three types of reuse possibilities: tests are reusable as scripts, suites of command templates and suites of commands. Table 25 shows a list of possible reuse applications at all three levels of test case generation. Many of these were suggested by the test engineers at StorageTek. From this list we identified three reuse scenarios: (1) Successive Software Releases, (2) Multiple System Configurations, and (3) Varying Test Case Construction. The following provides an overview of each class and describes how the test engineers at StorageTek use *Sleuth* to reuse archived tests. Even though the examples presented here are specific to the tape library, it is reasonable to assume that other systems will have similar reuse needs.

- **Scenario #1 - Successive Software Releases**

Between releases, the command language may change. New commands may be added, obsolete commands may be deleted, and command syntax may be modified. In some cases, a new release may influence rules like intra-command rules or parameter constraint rules.

At StorageTek, test engineers use HSC commands to generate tests for the robot tape library. These tests can be archived and recalled at all three stages of test case generation. For

Table 26: Test Case Reuse - Successive Software Releases

Archived Script `enter-tapes.s`	HSC 1.2 Template	HSC 2.0 Template
MODIFY	MODIFY CAP [cap-id]) ONline	MODIFY CAP [cap-ls] ONline
CAPPREF	CAPPREF [prefvlue] 000	CAPPREF [prefvlue] [cap-ls]
ENTER	ENTER [cap-id] SCRatch	ENTER 000:00 SCRatch
DRAIN	DRAIN [cap-id]	DRAIN ([cap-ls])

example, 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 requires software to be upward 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 suites for this simple form of regression testing.

This same reuse scenario can be used for testing applications on all its platforms and operating systems on which they are supposed to run. There may be many common commands with slight differences between the languages. For example, some commands may be needed for one operating system and not another. Using this same scenario, test cases can be recalled at the *Scripting* level, and new tests can be generated for various operating system versions. Since all test suites are identical at the Script level, we build uniform, comparable test suites for a variety of releases and platforms.

Table 26 shows an example of reusing the archived script `enter-tapes.s`. In the table, the script is used to generate command templates for HSC1.2 and HSC 2.0. The script could also be used for HP-UX OS, SunOS, HSC2.0.1, and IBM3594 domains. The utilization of archived scripts is typically very high. One script can generate a wide variety of command templates. In turn, each command template can generate several test cases.

• **Scenario #2 - Multiple System Configurations**

The set of valid parameter values represents a configuration of the problem domain. It represents a configuration of logical objects or physical devices.

The StorageTek robot tape library hardware can be configured in many ways. To test these configurations, testers execute tests on a Library Management Unit (LMU) simulator or on actual ACS hardware. ACS hardware can be configured in many ways. Testing on the actual hardware explores timing problems or real-time processing faults. Each ACS configuration needs a separate set of parameter value files and parameter constraint 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 different configurations. This saves generation time, but more importantly, it makes test cases uniform and comparable because they test "the same thing." Table 27 shows how a single archived command template can be reused to generate test cases for multiple hardware configurations. In this example, the number of LSM's, tape drives, and tape volumes were changed in Hardware Configuration 1 and Hardware Configuration 2.

Table 27: Test Case Reuse - Multiple System Configurations

Archived Template `mount-tapes.ct`	HSC 1.2 Configuration 1	HSC 1.2 Configuration 2
ENTER [cap-id]	ENTER 002	ENTER 011
MOUNT [drive] ([volser-id])	MOUNT A10 (EVT185)	MOUNT A2F (EVT297)
LOAD SLSLDQR	LOAD SLSLDQR	LOAD SLSLDQR
DISMOUNT [volser-id] [drive-id]	DISMOUNT EVT185 A10	DISMOUNT EVT297 A2F
DRAIN [cap-id] [cap-id]	DRAIN 002	DRAIN 011

- **Scenario #3 - Test Case Construction**

Often testers find a particular list of commands is good at detecting a certain error. Test engineers may also have a set of commands putting 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 stages of test generation. A new test case can be constructed from a set of "building blocks." One script might be included to put the system in a particular state, another might present a workload to the system, and a third might reuse a test that has been successful for a particular type of fault. Command

templates can be included to create a larger test sequence. Finally, archived test cases can be included for a regression test. To support test case construction, *Sleuth* provides a directive to *include* archived scripts, command templates, and test cases.

8 Conclusions and Further Work

We presented an approach to describe an application domain for testing purposes and showed the ways in which reuse of a domain model has been fruitful. While manual generation of test cases from such a domain model is of course possible, its benefits are much greater when the domain model is coupled with an automated test generator. *Sleuth* is a test generator that bases test generation on a model of the application domain. It has been in use at StorageTek for several years. While building a domain model is clearly an effort, the potential for domain and test case reuse was realized for the test scenarios we encountered during practical use.

How good is this method? In a recent case study [44] comparing testing with and without this method for a 12 week test cycle, we found that with domain based testing, the tester found about 3.5 times as many errors. Post-release incidence rate (tracked for 12 months after release) for the *Sleuth*-tested version was about 30% lower. This gives us great confidence that testing based on a specialized model of the application domain not only offers significant opportunities for reuse of test artifacts, but also provides an effective and efficient testing environment.

Further work is ongoing to answer a variety of questions and tester needs. The domain model currently does not represent full system state. Adding this capability is desirable as it makes the model more complete. On the other hand a fully complete model would in effect be a simulator. Just how much state information is necessary is an open question. Validating test results is still manual. We are working on adding test oracle information to the domain model. This requires new analysis steps to determine the nature of the oracle.

We also need to examine this method with existing domain analysis and domain models for software reuse [3, 4].

Currently, the application of the domain model is restricted to applications with a command or transaction language as a user interface. Extending the approach to testing systems with a graphical user interface would require mapping actions, objects, and object elements into GUI concepts like push buttons, toggles, text fields, basic or option menus, etc. as an additional component of the model. Syntactic representation describes the precise ways of activating these constructs (e. g. via Xrunner code).

Finally, we are in the process of automating test subdomain generation based on test objectives and to develop testing criteria specific to testing against an application domain model.

Application domain based testing and *Sleuth* have been in operation in an industrial system testing group for over four years. Results support its usefulness in practice, indicating that even a relatively simple model can go a long way in improving system testing.

Acknowledgements

Our research has been 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.

References

[1] Charles Anderson, Anneliese von Mayrhauser, and Rick Mraz. “On the Use of Neural Networks to Guide Software Testing Activities”, *Procs. International Test Conference*, Oct. 1995, Washington, DC.

[2] M. Balcer, W. Hasling, T. Ostrand; “Automatic Generation of Test Scripts from Formal Test Specifications”, *Procs. Third Symposium on Software Testing, Analysis, and Verification*, Dec. 1989, pp. 210 – 218.

[3] D. Batory, S. O’Malley, “The Design and Implementation of Hierarchical Software Systems With Reusable Components”, *ACM Trans on Soft Eng and Meth*, Oct 1992.

[4] D. Batory, L. Coglianese, M. Goodwin, S. Shafer, "Creating Reference Architectures: An Example from Avionics", *Symp on Soft Reusability 1995*, Seattle, Washington.

[5] J. Bauer and A. Finger. "Test Plan Generation Using Formal Grammars", *Procs. Fourth International Conference on Software Engineering*, 1979, pp. 425-432.

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

[7] Boris Beizer, *Software Testing Techniques*. VanNostrand - Reinhold, Second Edition, 1990.

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

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

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

[11] A. Celentano and S. Crespi Reghizzi and P. Della Vigna and C. Ghezzi and G. Gramata and F. Savoretti. "Compiler Testing using a Sentence Generator," *Software-Practice and Experience*, 1980:10, pp.987-918.

[12] Tsum S. Chow. "Testing Software Design Modeled by Finite State Machines," *Proceedings of the First COMPSAC*, 1977, pp. 58-64.

[13] F. L. DeRemer. "Simple LR(k) Grammars", *Communications of the ACM*, 14 (1971), p.453.

[14] P. Devanbu, D. Litman; "Plan-Based Terminological Reasoning", *Artificial Intelligence*, vol. 84, pp. 1 –35, 1996.

[15] A.G. Duncan and J.S. Hutchison, "Using Attributed Grammars to Test Designs and Implementations," *Proc of the Fifth Int'l Conf on Soft Eng*, 1981, pp. 170-177.

[16] S. Fujiwara, G. von Bochman, F. Khendek, M. Amalou, and A. Ghedamsi. "Test Selection Based on Finite State Models", *IEEE Transactions on Software Engineering SE-17*, no. 10(June 1991), pp. 591-603.

[17] J. B. Goodenough and S. L. Gerhart. "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, SE-1(2), June 1975, pp. 156-173.

[18] G. Gonenc. "A method for the design of fault-detection experiments", *IEEE Transactions on Computers*, vol. C-19, June 1970, p. 551 – 558.

[19] Dick Hamlet and Ross Taylor. "Partition Testing Does not inspire Confidence", *IEEE Transactions on Software Engineering*, SE-16(12), Dec. 1990, pp. 1402-1411.

[20] William E. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, Software Engineering and Technology Series, 1987.

[21] Adele E. Howe, Anneliese von Mayrhauser, Richard T. Mraz; "Test Case Generation as an AI Planning Problem", *Automated Software Engineering*, 4, pp. 77 – 106 (1997).

[22] J. Hutchison. *Private Communication*, 1992.

[23] D. C. Ince. "The Automatic Generation of Test Data", *Computer Journal*, vol.30(1), 1987, pp. 63-69.

[24] V. E. Kelly, M. A. Jones; "KITSS: A Knowledge-Based Translation System for Test Scenarios", *Procs. 11th National Conference on Artificial Intelligence*, Washington, DC, pp. 804 – 810.

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

[26] S. Naito, M. Tsunoyama. "Fault detection for sequential machines by transition tours", *Procs. Fault Tolerant Computing Systems*, 1981, p. 238 – 243.

[27] Peter M. Maurer; "Generating Test Data with Enhanced Context-Free Grammars", *IEEE Software*, July 1990, pp. 50 – 55.

[28] Peter M. Maurer; "The Design and Implementation of a Grammar-based Data Generator", *Software - Practice and Experience*, vol. 23(3), pp. 223 – 244 (March 1992).

[29] R. Mraz; "Automated Testing of Application Domains", PhD Dissertation, Colorado State University, Computer Science Department, December 1994.

[30] Glenford J. Myers. *The Art of Software Testing*, Wiley Series in Business Data Processing. John Wiley and Sons, 1979.

[31] T. Ostrand, M. Balcer", "The Category-partition Method for Specifying and Generating Functional Tests", *CACM*, 31(6), June 1988, pp. 676 – 686.

[32] Paul Purdom; "A Sentence Generator for Testing Parsers", *BIT*, 12 (1972), pp. 366 – 375.

[33] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. "Approaches to Specification-Based Testing", *Procs. ACM Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, December 1993, pp. 86-96.

[34] K. K. Sabnani, A. T. Dahbura. "A protocol testing procedure", *Computer Networks and ISDN Systems*, vol. 14, no. 4, 1988, p. 285 – 297.

[35] B. Shneiderman, *Designing the User Interface*, Addison-Wesley, Reading, MA, 1987.

[36] M. Shumway, A. von Mayrhauser; "Applying Domain-based Testing and Sleuth to USWEST's Personal Voice Messaging Services", Technical Report Colorado State University, Computer Science Department, January 1996.

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

[38] Markos Z. Tsoulakas, Joe W. Duran, and Simeon C. Ntafos. "On Some Reliability Estimation Problems in Random and Partition Testing", *IEEE Transactions on Software Engineering*, 19(7), July 1993, pp. 687-697.

[39] A. von Mayrhauser, J. Walls, R. Mraz, "Sleuth: A Domain Based Testing Tool", *Procs. of the International Test Conference 1994*, Oct. 1994, CS Press, pp. 840 - 849.

[40] A. von Mayrhauser and S. Crawford-Hines, “Automated Testing Support for a Robot Tape Library”, *Procs. of the Fourth International Software Reliability Engineering Conference*, Nov 1993, pp. 6-14.

[41] A. von Mayrhauser, J. Walls, R. Mraz, “Testing Applications Using Domain Based Testing and Sleuth”, *Procs. International Symposium on Software Reliability Engineering 1994*, Nov. 1994, CS Press, pp. 206 – 215.

[42] A. von Mayrhauser, R. Mraz, J. Walls, “Domain Based Regression Testing”, *Procs. International Conference on Software Maintenance 1994*, Sept. 1994, CS Press, pp. 26 – 35.

[43] A. von Mayrhauser, R. Mraz; “The Sleuth Approach to Aerospace Software Testing”, *Procs. 1995 IEEE Aerospace Applications Conference*, February 1995, Snowmass, CO, pp. 61 – 75.

[44] T. Figliulo, A. von Mayrhauser, R. Karcich, “Experiences with Automated System Testing and *Sleuth*”, *Procs. 1996 IEEE Aerospace Applications Conference*, February 1996, Snowmass, CO, pp. 335 – 349.

[45] Elaine J. Weyuker and Bingchiang Jeng. “Analyzing Partition Testing Strategies”, *IEEE Transactions on Software Engineering*, 17(7), July 1991, pp. 703-711.

[46] Steven J. Zeil and Christian Wild. “A Knowledge Base for Software Test Refinement”, Technical Report TR-93-14, Old Dominion University, Norfolk, VA.