, , 1–35 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Test Case Generation as an AI Planning Problem

ADELE E. HOWE howe@cs.colostate.edu
ANNELIESE VON MAYRHAUSER avm@cs.colostate.edu
Computer Science Department, Colorado State University, Fort Collins, CO 80523

RICHARD T. MRAZ mraz@cs.usafa.af.mil
HQ USAFA/DFCS, 2354 Fairchild Hall, Suite 6K41, U.S. Air Force Academy, CO 80840

Received May 1, 1991

Editor: Dorothy Setliff

Abstract. While Artificial Intelligence techniques have been applied to a variety of software engineering applications, the area of automated software testing remains largely unexplored. Yet, test cases for certain types of systems (e.g., those with command language interfaces and transaction based systems) are similar to plans. We have exploited this similarity by constructing an automated test case generator with an AI planning system at its core. We compared the functionality and output of two systems, one based on Software Engineering techniques and the other on planning, for a real application: the StorageTek robot tape library command language. From this, we showed that AI planning is a viable technique for test case generation and that the two approaches are complementary in their capabilities.

Keywords: System testing, AI planning, blackbox testing

1. Automated Test Case Generation

Testing consumes a large amount of time and effort in software development. Although critical for ensuring reliability and satisfaction, much of the process is tedious: constructing data sets and/or sequences of commands to probe for faults. Some of the test cases will uncover faults, but one expects that most will not. A variety of approaches have been put forth for automated test generation in Software Engineering (SE) (see section 2).

Another approach is to acknowledge a similarity between some types of test cases (i.e., sequences of commands for software with command language interfaces) and plans in Artificial Intelligence (AI). Both are sequences of commands to achieve some goal; both need to conform to syntactic requirements of the commands and the semantic interactions between commands. Considerable research in AI planning has addressed subgoal and operator interactions; thus, the mechanisms of planning seem ideally suited to test case generation. In this paper, we explore the similarities between test case and plan generation, propose another approach, as implemented in a prototype, to automated test case generation using an AI planner and compare the new approach to an existing SE approach.

We selected the SE approach for comparison with two criteria in mind: First, the approach should be applicable to system testing and represent knowledge about the application at the system level. Second, the generation method must have been successful in practice as evidenced by at least a field study. Although our prototype might be small and limited in capabilities, testing it against a real application is crucial to heading off concerns about AI only being applicable to "toy problems".

These two criteria directed us to select Application Domain Based Testing (ADBT) (von Mayrhauser et al., 1994c), (von Mayrhauser et al., 1994d) and its test generation tool *Sleuth*. They support system testing of applications by building a model of the application domain which is used in the associated tool *Sleuth* to customize a test generation engine. Test generation is black box and is for user interfaces represented by a command language, or for transaction or request based systems. These characteristics fulfill our first selection criterion.

Application Domain Based Testing also fulfills the second criterion. Storage Technology Corp. has done a field study on the test generation method's usefulness (Figliulo et al., 1996). An experienced tester tested two major product releases, one with and one without using *Sleuth* over a full 12 week test cycle. The amount of new code and the development process were comparable. The test cycle without *Sleuth* uncovered 38 incidents, using *Sleuth* produced 135 incidents. Further, the post-release incident rate was 30% lower for the version tested with *Sleuth*.

2. Background on Testing

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.

Automated test generation for systems with a command language interface represents each command using a grammar, generates commands from the grammar, and runs the list of commands as the test case (for early work see (Purdom, 1972), (Bauer and Finger, 1979)). When using grammars for test case generation, we also need to address command language semantics (Bazzichi and Spadafora, 1982), (Celentano et al., 1980), (Duncan and Hutchison, 1981), (Ince, 1987).
(Duncan and Hutchison, 1981), (von Mayrhauser and Crawford-Hines, 1993) used attribute grammars for test case generation. The syntax and semantics of the command language were encoded as grammar productions. Test case generation is a single stage algorithm. This encoding poses difficulties (von Mayrhauser et al., 1994c); not the least of which is 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.

Transaction based systems and state transition aspects of some other systems have been tested using state machine representations (Chow, 1977), (Fujiwara et al., 1991).

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.

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. (Purdom, 1972) integrates "coverage rules" for grammar productions to reduce choice, while Maurer (Maurer, 1990) uses probabilistic context free grammars that are enhanced by selection rules including permutations, combinations, dynamic probabilities, and Poisson distribution. Thus, value selection is based on making choices relating to the representation of the command language or state machine.

Alternatively, one can argue that choices should be made depending on the functional characteristics of the system. Functional testing according to (Myers, 1979) uses heuristic criteria related to the requirements. (Goodenough and Gerhart, 1975) suggest partitioning the input domain into equivalence classes and selecting test data from each class. Category-partition testing (Ostrand and Balcer, 1988) accomplishes this by analyzing the specification, identifying separately testable functional units, categorizing each function's input, and finally partitioning categories into equivalence classes. (Richardson et al., 1989) considers 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 (Hamlet and Taylor, 1990), (Tsoulakas et al., 1993), (Weyuker, 1991). 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.

(von Mayrhauser and Crawford-Hines, 1993), (von Mayrhauser et al., 1994c), (von Mayrhauser et al., 1994d) developed a test generation method, Application Domain Based Testing, that 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). Each level has its own generation rules that represent syntax and semantics at that level. Definition of content of rules at each level is the result of a specialized application domain analysis. The resulting domain model forms the basis for test case generation.

So far, few approaches to software testing use artificial intelligence methods. (Deason et al., 1991) uses a rule-based system to test programs. The rules reflect white box criteria and use information about control flow and data flow of the code. To apply this method to system testing would require an entirely new set of rules. (Chilenski and Newcomb, 1994) use a resolution-refutation theorem prover to determine structural test coverage and coverage feasibility. Again, this aids white box testing, rather than system testing. (Zeil and Wild, 1993) 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.

(Anderson et al., 1995) use neural networks as classifiers to predict which test cases are likely to reveal faults. 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. After a subset 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. While the study is preliminary, the results are very encouraging for guiding and focusing testing, making it more efficient and effective.

3. AI Planning for Testing

Test cases are *sequences* of operations where the order matters. Operations may be executed only in particular contexts or in particular orders. Thus, coordinating the inclusion of operations and their interactions is a source of complexity in test case generation. Fortunately, representing and reconciling operation interactions is the purpose of most AI planning systems.

Roughly speaking, to generate a plan, a planning system is given: (1) a description of the operators, (2) an initial state of the world and (3) a goal state. Operator descriptions have parameters (i.e., what objects are involved in the operator), preconditions (i.e., what must be the case to use this operator) and effects (i.e., what happens to the system after the operator is executed).

Many classical or deliberative planning systems generate a plan by in effect proving that a sequence of actions will transform the initial state into the goal state. Planning works as follows: pick a goal to achieve, find an operator whose effects include the goal, add the preconditions of the operator to the list of goals to achieve and repeat the three steps until no goals remain unresolved or all unresolved goals are satisfied by the initial state.

What makes planning an attractive paradigm for software testing is the similarity of plans to programs and its emphasis on goals. From its early days, planning was thought of as a type of automatic programming (Sussman, 1973); the form of a plan and a test case can be made to be extremely similar. Unlike programs, plans do not always include control structures and new plans often must be generated for each test case.

The emphasis on goals means that sequences of actions (e.g., plans or test cases) are generated specifically to fulfill some purpose and that it is relatively easy to generate different plans for different goals. So, instead of focusing on *what* cases to generate, we think about *why* we wish to test certain aspects of the system and let the planning system determine *what* cases to generate. This appears to complement a goal oriented testing approach by human testers who start thinking about *what* they want to test before deciding *how* they will do so. Traditional generation methods are procedural, emphasizing how generation has to proceed.

Planning has been used for a variety of applications within software engineering. For example, Anderson and Fickas used planning as the underlying representation for software requirements and specification (Fickas and Anderson, 1988), (Anderson, 1993). The planner automates portions of the requirements engineering process: proposing a functional specification, critiquing the specification and modifying the specification to remove deficiencies. Another system, Critter, assisted analysts in designing composite systems (Fickas and Helm, 1992) by viewing the design of composite systems as problem solving. The system included tools for generating example plans/scenarios that violate problem constraints (a specific type of design test case), simulating portions of the design and expediting design decision making and evaluation.

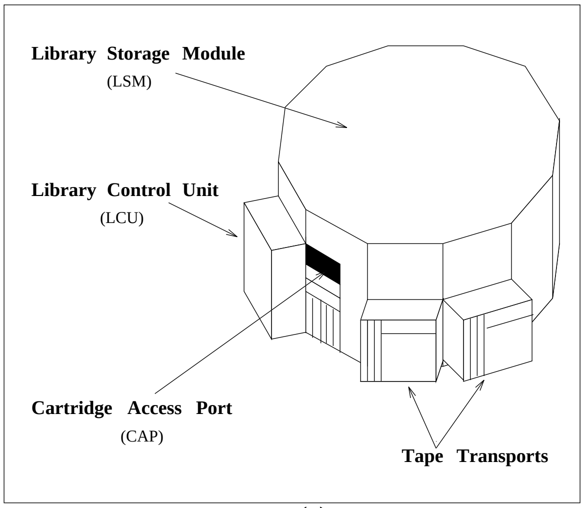
Huff has exploited the structure of plans and their ability to relate disparate goals in several applications in software engineering, e.g., (Huff88) in process engineering and (Huff, 1992) in software adaptation. Rist represented different levels of functionality and goals in programs using a plan representation (Rist, 1992); his PARE system extracted the abstract plan structure from PASCAL programs to aid in program design and re-use.

3.1. The Planning System for the Prototype

We used the UCPOP 2.0 planner as the basis for our prototype test case generator (Barrett et al., 1993), (Penberthy and Weld, 1992). The planner was selected because it is relatively easy to use and the software is easily obtained. UCPOP is a "Universal Conditional Partial Order Planner" which means that it can represent goals that include universal quantifiers (e.g., execute some operation on *all* possible objects) and that it does not order the sequence of operators in the plan until necessary, which makes it more flexible.

The planner requires a domain knowledge base of operators. Operators are represented in structures and are described in terms of their parameters, preconditions, and effects. Parameters in the operators refer to all of the objects appearing in the action description and are included in the preconditions and effects. Preconditions describe any state information that makes an operator eligible to be executed, and effects describe how state is changed by an operator.

As with many other planners, UCPOP builds a plan by incrementally adding actions that reduce the difference between the initial state and the goal state. To support that process, the domain description should be complete: include all effects and preconditions of all known operators. If it is incomplete or incorrect, the plan may be as well. Rules about which operator to apply are mostly handled by the



(a)

Figure 1. Library Storage Module (StorageTek, 1992)

planning system's manipulation of the operators, but may be tuned through the use of search control rules that direct selection of goals and operators.

4. Experimental Domain: StorageTek Robot Tape Library Command Language

Storage Technology Corporation (**StorageTek**) produces an Automated Cartridge System (ACS) that stores and retrieves cartridge tapes (StorageTek, 1992). The system maintains magnetic tape cartridges in a 12-sided "silo" called a Library Storage Module (LSM). Each LSM contains a vision-assisted robot and storage for up to 6000 cartridges. Tapes occupy cells in the panels. New tapes are entered through a special door called a Cartridge Access Port (CAP). Figure 1 shows a single LSM with tape drives, access port, and control unit. The robot inside the LSM identifies tapes using an optical scanner. Once a tape is identified, the robot moves the tape to a cell, mounts the tape in a tape drive, dismounts the tape, or ejects the tape through a CAP. One ACS can support up to sixteen LSMs. Figure 2 shows a "top-down" look at an ACS with three LSMs. Tapes move between LSMs through special doors called "pass-through-ports."

The ACS and its components are controlled through a command language interface called the *Host Software Component* (HSC). 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.

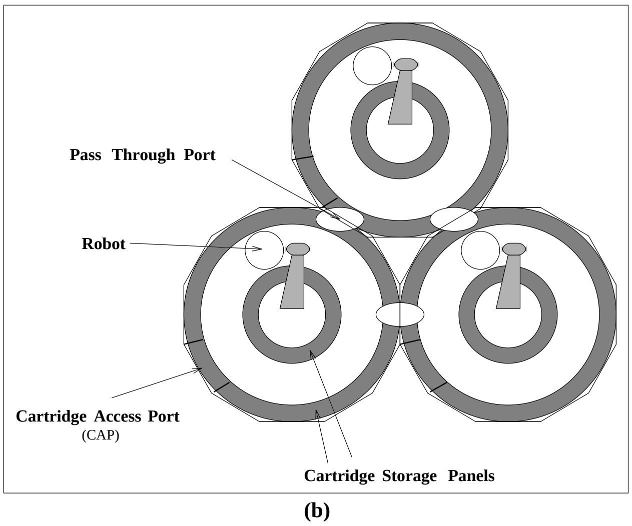


Figure 2. Automated Cartridge System with Three LSMs (StorageTek, 1992)

The experimental subdomain uses 9 commands and 11 parameters. The subdomain models "Moving" tapes within the ACS and "Mounting/Dismounting" tapes to/from tape drives. We chose this subdomain because it captures all the sophisticated aspects of the application while stripping from it several simple commands (involving command name and one or two possible parameter values) and those commands that are not part of behavioral rules. This gives the subdomain the realism of the application, without all the bulk.

5. Domain Analysis

Domain analysis is well known in the reuse community (Hooper and Chester, 1991), (Biggerstaff and Perlis, 1989). Here we apply it to build an application domain model from which to generate test cases. Thus our domain analysis is specialized due to its purpose: testing the application. In addition to domain components that are command language independent, we also need to capture the syntax of the command language. The representation should be such that no syntax independent changes are necessary when changing command languages (e.g., when comparing Storage Technology's versus IBM's versions of robot-controlled tape libraries).

Table 1 lists the steps for the domain analysis and the domain model components generated at each step. The next subsections describe details of applying this analysis (von Mayrhauser et al., 1994c). We will use the **StorageTek** automated tape library to illustrate each step.

Table 1. Domain Analysis Steps for Domain Based Testing

Domain Analysis Step	Domain Model Component
1. Object Analysis	
1.1. Define Objects and Object Elements	Set of Objects
1.2. Define Default Parameter Values and Object/Object Element Glossaries	Default Parameter Sets
1.3. Define Object Hierarchy	Object Hierarchy
1.4. Annotate Hierarchy with Parameter Constraints	Parameter Constraint Rules
2. Command Definition	
2.1. Command Language Representation	Command Language Syntax
2.2. Identify Pre/Post Conditions	
2.3. Identify Intracommand Rules	Intracommand Rules
3. Script Definition (Command Sequencing)	
3.1. Script Class Definition	Script Classes
3.2. Script Rule Definition	Command Sequencing Rules

5.1. Object analysis

The first step identifies the *objects* of the system, *object elements*, and *relationships* between the objects. *Objects* denote physical or logical entities from the problem domain. *Object elements* define qualities and properties of the object. Object relationships are used to define parameter value constraints. In Object Oriented Design (OOD), analysts and designers employ a variety of rules to identify objects (Booch, 1991). This specialized domain analysis focuses on *syntactic elements*, *user documentation*, and *user semantic interpretation* for object identification.

Step #1.1 identifies objects of the system under test. Each parameter in the command language is categorized according to the object it influences. This classification gives a first cut of the objects and their properties. Figure 3 shows how two HSC commands from the robot tape library have parameters that relate to three domain objects **Cartridge**, **Tape Transport**, and **HSC**.

The parameters within each object are *object elements*. Each object element is classified by defining its *object element type*. Object elements are similar to the concept of *object attributes* in OOA/OOD (Booch, 1991), (Rubin and Goldberg, 1992), except that for testing, we do not need as much information about an object when compared to the amount of information needed to implement one.

Each object element is classified as a parameter or non-parameter of the command language. Object elements related to command language parameters can be *parameter attributes*, *mode parameters*, or *state parameters*. A parameter attribute

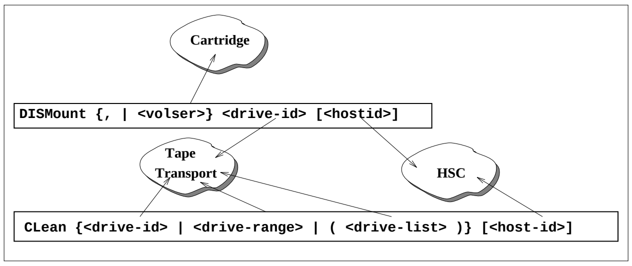


Figure 3. Analyzing HSC Commands for Objects and Object Elements

uniquely identifies an object. Mode parameters set operating modes for the system under test. State parameters hold state information for the object. Sometimes domain analysts provide semantic information that cannot be found in the parameters of the command language. These object elements are called *non-parameters* and may be important for test case generation. A non-parameter event is an event caused by the dynamics and consequences of issuing a sequence of commands. A non-parameter state is state information that cannot be controlled through the command language.

Step #1.2 defines default parameter sets for each object element and creates two glossaries, the *object glossary* and the *object element glossary*. The object glossary maintains information about each object by recording its name, a short description, a list of commands associated with the object, and the names of its object elements. Table 2 shows the **LSM** entry from the Object Glossary.

Another glossary stores detailed information about each object element. To select parameter values, automated test generation 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. Table 3 shows an entry from the Object Element Glossary for the StorageTek HSC command language.

The next step in the domain analysis determines relationships between objects. These relationships are captured in an *object hierarchy* in the form of a structural or "part-of" hierarchy. Step #1.4 annotates the object hierarchy with parameter constraint rules which describe how the choice of one parameter value constrains the choices for another. For example (see Figure 4), each ACS supports up to sixteen LSMs (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 state 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).

Table 2. 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*
Parameter State	*lsm-status*
Non-parameter Event	*lsm-full*
Non-parameter State	none

Table 3. LSM Entry from the HSC Object Element Glossary

Parameter Attribute	*lsm-id*
Full Name	Library Storage Module (LSM) Identifier
Definition	Names an Instance of an LSM within an ACS
Values	000...FFF
Object	LSM
Representation	Range of values

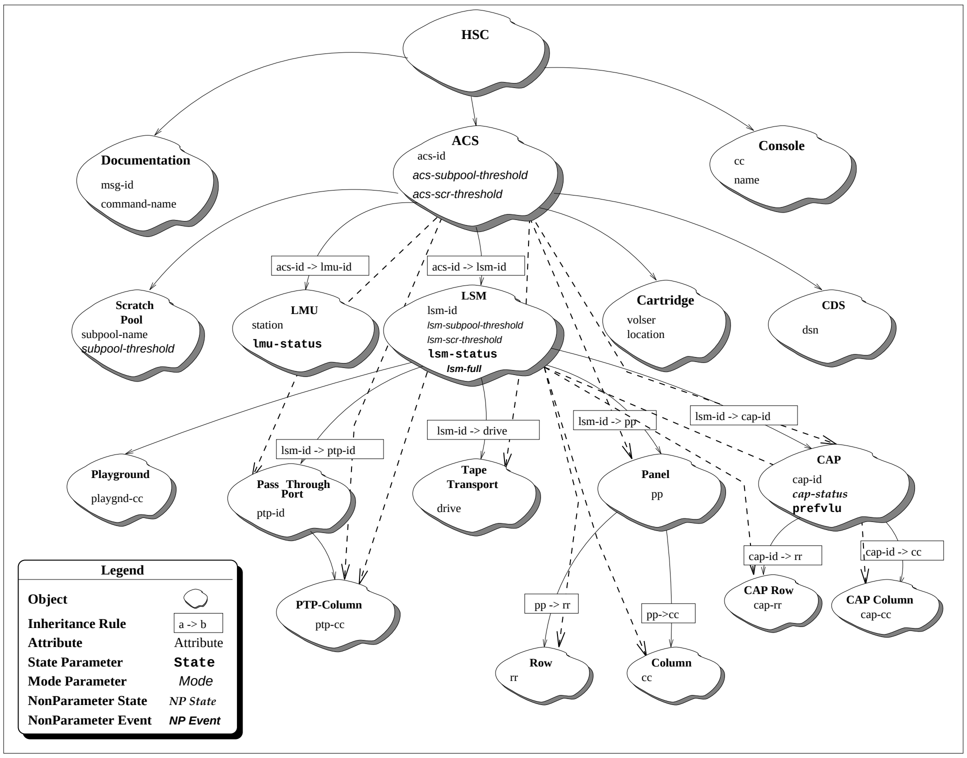


Figure 4. StorageTek Object Hierarchy

5.2. Command definition

Step #2 of the domain analysis defines command syntax and semantic rules for each command. Three types of semantic rules are defined for commands: *preconditions*, *postconditions*, and *intracommand rules*. Preconditions ensure proper state or mode for a command and may constrain valid parameter values. Postconditions state effects on object elements and influence future command sequences or parameter value selection. The third command level semantic rule is called an *intracommand rule*. These rules identify constraints placed on parameter value selection within a command. To illustrate, user documentation for the StorageTek HSC `MOVE` command states, "when moving a tape within the same LSM, the source and destination panels must be different." The domain model captures this intracommand rule as: `if (lsm$1=lsm$2) => (panel$1` $\neq$ `panel$2)`.

5.3. Script definition

Step #3 describes dynamic system behavior by capturing rules for sequencing commands and by classifying commands from the problem domain. Sequencing information is necessary because arbitrarily ordering a list of commands rarely produces semantically correct test cases.

The first part of scripting analysis is to group related commands into *scripting classes*. Scripting classes can partition by function, object, or object element. Functional partitioning creates scripting classes that include commands that perform similar actions. For example, in the StorageTek domain, the *set-up* class includes all commands that perform system set up functions; the *action* class includes commands that manipulate and exercise the robot tape library. If we partition the commands by object, then we examine each object and create a class that contains all commands that influence that object.

Step #3.2 of the scripting analysis defines command sequencing rules. The results from this step include command sequencing information (i.e., script rule) and parameter binding information for each rule. For example, in the robot tape library, tapes must be mounted before they can be dismounted. Scripts are visualized as state transition diagrams (see Figure 5). Arcs are labelled with the names of specific commands or script classes. By restricting the commands that can be executed on each arc, we can create proper command sequences.

A command sequence can be annotated with parameter selection rules, as shown in Table 4. The first rule, p*, states that the value for parameter p can be selected from any valid choice as long as it fulfills parameter 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.

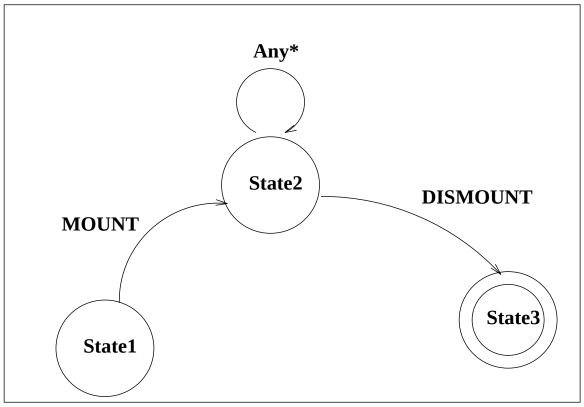


Figure 5. State Transition Diagram for the MOUNT-DISMOUNT Script Rule

Table 4. 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*

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

This rule states that the *tape-id* and *drive-id* parameters can be selected from any valid choice for the MOUNT command while the DISMOUNT command must use the previously bound value for the *tape-id* and the *drive-id* parameters. Simply stated, the tape that is mounted in a drive should be dismounted from the same drive.

6. Test Generation in Sleuth and UCPOP

The two approaches were developed from the StorageTek domain analysis. The Application Domain Based Testing method was implemented in *Sleuth*, an automated test generation tool developed at Colorado State University. The AI planning approach was implemented in UCPOP augmented by Lisp code.

Sleuth supports Domain Based Testing by providing tools and utilities for test generation. The main window directs the test generation process (see figure 6). *Sleuth* provides utilities to create domain models, configure test subdomains, and to generate tests. *Sleuth* is designed to maximize reuse of commands at all levels of abstraction (von Mayrhauser et al., 1994b), simplify uniform testing across configurations and versions, and support regression testing (von Mayrhauser et al., 1994a).

In this section, *Sleuth* and the planner implementation are described in terms of their architectures and knowledge representations. The two approaches are compared in terms of the test cases they generate and their basic capabilities.

6.1. Architectures

6.1.1. Sleuth Architecture

Sleuth is based on the Test Generation Process Model (shown in figure 7). The domain model, D_0^v, captures the syntax and semantics of the system under test. Tests generated from D_0^v are "valid" sequences of commands that follow all syntax and semantic rules in the domain model. Often, the domain model is modified to test a specific system configuration or to test a particular feature of the system under test. This creates a *test subdomain*, TSD_j^v, one for each modification j. *Test Criteria* influence the test subdomain definition and the test generation steps. Test engineers use their knowledge to modify the domain model. They also guide test generation by recalling archived test suites, identifying how many commands to generate, and what commands to generate.

Test Generation uses the test subdomain and instructions from the test engineer to create test suites, T_j^v. A *test suite* for DBT may contain *test cases*, *test templates*, and *test scripts*. A test case is a list of fully parameterized commands from the syntax of the problem domain. A test template is a list of commands with place holders for parameters. Test scripts are lists of command names.

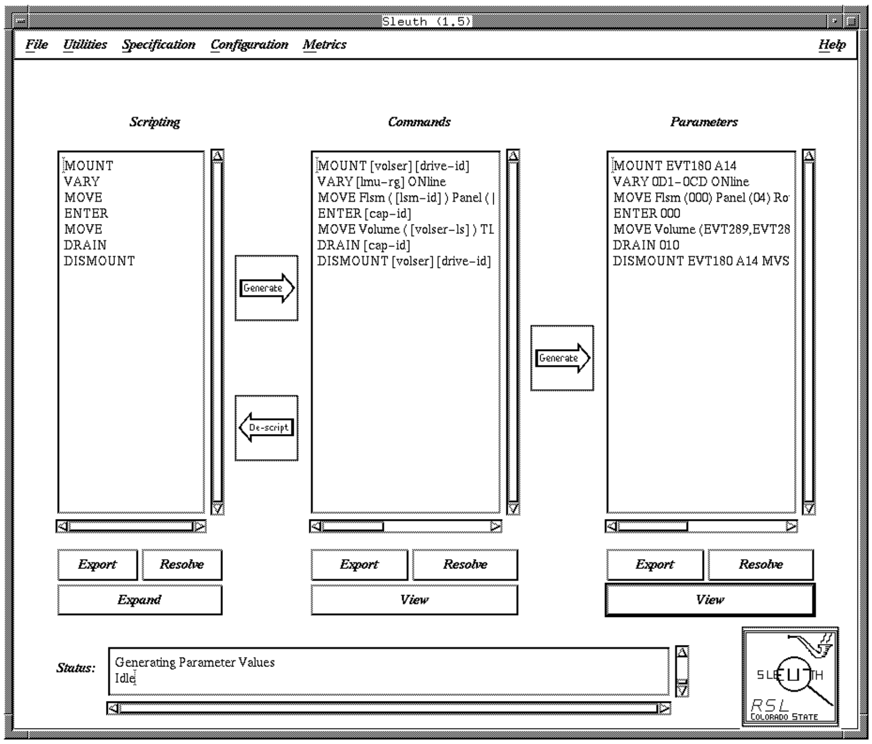


Figure 6. Window Based Test Generation Tool

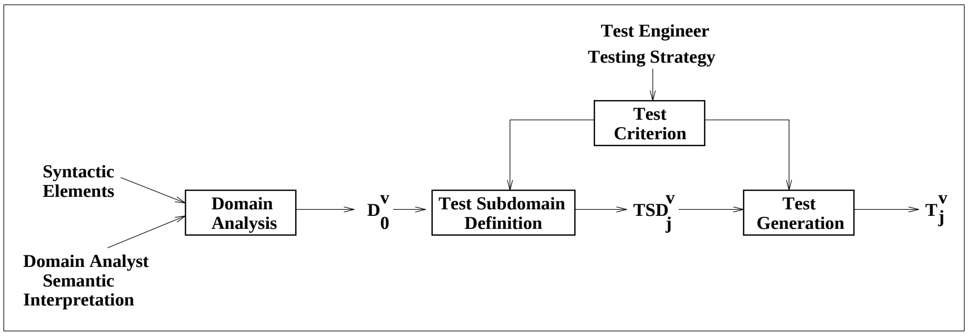


Figure 7. Test Generation Process Model

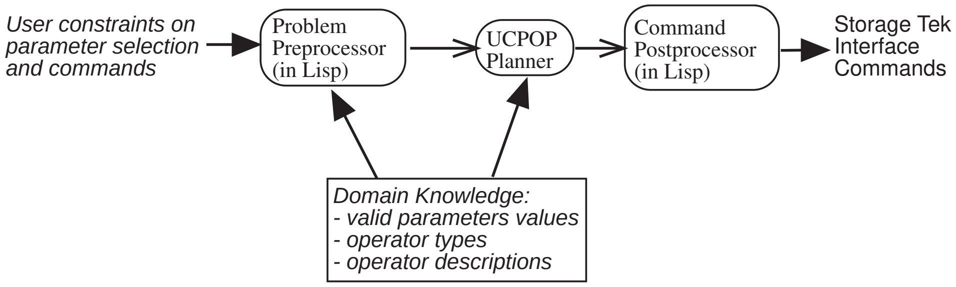


Figure 8. Sequence for generating test cases using UCPOP

As shown in figure 6, *Sleuth* uses a three stage test generation process. In the first stage, script classes and script rules are expanded. This produces a list of command names. The second stage creates a command template. Parameters remain as place holders. The last stage uses script parameter binding rules, parameter value sets, and parameter constraint rules to create a fully parameterized list of commands.

6.1.2. Planning Architecture

The planning version divides test case generation into three steps: generating a problem description, creating a plan to solve the problem, and translating the plan into test case notation. As shown in Figure 8, these three steps correspond to three modules, respectively: 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 is the knowledge base that describes the commands in the language. The knowledge base is described in section 6.2.2; ways of manipulating the knowledge base are described in section 7. 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 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. For example, if the user requests three move operations to be accomplished, the preprocessor defines an initial state with at least three tapes in randomly selected positions and a goal state which specifies three new randomly selected locations for the tapes. The initial state also includes information about the robot tape library configuration and initial status information; the configuration information is taken directly from the knowledge base and the initial status information is randomly generated from the problem constraints.

Table 5. Domain Model Components and Hybrid Representation

Domain Model Component	Hybrid Representation
Script Classes	Sets of Command Names
Script Sequencing Rules	Macro Expansion
Script Parameter Binding	Macro Expansion
Command Language Syntax	Syntax Diagrams
Command Preconditions	Implicit representation
Command Postconditions	Implicit representation
Intracommand Rules	First Order Logic
Parameter Constraint Rules	Parameter Value Sets Parameter Constraints based on Set Operations

The planner, UCPOP, constructs a plan to transform the initial state into the goal state. If a plan cannot be found within a set amount of time, then the planner fails. In this case, we try a different initial state and goal state that satisfies the user's requirements. Because UCPOP is a fairly simple and inefficient planner, more complex and demanding problem descriptions are more likely to fail. Because this is a prototype, we did not attempt to improve UCPOP's performance on our problems.

The postprocessor translated from UCPOP's plan representation to the HSC syntax. The transformation was purely syntactic and relatively straightforward.

UCPOP is written in Lisp. Thus, the supporting code was written in Lisp and could create data structures for UCPOP, run UCPOP and access and modify the return values from UCPOP. The declarative nature of the representations in Lisp expedited automatically modifying the knowledge base and so changing the nature of the kinds of test cases generated.

6.2. Knowledge Representation

6.2.1. Sleuth Representations

Sleuth uses a hybrid domain model representation (see Table 5) during test generation.

Script Representation

Table 6. Commands and Script Classes for the StorageTek Experimental Subdomain

Command Name	Description	Script Class	
Dismount	Dismount a tape from a tape drive	Any	Action
Display	Display information to the console	Any	Action
Drain	Release the Cartridge Access Port	Any	Action
Eject	Eject tapes through CAP	Any	Action
Enter	Enter tapes through CAP	Any	Action
Modify	Turn an LSM Online or Offline	Any	Set-Up
Mount	Mount a tape in a tape drive	Any	Action
Move	Move a tape within an ACS	Any	Action
Srvlev	Set system service level	Any	Set-Up

Scripts capture dynamic behavior of the system under test. We represent three categories: Script Classes, Script Rules, and Script Parameter Binding. A scripting class helps testers select *what type* of commands should be generated for a test case. A command can be a member of more than one class; the number of classes and the types of script classes is problem dependent. The experimental subdomain for this research used the commands in Table 6 and assigned them to three script classes. The Any class contains all commands. Set-up commands perform machine set-up. Action commands cause physical actions within the ACS.

For script rules, macro expansion ensures that commands are sequenced properly. For example, in the robot tape library, one cannot "dismount" a tape from a tape drive unless one has previously been "mounted." The macro representation for this rule is: MOUNT <List of Commands> DISMOUNT. *Parameter binding* makes sure the parameters in the command sequence are meaningful. For instance, parameter binding information ensures that the tape that is "mounted" is the same tape that is "dismounted." This implicitly handles preconditions and postconditions in the domain model as they relate to command sequencing.

Command Representation

The second component of the domain model captures the syntax and semantic rules for each command. Command syntax is needed to generate a command from the command language. *Sleuth*'s representation uses syntax diagrams for each command and a random-walk through the syntax creates an instance of a command.

Intracommand rules identify constraints placed on parameter value selection within a single command. These rules are represented in the domain model in first-order logic.

Parameter Representation

Parameters of the command language are represented as Parameter Value Sets. Using the set operations of Union, Intersection, and Difference, a variety of parameter sets can be defined. During test generation, parameter values may be constrained. For example, the value for a particular LSM (*lsm-id*) constrains possible values for panels, rows, and columns for cartridge storage. Parameter constraint rules are represented using a "parameter" hierarchy that denotes relationships between "parameters" and set operations to modify parameter value sets from which the test suite generator may select.

6.2.2. Planning Representations

UCPOP provides its own representations for planning operators; in addition, we represented domain knowledge in structures, lists and procedures in Lisp to support the pre- and post-processors. Table 7 lists each domain model component and its planning representation; each one is described subsequently.

Script Representation

A script class is represented as the planner's domain. In UCPOP, the domain is simply a list of operator definitions. In the experimental subdomain, the list contains eighteen operations: four versions of the move command, eject, enter, connect, servicetofull, servicetobase, modifytooff, modifytoon, drain, mount, dismount, and four versions of the display command.

Script sequencing rules and parameter binding together legislate the correct ordering of commands. The AI planning representation includes this information implicitly in the preconditions and effects of the planner operators. For example, one sequencing rule requires that `MOUNT` precede `DISMOUNT`, which is implemented with the planner operators listed in Figure 9. Preconditions for `DISMOUNT` require a tape to be in a tape drive. A tape can be placed into a drive in one of two ways: (1) a tape can be in the tape drive in the initial state, or (2) the tape can be loaded as an *effect* of the `MOUNT` operator.

Command Representation

The next domain model component is the command language syntax. Unlike the grammar based methods, the AI planner does not explicitly represent the syntax of the command language. Instead, each path through a command is represented as a separate planning operator. A path through a command differs primarily in the type and number of parameters for the command. Thus, the resulting plan operators may differ in their parameters, possible parameter values and preconditions. For instance, the StorageTek command language uses the `SRVLEV` command to "toggle" the system's service level: `service-level-cmd ::= SRVLEV {BASE | FULL}`. As seen in Figure 10, two operators encode the command for the planner: one to change to "full" and the other to change to "base". To implement the toggling

Table 7. Domain Model Components and AI Planner Representation

Domain Model Component	Planner Representation
Script Classes	Collection of Planner Operators
Script Sequencing Rules	Operator preconditions
Script 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
Intracommand Rules	Operator preconditions
Parameter Constraint Rules	Preprocessor : Initial State Generator Preprocessor : Goal Generator Preprocessor: supporting data structures

```
;;Description : Mount a tape in a tape drive.
;;Precondition : Service-Level = FULL.
;;               Tape is inside the LSM.
;;               LSM-status = ONLINE.
;;Postcondition: Tape is in the tape drive.
(define   (operator mount)
          :parameters ((loc ?slsm) ?vid ?m_did ?p ?c ?r )
          :precondition (and (full slev)(in ?vid ?slsm ?p ?r ?c)(on ?slsm))
          :effect (and (at ?vid ?m_did)(not(in ?vid ?slsm ?p ?r ?c))))

;;Description : Dismount a tape from a tape drive.
;;Precondition : Service-Level FULL
;;               Tape is in the tape drive.
;;Postcondition: Tape is placed into the LSM.
(define   (operator dismount)
          :parameters(?vid ?m_did ?d_did ?p ?c ?r)
          :precondition(and (full slev)(at ?vid ?m_did)(eq ?m_did ?d_did))
          :effect (and (not (at ?vid ?m_did))(backtolsm ?vid through ?d_did)
                       (in ?vid unknown unknown unknown unknown)))
```

Figure 9. Planning representations of the Mount and Dismount commands

```
;;Description  : Change Service-Level to FULL.
;;Precondition : None.
;;Postcondition: Service-Level = FULL.
(define    (operator servicetofull)
           :precondition (base slev)
           :effect (and(not(base slev))(full slev)))

;;Description : Change Service-Level to BASE.
;;Precondition : None.
;;Postcondition: Service-Lever = BASE.
(define    (operator servicetobase)
           :precondition (full slev)
           :effect (and (not (full slev)) (base slev)))
```

Figure 10. Example planning representations of the command language syntax: changing service level

effect, the precondition is that the current level must be other than the value to which it is being set.

Each planning operator is represented in a form similar to the command syntax (i.e., same number and type of parameters but formatted in the planner's representation). In the case of the two service level commands, the planning action `SERVICETOFULL` is converted to the StorageTek command `SRVLEV FULL`. A postprocessor translates the planner output into the correct syntax.

The next three domain model components are command preconditions, command postconditions, and command intracommand rules. All three are represented as preconditions and effects of planner operators. A command language precondition denotes sequencing information based on system state (in contrast to the script sequencing rules which are not assumed to involve system state). If the system is not in the correct state, the precondition provides information to put the system in the proper state. Likewise, command language postconditions specify how the state of the system changes upon executing an operator. For a command to be included in a valid plan, its preconditions must be satisfied by the initial state or by the effects of an action that precedes it.

Intracommand rules are also specified as preconditions to planner operators. These preconditions check parameter values within the command. Figure 11 shows one version of the `MOVE` command, which encodes the previously discussed intracommand rule by requiring `slsm` (source LSM) and `dlsm` to be equal (`eq`) and `sp` (source panel) and `dp` (destination panel) to be not equal (`neq`). Another version of the move command covers the case in which the LSM's differ.

Parameter Representation

The last domain model component represents command language parameters and parameter constraints. The AI planner uses two components of the preprocessor to capture this information: Initial State Generator and Goal Generator. The

```
;;Description : Move a volume in a specific location to the destination LSM,panel.
;;Precondition : Source and destination LSM are online.
;;               Service level is full.
;;               Volume location is specified by LSM, panel,row,column.
;;               Source and destination LSMs are connected.
;;Intracommand : Source and destination LSMs must be equal.
;;               Source and destination panels must be different.
;;Postcondition: Move the volume to a new panel inside the same LSM.

(define   (operator movefour)
          :parameters ((loc ?slsm) ?sp ?sc ?sr (loc ?dlsm) ?dp ?dc ?dr (tape ?vid))
          :precondition (and (full slev)(on ?slsm)(on ?dlsm)
                             (eq ?slsm ?dlsm) (neq ?sp ?dp)
                             (in ?vid ?slsm ?sp ?sr ?sc)
                             (eq ?dc unknown) (eq ?dr unknown))
          :effect (and (from ?vid ?slsm ?sp ?sr ?sc ?dlsm ?dp ?dr ?dc)
                       (in ?vid ?dlsm ?dp ?dr ?dc)
                       (not (in ?vid ?slsm ?sp ?sr ?sc))))
```

Figure 11. Example planning representation of the command preconditions, postconditions and intracommand rules: move a volume

initial state generator creates an initial state vector for the planning system by randomly choosing from options constrained by the user's goals and the parameter constraints. Parameter constraints are represented declaratively where possible and procedurally where necessary. For example, the configuration of the LSMs (their panels, rows, columns, etc.) are represented in structures and lists; the relationship of the variables in the configurations (e.g., panels have rows and columns) is represented in the preprocessing code.

In the StorageTek experimental subdomain, we concentrated on two test generation goals: Moving tapes and Mounting/Dismounting tapes. Therefore, all state information necessary for these experiments was included in the state vector. Figure 12 shows an example of an initial state which includes basic system status (e.g., service level), configuration (e.g., three LSM locations are available) and inventory (e.g., one tape is in `LSM 10`).

The second preprocessor uses object and parameter constraint information to generate a goal for the planner. A single goal for these experiments is to move an individual tape, to mount/dismount a single tape, or display information. If more than one command is needed, a conjunctive goal is created. An example of a compound goal is shown in Figure 12. This compound goal was generated based on a request to create a test case with two tape movements in it. The fields in the goal statement are:

```
(FROM [tape-id] [src  lsm] [src  panel] [src  row] [src  column]
                [dest lsm] [dest panel] [dest row] [dest column])
```

initial-state =	((BASE SLEV) (LOC 0) (ON 0) (CAP 0 ENTERING) (LOC 1) (OFF 1) (CAP 1 ENTERING) (LOC 10) (OFF 10) (CAP 10 ENTERING) (CONNECT 0 1) (TAPE EVT297)(IN EVT297 10 UNKNOWN UNKNOWN UNKNOWN))
goal =	(AND (FROM EVT280 0 UNKNOWN UNKNOWN UNKNOWN 1 UNKNOWN UNKNOWN UNKNOWN) (FROM UNKNOWN 0 1 2 3 0 4 5 6))

Figure 12. Example planning representation of objects, object elements and parameter constraints: initial state list and goal list

The first subgoal requests the system to move tape `EVT280` from `LSM 000` to `LSM 001`. In this goal, we are not concerned about where the tape is located. We are only concerned about moving it to a different LSM. In the second subgoal, a tape located in panel 1, row 2, and column 3 is moved within the same LSM to panel 4, row 5, and column 6. These two examples show how testers can focus test generation at different levels of abstraction through planning system goals.

6.3. Example Test Cases

The planner based generator and Sleuth produce qualitatively different test cases. Sleuth generates commands from user directions on number and type and primarily by resolving sequencing rules with macro expansion and by making choices at points in a grammar; the planner version generates initial and goal states from choices based on user directions on number and type of commands and by generating a legal sequence of actions to achieve the selected goals.

Table 8 shows test cases generated by both systems when they were asked for a test case with a move command. The solution formulated by *Sleuth* generates a sequence of `MODIFY` commands. The first two perform useful work, but the third is redundant because *Sleuth* does not maintain state information. The `ENTER` command is a result of *Sleuth*'s random command selection. The `ENTER` command requires a sequencing rule: `ENTER` followed by one or more other commands followed by a `DRAIN`. Command #5 finally issues the `MOVE` command as required. The last two commands complete the test case. StorageTek testers interpret test cases like this from the point of view of testing a *shared device*. While this test case seems to have redundant or extraneous commands, the testers consider the sequence a merged list of instructions from multiple users.

The UCPOP test case must be generated in the context of the initial state and the goal. The initial state sets `LSM 000` offline, Service level to Base, `CAP 000` to Entering, and tape `EVT280` located outside the ACS. The goal was to move a particular tape to `LSM 001`. The first two commands issued by the planner adjust

Table 8. Comparing *Sleuth* and UCPOP Tests

	Sleuth Test Case	UCPOP Test Case
1	MODIFY 001 ONline	MODIFY 000 ONLINE
2	MODIFY 000 ONline	SRVLEV FULL
3	MODIFY 000 ONline	DRAIN 000
4	ENTER 000	ENTER 000
5	MOVE (EVT289) Tlsm(000)	MOVE VOLUME(EVT280) Tlsm(001)
6	MODIFY 001 ONline	
7	DRAIN 000	

the state of the system such that other commands are meaningful and can be executed. The `DRAIN` command is necessary because the initial state of the CAP is Entering. Since the CAP is a shared device, it must be released by the `DRAIN` command first. The `ENTER` command is important because tape `EVT280` is currently located outside the robot tape library. After the tape is entered, the system issues the `MOVE` and achieves the goal. The UCPOP test case is shorter and more focused than the Sleuth case.

The planner's test cases incorporate more assumptions about the state of the system and compose the test cases based on different criteria. For example, Figure 13 shows a simple example of a test case developed by the planner and supporting code and represented in UCPOP's language. In response to a request for a test case with one dismount and one display command, the preprocessor generates an initial state with a tape, `EVT297`, located in `LSM 010` `(IN EVT297 10 16 7 23)` and its LSM online `(ON 10)`. The goal is to display a console status report about the LSM and to move the tape from tape drive `A36` to its LSM.

UCPOP's solution is listed in step 1-3, and the postprocessor output is listed last. In this case, to generate a dismount, a tape must be in the drive, so a mount command is generated first. Each step is listed with its preconditions and each precondition is prefaced by the step number that satisfies it (a zero indicates it is satisfied by the initial state). Thus, a minimal plan for the test case requirements is three steps because the required preconditions concerning status are all fulfilled by the initial state. The postprocessor examines each step and generates the correct syntax for the StorageTek Robot Tape Library.

6.4. Comparison of Capabilities

We implemented 18 operators in the planning system which correspond to 9 commands in the StorageTek domain. In the *Sleuth* representation, the 9 commands

```
Initial:        ((FULL SLEV) (LOC 0) (OFF 0) (CAP 0 DRAINED) (LOC 1) (OFF 1)
                (CAP 1 DRAINED) (LOC 10) (ON 10) (CAP 10 ENTERING) (CONNECT 0 1)
                (LOC ACS) (TAPE EVT297) (IN EVT297 10 16 7 23))

Step 1:         (MOUNT 10 EVT297 A36 16 23 7) Created 2
                0 -> (IN EVT297 10 16 7 23)
                0 -> (LOC 10)
                0 -> (ON 10)
                0 -> (FULL SLEV)
Step 2:         (DISPLAY2 0) Created 3
Step 3:         (DISMOUNT EVT297 A36 A36 ?P1 ?C1 ?R1) Created 1
                0 -> (FULL SLEV)
                1 -> (AT EVT297 A36)

Goal:           (AND (CONSOLEMSG DISPLAY2 0)
                    (BACKTOLSM EVT297 THROUGH A36))

Postprocessor:

step 1 is:      (MOUNT EVT297 A36)
step 2 is:      (DISPLAY LSM 010)
step 3 is:      (DISMOUNT EVT297 A36)
```

Figure 13. Example Results from UCPOP : Goal = Display console message about LSM and replace a tape from tape drive A36 back to the LSM

required 9 syntax diagrams, 4 scripting rules, 1 intracommand rule, and eleven parameter files. We compared the two approaches in three ways: how much effort was required for model development and test generation, whether the test cases covered similar aspects of the domain, and what kinds of test cases were generated by the planning system.

Although the representation language required by the planner was unfamiliar to those programming the test cases, the planner implementation progressed fairly easily and resulted in remarkably little code. The bulk of the implementation was done by Li Li, an undergraduate who had no background in AI, but some in Software Engineering, over a 12 week period which included learning about domain based testing, *Sleuth* and AI Planning. Of course, this rapid development was expedited by the fact that the domain was well understood and had already been implemented in *Sleuth*.

UCPOP required very little "code" to represent the experimental subdomain. The two preprocessors, one postprocessor, and 18 operators needed 414 lines of code. For *Sleuth*, the entire test generation tool required about 25,000 lines of C (internals and Motif interface). However, *Sleuth* implemented a much larger test domain than UCPOP and included utilities for domain model definition, test subdomain configuration, metrics, and a sophisticated user interface. Despite the small code size in the planner, computation time for a test case could take from minutes to hours due to the inefficiency of the underlying planner.

We compared the domain coverage of the planning and *Sleuth* implementations by considering how each represented the seven levels of the domain model (as appear in Table 5). Both the *Sleuth* and planner representations employed similar mechanisms to implement Script Classes. The *Sleuth* representation stores command names in a set, and the planner includes or excludes operators to form script classes.

Scripting Rules (sequencing rules and parameter bindings) require different approaches in the two representations. The *Sleuth* scheme needs command sequencing information in the first stage of test generation but it does not need parameter binding information until the last stage. The AI planner includes sequencing information and parameter binding rules in each operator.

The Command Syntax is represented differently in the two test generation engines. The planning system encodes each "path" through a command as a separate operator. One operator could represent multiple paths so long as all paths use the same parameters, the same preconditions, and produced the same effects; UCPOP allows some flexibility in the preconditions but little in the definition of the effects[1]. The test case is generated using a postprocessor to translate the UCPOP output into the appropriate StorageTek syntax. *Sleuth* stores each command as a syntax diagram and takes a random-walk through the syntax diagram to create a command template.

Command Preconditions and Postconditions are not explicitly represented in the *Sleuth* representation. They are either implicit in the script rule macros or testers must issue set up commands or execute a list of commands to put the system in the correct state. *Sleuth* does not keep track of non-parameter state information.

The planner was able to represent pre/post conditions in each planning system operator. The trade off between these two representations is in the amount of state information required during test generation.

Parameter value sets and parameter constraints are handled differently in both domain model representations. *Sleuth* uses parameter value files, set definitions, and set operations. The default values for a parameter are defined using sets and set operations. If one parameter value constrains the choices for another, additional set operations allow *Sleuth* to choose from a constrained set. In the planning system, all parameter information was encoded into the preprocessor, which uses information about default parameter values and parameter constraint rules to define initial states and goals that do not violate the parameter selection rules.

One of the most interesting aspects of our results were the differences between the planning approach and the *Sleuth* approach to test data generation. Testers take different views of the problem during test generation. Using *Sleuth*, the tester focuses on what subset of commands to generate and how many commands. Using the planner, the tester describes the desired *outcome* and allows the planner to choose the appropriate sequence of commands to achieve the goal. We view planner based (goal oriented) test generation as a natural way to generate tests.

An unexpected result of our comparisons was the kind of tests that were generated. The planner can potentially "discover" unusual command sequences to achieve the goal. This is beneficial in test data generation because obvious approaches get tested most often and therefore find few or no faults. Unusual command sequences may achieve the same goal but uncover faults from command sequences that were not considered. For instance, one of our experiments required UCPOP to move a tape from one LSM to another. Instead of generating a `MOVE` command, it `EJECT`ed the tape from one LSM and `ENTER`ed it into the next. While this is a simple example, it shows how the planner can create innovative test sequences that the test engineers may not think about.

7. Extensions to the Basic System

Test generation systems should be flexible enough to accommodate a variety of test intents and testing criteria. Therefore, the extensions to the basic planning system included capabilities for testing against a variety of common testing criteria. Usually, testers test both valid and invalid cases. Testers may want to focus testing on particular parts of the system (a specific testing subdomain) for a variety of testing objectives: testing a particular (set of) objects, testing specific commands, regression testing (testing new features and making sure that nothing was broken), or testing specific configurations (represented as subsets of values in the domain model). At other times, testers may need to test all possible values of a specific parameter (e.g., a parameter reflecting possible tape media). Testers may or may not be concerned about the order in which parameter values are chosen. This section illustrates how these testing objectives can be achieved with *Sleuth* and with the Planning system UCPOP. *Sleuth* provides test subdomain utilities for this purpose.

For the planning system, operator descriptions are declarative and accessible from Lisp; thus, functions are easily built that modify the underlying knowledge base.

7.1. Invalid Test Cases

For systems with a command language as a user interface, invalid cases can represent themselves in a variety of ways from testing erroneous syntax and violating rules of proper behavior (violating scripting rules and intracommand rules in the script) to purposely generating invalid parameter values, either by using values that are always invalid, or by generating values that are invalid in a specific context (e.g., values that violate parameter inheritance rules). This type of testing concentrates on the syntax of the command language and tests the error recovery capabilities of the parser. It also tests error recovery code related to semantically erroneous commands.

In *Sleuth*, one can turn off scripting and intracommand rules, making it possible to generate dynamically invalid command sequences interspersed with semantically correct command sequences. If a tester wants only sequences that violate rules of proper behavior, a *Sleuth* utility allows editing the rules (modification or negation) which would then guarantee breaking of the rules in specific ways and thus test error recovery code of the system under test. We can also edit the syntax with mutation operations. This causes commands with incorrect syntax to be generated. Similarly, parameter value utilities allow the tester to define parameter values that are invalid and to modify parameter inheritance rules so they violate the actual domain's.

We can duplicate these capabilities in the AI Planner version. Scripting and intracommand rules are built into the preconditions of the planner. Thus, the first mechanism for generating invalid test cases is to identify such preconditions and either remove or change them. For example, one scripting rule is that `DISMOUNT` should be proceeded by `MOUNT`. If the precondition that legislates that sequencing (i.e., that the tape be at the drive) is removed, then `DISMOUNT` may not be preceded by `MOUNT`, as is the case in figure 14a.

One of the intracommand rules for the domain dictates whether a destination and source `LSM` must be equal (or not) for particular types of `MOVE` commands. The preconditions for the different move operators include these constraints; thus, the intracommand rules can be invalidated by toggling the equality constraints in the operator descriptions. Doing so produces test cases that are no longer syntactically correct. Given this change to the operators and a request for a test case with a `MOVE` command, the planner version produced the test case in figure 14b, in which the `MOVE` command includes the value "unknown" for a panel.

Additionally, we can invalidate the parameter set used by the preprocessor. For example, we can generate invalid tape ids, ones composed of illegal character combinations (e.g., `123ABC`, `ABC`, or `!$#@&*`), use these in the preprocessor and so generate illegal syntax, as in the tape number listed in figure 14c.

Test Case A	Test Case B	Test Case C

DISMOUNT EVT280 A36
ENTER 010
SRVLEV FULL
MOUNT EVT289 A29
	MODIFY 001 ONLINE
	MODIFY 010 ONLINE
	MOVE FLSM(01) PANEL(19) ROW(14)
	   COLUMN(10) TLSM(10)
	   TPANEL(UNKNOWN)
	EJECT 010 EVT289
	DRAIN 000
	ENTER 000
		MODIFY 001 ONLINE
		DRAIN 001
		ENTER 001
		MODIFY 000 ONLINE
		MOVE VOLUME (X12B&C)
		   TLSM (000)

Figure 14. Invalid test cases: a) removing scripting rules, b) invalidating intracommand rules, c) illegal parameter values

7.2. Subdomains

During feature testing with *Sleuth*, testers may decide to focus their testing efforts by excluding certain parts of the domain. This can take several forms: If scripting classes were defined around similar functions, a tester may decide to turn off commands in some classes, which ensures that these commands will never be generated. Similarly, if scripting classes were defined around objects (such as tapes in the HSC domain), then a tester may choose to turn off all but the commands in the `tape` scripting class (which contains all commands that operate on tapes).

Further, testers may want to subset the application domain by constraining possible values for parameter values (subsetting parameter value sets). *Sleuth* supports a parameter value editor. For example, in the HSC domain, testers may only have a limited supply of named tapes (and their associated `tape-id` parameter).

Test subdomains can also be generated by test criteria rules. An example of such rules are the regression testing rules of (von Mayrhauser et al., 1994a). The approach underlying these rules is to determine the changes made to the original domain (adding, deleting, and modifying commands and associated rules) and from there to determine which parts of the domain are affected and therefore should be regression tested. Regression testing rules define the regression testing subdomain based on the types of changes. (von Mayrhauser et al., 1994a) shows how to automate this for *Sleuth*.

With UCPOP, testers guide test case generation by loading or excluding portions of the operator set during the planner's initialization. For instance, the experimental subdomain focused on "moving" tapes within the tape library. By including or excluding certain "move" operators, we could change test case generation, produce different command sequences for similar goals, or focus on specific types of tape movement.

In UCPOP, the domain (the list of operators) is part of every problem definition. Thus, the domain can be changed dynamically to reflect desired subdomains. For example, we can search the knowledge base for which of the move operators require the destination and source LSMs to be equal and restrict the domain to only those move commands. Because the preprocessor is not given the information about the subdomain, it may be unable to generate test cases from the subdomain; in this case, it returns the best test case possible, which will be partially instantiated (i.e., some of the parameters will have no value) and will be incomplete (i.e., it will not include all the commands necessary to complete the desired goal).

7.3. Partition Testing

Partition testing (Goodenough and Gerhart, 1975), (Hamlet and Taylor, 1990) includes all testing criteria that define equivalence classes of input values with the proviso that a set of tests is adequate if it contains tests with values from each of the equivalence classes. How equivalence classes are developed varies: white box testing criteria may define equivalence classes through branch or dataflow criteria. A black box criterion example is category partition testing (Ostrand and Balcer, 1988).

In *Sleuth*, parameter values can be grouped into sets, from which values may be selected. This provides an obvious mechanism for defining partitions. Rules regulate selecting specific sets (and thus values from them).

In the planner version, the preprocessor uses the parameter sets to generate the initial state description. If the parameter sets are changed, then the preprocessor generates different initial conditions, which conform to the testing criteria of the partition. For example, we may have three partitions of tape ids: legally named valid tapes, legally named invalid tapes and illegally named tapes. If we wish to generate test cases that move each type, then we can generate three problem descriptions with the preprocessor, each relying on a different set of tape ids, and then generate test cases for each problem description. Figure 15 lists an initial state description for each of the three types of tape ids; the primary difference in the test cases is the tape ids referenced.

7.4. Other Testing Strategies

On occasion, testers want to test all values of certain sets of parameter values. They may want to do this in order or according to some random permutation of the values. In *Sleuth*, value selection is currently random, but could easily be enhanced to include capabilities such as Maurer's probabilistic grammars (Maurer, 1990). Even so, setting the number of commands to be generated high enough will guarantee that all parameter values will be generated. While this will not be a minimal set (in terms of number of commands generated), it is sufficient. The random selection also assures that it will not be pathologically large.

As with partitioning, the preprocessor can step through parameter values as easily as through parameter sets. In each case, the top level function in the preprocessor (which is called "generate-problem") is called with modified values for the parameter sets; at present, these sets are stored as global variables. In the case of partitioning, a whole new set is substituted; for parameter value coverage, the code

	Legal, Valid	Legal, Invalid	Illegal, invalid
Init	((FULL SLEV) (LOC 0) (ON 0) (CAP 0 DRAINED) (LOC 1) (OFF 1) (CAP 1 DRAINED) (LOC 10) (ON 10) (CAP 10 DRAINED) (CONNECT 0 1) (LOC ACS) (TAPE EVT185) (IN EVT185 1 10 5 13))	((FULL SLEV) (LOC 0) (OFF 0) (CAP 0 ENTERING) (LOC 1) (OFF 1) (CAP 1 ENTERING) (LOC 10) (ON 10) (CAP 10 DRAINED) (CONNECT 0 1) (LOC ACS) (TAPE SCR185) (OUT SCR185 ACS) (FROM SCR185 ACS UNKNOWN UNKNOWN UNKNOWN))	((FULL SLEV) (LOC 0) (ON 0) (CAP 0 DRAINED) (LOC 1) (ON 1) (CAP 1 ENTERING) (LOC 10) (OFF 10) (CAP 10 DRAINED) (CONNECT 0 1) (LOC ACS) (TAPE ABC) (IN ABC 0 4 2 3))
Goal	(FROM EVT185 1 10 5 13 1 19 unknown unknown)	(FROM SCR185 0 unknown unknown unknown 10 unknown unknown unknown)	(FROM ABC 0 4 2 3 10 4 unknown unknown)

Figure 15. Partition Testing Example: partitioning cases by tape values

iterates over the set of values forcing a single one to be used in each test case that is generated.

8. Future Work

The planning based test case generator described in this paper is just a prototype, intended to assess whether the planning paradigm could generate interesting test sequences under a variety of test criteria. Our explorations indicate that the domain knowledge about testing is naturally represented in a planner and that the planner's representation and reasoning is flexible enough to support different testing criteria.

However, a number of questions remain about the long term viability of planning as a platform for automated test case generation. Can the planner be scaled up to larger test cases and a larger domain model? What extensions are necessary for this to be a useful tool for testers? What constitutes reasonable testing goals and should goal synthesis be automated as well?

On the question of scale up, we have already encountered difficulties in producing long test cases using the UCPOP planner. The planner is designed to be correct and complete, which means that if a solution is possible, UCPOP will find it *eventually*. Because it was designed as a theoretically and pedagogically interesting tool, its built-in search strategy is not efficient. We have alleviated that problem to some extent by incorporating more efficient general search strategies (Srinivasan, 1995), but these changes are not enough. We will explore two possibilities for expediting scale-up: including domain specific search strategies in UCPOP and transferring the knowledge base to another planner. Some preliminary explorations suggest that UCPOP is occasionally going down garden paths in the search space for this domain; domain specific strategies should eliminate those pathologies. If these strategies are inadequate for scale-up, then we will port the knowledge base to a more efficient planner which is, however, harder to program. The simplicity of UCPOP is attractive for expediting use by potential users.

In parallel with the search strategy effort, we will be implementing the full domain. Because the subset includes examples of all types of representation for the domain, we anticipate little problem with adding to the knowledge base.

On the question of encapsulating the planning version into a useful tool, the primary lack is a user interface. At present, all interactions are through a set of Lisp functions, useful for the developers who need to exert control, but awkward for any users. We will be building a menu and query driven interface in CLIM (Common Lisp Interface Manager) for the system. The composition of the interface (i.e., the options for the user) will depend in part on determining how user's testing goals should be accommodated.

The primary research question remaining is representing and automating goal generation. Our approach has been passive, requiring the user to indicate what should be done, and command oriented, requiring goals specified in terms of number and types of commands. Many other testing goals might be incorporated as well: focused testing of suspect parts of the system, coverage of parameter or command sets, or testing for interaction effects between sets of commands. These higher level goals could form the basis for strategic reasoning about test suites; at present, we reason only about single test cases. Generating criteria for test suites can be viewed as a higher level planning or search problem, making it conducive to solution by a variety of AI techniques (e.g., hierarchical planning, heuristic search or active learning). We need to enumerate the desired testing goals, acquire knowledge about when goal types are most appropriate and how test suite goals are manifest in individual test cases and incorporate test suite goal reasoning into the system and its interface.

Automated planning systems offer several potential advantages for test case generation. First, ordering the operations in the test case and checking that the order is correct is handled automatically by the planning system. Second, the representation is natural for describing commands and their interactions, information that is necessary for developing test cases. Third, the flexibility of describing new initial states and goal states makes it amenable to generating many different test cases for the same system. Finally, the full description required by the planner ensures that only correct cases will be generated. Conversely, should illegal test cases be desired, they can be generated relatively easily by "mutilating" the operator descriptions (e.g., removing parts of the preconditions or effects).

Planning for testing shows promise as a new method for automatically generating test cases. We have demonstrated its value on a realistic subset of an industrial testing domain. What remains is extending the method to larger sets and to test suites.

Acknowledgments

This research was partially supported by the Colorado Advanced Software Institute (CASI), StorageTek, the Air Force Institute of Technology, the CRA Mentoring Project, a Colorado State University Diversity Career Enhancement grant, and a National Science Foundation Research Initiation Award #RIA IRI-9308573. 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. The authors thank Li Li for implementing the system while working as a summer research assistant and anonymous reviewers for the 1995 Knowledge Based Software Engineering Conference for their pointers and comments.

Notes

1. This problem has been alleviated in UCPOP 4.0

References

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 (to appear).

John S. Anderson. *Automating Requirements Engineering Using Artificial Intelligence Techniques.* PhD thesis, Dept. of Computer and Information Science, University of Oregon, Dec. 1993.

Anthony Barrett, Keith Golden, Scott Penberthy, and Daniel Weld. *UCPOP User's Manual.* Dept of Computer Science and Engineering, University of Washington, Seattle, WA, October 1993. TR 93-09-06.

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

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

Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability: Volume I, Concepts and Models*, ACM Press, Fronier Series, 1989.

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

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.

John J. Chilenski and Philip H. Newcomb. "Formal Specification Tools for Test Coverage Analysis", *Procs. Ninth Knowledge-Based Software Engineering Conference*, September 1994, Monterey, CA, pp. 59-68.

Paul R. Cohen and Edward A. Feigenbaum. *Handbook of Artificial Intelligence*, volume 3, chapter Planning and Problem Solving, pages 513–562. William Kaufmann, Inc., Los Angeles, 1982.

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

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

W. Deason, D. Brown, K.-H. Chang, and J. Cross. "Rule-Based Software Test Data Generator", *IEEE Transactions on Knowledge and Data Engineering*, 3(1), March 1991, pp. 108-117.

Stephen Fickas and John Anderson. A proposed perspective shift: Viewing specification design as a planning problem. Department of Computer and Information Science CIS-TR-88-15, University of Oregon, Eugene, OR, November 1988.

Stephen Fickas and B. Robert Helm. "Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Transactions on Software Engineering*, SE-18(6), June 1992, pp. 470-482.

Tom Figliulo, Anneliese von Mayrhauser, and Richard Karcich. "Experiences with Automated System Testing and SLEUTH", *Procs. IEEE Aerospace Applications Conference 1996*, February 1996 (to appear).

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.

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.

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

James W. Hooper and Rowena O. Chester. *Software Reuse: Guidelines and Methods*, Plenum Publishers, 1991.

Karen Huff and Victor Lesser. A plan-based intelligent assistant that supports the software development process. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Nov. 1988.

Karen Huff. Software adaptation. In *Working Notes of AAAI-92 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, pages 63–66, Stanford University, March 1992.

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

P. Maurer. "Generating Test Data with Enhanced Context-Free Grammars", *IEEE Software*, July 1990, pp. 50-55.

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

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

J.S. Penberthy and D. Weld. "UCPOP: A sound, complete, partial order planner for ADL", In *Proceedings Third International Conference on Principles of Knowledge Representation and Reasoning*, October 1992, pp. 103-114.

P. Purdom. "A Sentence Generator for Testing Parsers", *BIT*, 12(3), 1972, pp. 366-375.

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.

Robert S. Rist. Plan Identification and Re-use in Programs. In *Working Notes of AAAI-92 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, pages 67–72, Stanford University, March 1992.

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

Raghavan Srinivasan and Adele E. Howe. Comparison of Methods for Improving Search Efficiency in a Partial-Order Planner. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1620–1626, Montreal, Canada, August 1995.

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

Gerald A. Sussman. A computational model of skill acquisition. Technical Report Memo no. AI-TR-297, MIT AI Lab, 1973.

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.

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

Anneliese von Mayrhauser, Richard T. Mraz, and Jeff Walls. "Domain Based Regression Testing," *Proceedings of the International Conference on Software Maintenance*, Sept 1994, p. 26-35.

Anneliese von Mayrhauser, Richard Mraz, Jeff Walls, and Pete Ocken. "Domain Based Testing: Increasing Test Case Reuse," *Proc. of the International Conference on Computer Design*, October 1994, p. 484-491.

Anneliese von Mayrhauser, Jeff Walls, and Richard Mraz, "Testing Applications Using Domain Based Testing and *Sleuth*," *Proceedings of the Fifth International Software Reliability Engineering Conference*, November 1994, p. 206-215.

Anneliese von Mayrhauser, Jeff Walls, and Richard Mraz. "*Sleuth*: A Domain Based Testing Tool," *Proc. of the International Test Conference*, October 1994, p. 840-849.

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

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