System Testing with an AI Planner
Research Paper

Richard T. Mraz
mraz@cs.colostate.edu

Adele Howe
howe@cs.colostate.edu

Anneliese von Mayrhauser
avm@cs.colostate.edu

Li Li
Hunter College

Dept. of Computer Science
Colorado State University

Abstract

System testing of software with command language interfaces can be automated using grammar based test generation or through generating tests from an application domain specification. When viewing test case generation as constructing a sequence of commands to achieve a testing goal, AI planning systems appear very promising. This paper reports on automated test generation using an AI planning system and compares the results to tests generated by *Sleuth*, a tool for automated application domain testing.

1 Introduction

Large commercial software systems must be tested extensively before being released to customers. The process of generating and executing test sequences currently involves programmers deriving test cases based on their experience with the software, a process which can be costly and error prone. Alternatively, we can automatically generate these sequences. Automated test generation can significantly affect the success of a software product by reducing the time required to test the software and by increasing the reliability of the product in the field through systematic generation of the sequences.

Approaches to automatic test generation include grammar based test generation and domain based testing. Alternatively, if we view test case generation as constructing a sequence of commands to achieve some goal (i.e., to test for correct behavior), then the problem is exactly that addressed by Artificial Intelligence Planning systems [4]; such systems generate sequences of operations, propagating constraints between different levels of abstraction and checking for possible harmful interactions. So, given knowledge about the target software and its expected input/output behavior, an AI planning system can be used to generate test cases from a test domain model.

To validate this hypothesis, we used the application domain model of an existing automatic test generation system, *Sleuth* ([12, 15, 16]), as starting point for the AI Planning system and compared the kinds of cases generated by the Planning system with the domain based testing system. The major objective of this project was to determine whether AI planning provides an easy and natural method of generating test sequences. As an aside, we also wanted to see whether the AI planner would come up with test cases that were unusual or would a need more detailed specification of test intent as compared to a test generation method using application domain information ([12, 15, 16]), or command language specification [2, 3].

Section 1 presents related work and provides background on domain based testing. Section 2 describes application domain testing [15] and *Sleuth*. This includes a description of domain analysis and the resulting domain model. Test criteria transform the domain model into a test subdomain which forms the basis of test generation. We also describe the test generation process and the mechanisms used. Section 3 explains the AI planning concepts relevant to test generation, particularly how to represent goals and commands, and how the AI planner reaches a goal, in our case a sequence of commands. Section 4 applies these principles to test generation. We describe domain, test criteria, and test subdomain as goals in an AI planner. Section 5 reports on the results of using the AI planner and compares them to tests generated with *Sleuth* for comparable test subdomains. We evaluate the results in terms of domain coverage, nature of the test cases, and flexibility of test criteria. Section 6 summarizes our conclusions.

2 Background

For systems with a command language interface, system tests consist of sequences of commands that test the system for correct behavior. To automate this, we can represent each command using a grammar, generate commands from the grammar, and run the list of commands as the test case. When using grammars for test case generation, we also need to address command language semantics [2, 3, 5]. [6, 12] used attribute grammars for test case generation. Syntax and semantics of the command language were encoded as grammar productions and test case generation is a single stage algorithm. This poses several difficulties [15], 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 in terms of successive levels of abstraction.

[12, 15, 16] developed a test generation method that addresses the need of software testers for a tool that supports their thought process. Test generation addresses three levels of abstraction of testing software: 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. In its industrial application on a robot controlled tape library, the approach has been very successful and well received. *Sleuth*, the tool for this testing approach, uses a variety of methods to implement the three tiered approach to test case generation. A source of complexity is the coordination of each level's rules and

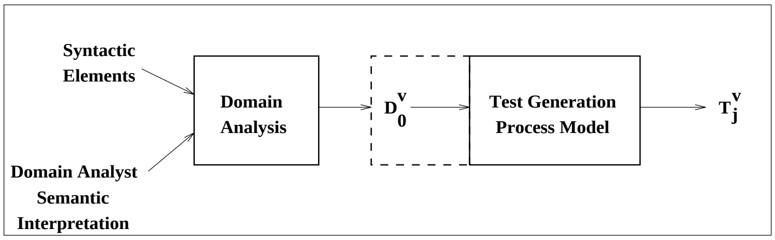

Figure 1: Top Level Domain Analysis IPO Diagram

their possible interaction. Consequently, we explored AI Planning as a natural method of addressing interaction effects.

Using an AI Planner, as opposed to a command language specification method for test data generation, exhibits fundamental differences. AI planners are declarative, stating goals that need to be reached, and matching them. 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.

In our experiments with an AI Planner, we used a subdomain of the StorageTek Robot Tape Library [10]. The Automated Cartridge System (ACS) stores 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 outer and inner panels. New tapes are entered through a special door called a Cartridge Access Port (CAP). One ACS supports up to 16 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. The language consists of 28 commands and 45 parameters. 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.

3 Application Domain Based Testing

Figure 1 presents a high level view of Application Domain Based Testing. Syntax and semantic information is used during domain analysis to construct a domain model, D_0^v for version v of the software under test. The domain model specifies all information needed for test generation. The results of test generation are *test suites*, T_j^v ($j = 1, 2, \ldots, n$; where n = total number of test). For details of the domain analysis and domain model see [15, 16]. *Sleuth* is the first tool to generate test cases based on a domain model. It uses the hybrid representation as shown in Table 1.

Domain Model Component	Hybrid Representation
Script Classes	Sets of Command Names
Script Sequencing Rules	Macro Expansion
Script Parameter Binding	Marco 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

Table 1: Domain Model Components and Hybrid Representation

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

Table 2: Commands and Script Classes for the StorageTek Experimental Subdomain

3.1 Script Representation

Scripts capture dynamic behavior of the system under test. We represent three categories: Script Classes, Script Rules, and Script Parameter Binding. A scripting class is a set of command names with similar functionality. The notion of 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 2 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.

Script Rules define command sequencing information. In the hybrid representation, 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`. Once command sequencing has been defined, *parameter binding* makes sure the parameters in the command sequence are meaningful. For instance, parameter binding information ensure that the tape that is "mounted" is the same tape that is "dismounted."

3.2 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. The hybrid representation uses syntax diagrams for each command and a random-walk through the syntax creates an instance of a command.

Three types of semantics rules are defined at the command level: Preconditions, Postconditions, and Intracommand Rules. *Preconditions* identify the conditions that must hold before the command can execute. *Postconditions* list the conditions that are true after the command executes. For example, before entering a tape through a Cartridge Access Port (CAP), the Library Storage Module (LSM) must be `Online`. After the tape is entered into the LSM, the CAP is set to the `Entering` state and the tape location is updated. Preconditions and postconditions are implicitly represented in the hybrid representation. This information is handled in the test generation process and the test generation tool. Intracommand rules identify constraints placed on parameter value selection within a single command. For example, StorageTek's automated tape library requires that if moving tapes within the same LSM, then the source and destination panels must be different. This is represented in the domain model in first-order logic : `(lsm$1=lsm$2) => (panel$1` $\neq$ `panel$2)`.

3.3 Parameter Representation

Parameters of the command language are represented as Parameter Value Sets. Using set operations 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. These constraints are defined because the physical layout of an LSM can vary. Parameter constraint rules are represented using a "parameter" hierarchy that denotes relationships between "parameters" and set operations to modify parameter value sets.

3.4 Test Generation Process Model

Figure 2 shows a detailed view of the test generation process. The Test Generation Process Model defines how the domain model is used for test generation and it shows how test engineers use the domain model to specify test subdomains, define test criteria, and to generate test cases.

The domain model, D_0^v, captures the syntax and semantics of the system under test. It is a persistent view of the system because it represents the default conditions to generate test suites. 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

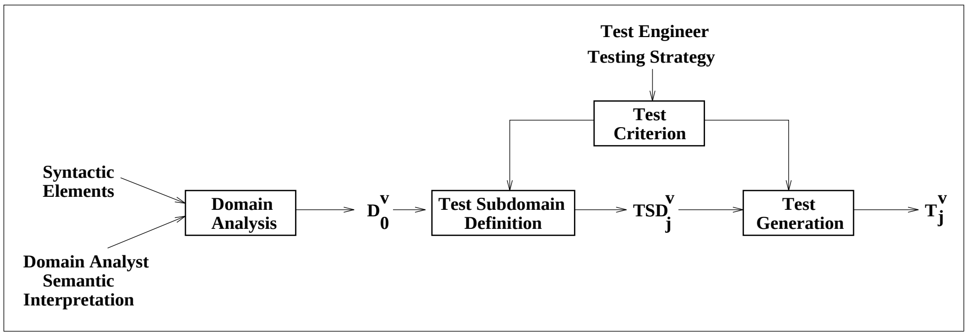

Figure 2: Test Generation Process Model

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. *Sleuth* provides each tester with a "workspace" to create test subdomains. This leaves the domain model intact while allowing multiple test engineers simultaneous access to the test generation tool. *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, and test scripts are lists of command names.

3.5 *Sleuth*

Sleuth is an automated test generation tool developed at Colorado State University. *Sleuth* supports Domain Based Testing by providing tools and utilities for test generation. The main window models the test generation process (see Figure 3). *Sleuth* provides utilities to create domain models, configure test subdomains, and to generate tests. *Sleuth* uses the hybrid domain model representation and a three stage test generation sequence. 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. This provides maximum reuse of commands at all levels of abstraction [14], and simplifies uniform testing across configurations and versions. It also supports regression testing [13].

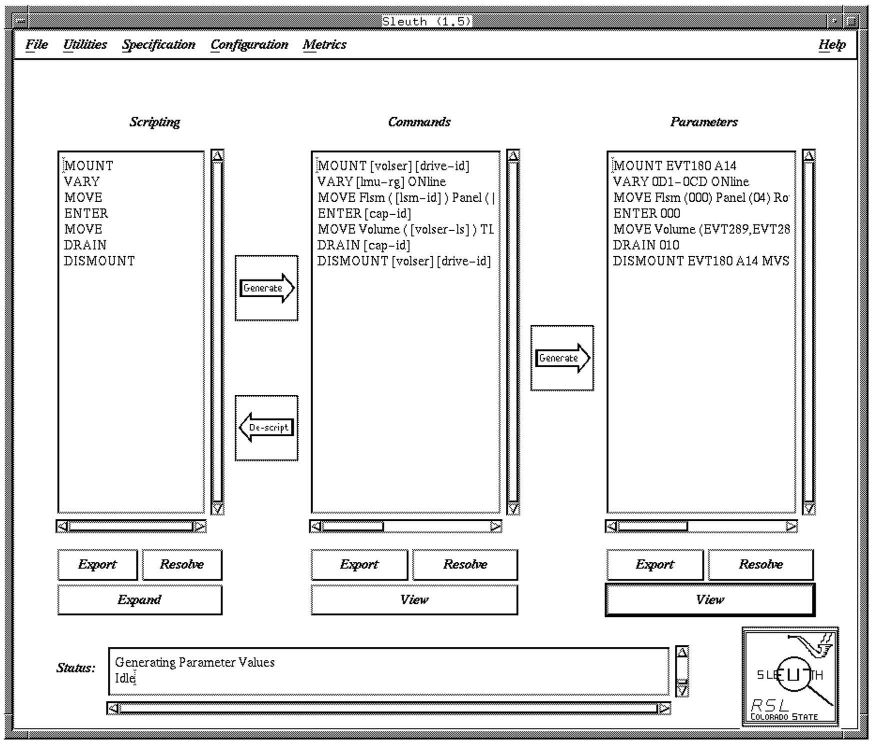

Figure 3: Window Based Test Generation Tool

4 AI Planning Applied to Testing

In the field of Artificial Intelligence, planning is the process of generating a sequence of actions to satisfy some goal or goals. From its early days, planning was thought of as a type of automatic programming [11]. Recent uses of planning in software engineering are as a representation for specifications [7] and for supporting software reuse [8].

What makes planning an attractive paradigm for software engineering applications is its emphasis on goals and the similarity of plans to programs. The emphasis on goals means that sequences of actions (e.g., plans, programs, or test cases) are generated specifically to fulfill some purpose) and that it is relatively easy to generate different plans for different goals. For example, in test case generation, 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.

Plans are a constrained form of program. Like programs, plans are developed to achieve a solution/goal; for generating test cases, the plan is the test case and the goal is some end state or a combination of states and operations that the developer wished to test. For example in the StorageTek robot tape library domain, our goal may be to move every tape in the silo or generate five move commands interleaved with five requests for information.

Like programs, plans are a representation of a procedure, sequences of operators that could be executed, but unlike programs, the operator descriptions (internal information used to develop the plans) are declarative. Declarative representations clarify the relationships between the actions, making it easier to determine when operators build on each other or conflict toward satisfying goals. For example, by specifying the commands that can be issued to the StorageTek device as planning operators, we are forced to make transparent the effects of every command in such a way that one could easily see how one command influences the execution of others.

Unlike programs, plans typically do not include control structures and a new plan is generated for different input. For test case generation, this means that we generate a new plan for every goal and for every initial state of the system that should be tested.

Basically, in order to generate a plan, a planning system is given: (1) a description of the operators (i.e., commands for the robot tape library), (2) an initial state of the world (i.e., the system being tested) and (3) a goal state (i.e., what should be tested). Operator descriptions have parameters (what objects are involved in the operator), preconditions (what must be the case to use this operator) and effects (what happens to the system after the operator is executed). For example, the planning description of the Mount command appears in Figure 4. The parameters of the command are: a variable that must be bound to a location or LSM and a set of variables specifying the address of the tape. The preconditions of the command are: the service level of the robot should be `full`, a tape should be `in` an LSM at some yet to be specified location (any variable proceeded by a "?" is one determined when the plan is generated), and the LSM should be `on`. If we execute the command in the robot library, then the tape is now `at` a tape drive (designated by `?m_did`) and the tape is no longer `in` the LSM. The initial state corresponds to what tapes are in the library, where the tapes are and what readiness state describes the robot (an example initial state appears in Figure 7). As mentioned, the goal can refer to the operations executed, the final state of the robot library at the end of the plan or a combination of these possibilities; Figure 7 includes a goal requiring a tape movement. Constraints on the operators and problem state are not represented explicitly, but rather are a result of the precondition and effect descriptions.

Most 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. For example, if our goal is to move all tapes in the library, then the system will find operators one at a time that move a tape, placing the preconditions of the move operators onto the goal list, and then will find operators that satisfy the preconditions; finally, all the actions will be ordered to make sure that preconditions are satisfied before operators are executed.

Because the generation of the plan is based on a proof, the operator 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 then are mostly handled by the planning system's manipulation of the operators, but may be tuned through the use of control rules that direct selection of goals and operators.

We implemented experimental subdomain for the StorageTek robot tape library problem

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

Table 3: Domain Model Components and AI Planner Representation

in the UCPOP planner [1]. 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., move *all* tapes) and that it does not order the sequence of operators in the plan until necessary, which makes it more flexible. Operators are represented like the ones in Figure 4. Figure 8 shows an example of a test case generated by UCPOP. Parameters are incorporated into the state and operator descriptions. Thus, UCPOP fills in the parameters as it is generating the test case.

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).

5 AI Planner Representation of DBT

We implemented the experimental subdomain (of section 3.1) in the UCPOP planning system to show how to use an AI Planner for test case generation. The subset includes all domain model components. Table 3 lists each domain model component and its AI planning representation. Each one is described below.

```
;;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 back 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 4: Planning representations of the Mount and Dismount commands

5.1 Script Representation

A script class is a set of commands with similar functionality. This is represented as a collection of planner operators. Testers guide test case generation by loading or excluding these collections 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.

Script sequencing rules and parameter binding make sure commands are issued in the correct order. The AI planning representation includes this information in the preconditions and effects of the planner operators. The `MOUNT-DISMOUNT` sequencing rule is implemented with the planner operators listed in Figure 4. Preconditions for the `DISMOUNT` require a tape to be loaded in a tape drive. A tape can be placed into a drive in one of two ways: (1) The tape drive can be loaded as part of the initial state, or (2) the tape can be loaded as an *effect* of the `MOUNT` operator.

5.2 Command Representation

The next domain model component is the command language syntax. The AI planner does not explicitly use the syntax of the command language. Instead, each path through a command is represented as a separate planning operator. A postprocessor translates the planner output into the correct syntax. For instance, the StorageTek command language uses the `SRVLEV` command to "toggle" the system's service level: `service-level-cmd ::=`

```
;;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 sevicetobase)
           :precondition (full slev)
           :effect (and (not (full slev)) (base slev)))
```

Figure 5: Example planning representations of the command language syntax: changing service level

SRVLEV {BASE | FULL}. As seen in Figure 5, two operators encode the command for the planner.

The next three domain model components are command preconditions, command postconditions, and command intracommand rules. All three are represented as preconditions and effects to planner operators. A command language precondition denotes sequencing information based on 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. Intracommand rules are also specified as preconditions to planner operators. These preconditions check parameter values within the command. Figure 6 shows one version of the MOVE command. This operator encodes the intracommand rule: `if (lsm$1=lsm$2) => (panel$1` $\neq$ `panel$2)` by requiring `slsm` (source LSM) and `dlsm` to be equal (`eq`) and `sp` (source panel) and `dp` (destination panel) to be not equal (`neq`).

5.3 Parameter Representation

The last domain model component represents command language parameters and parameter constraints. The AI planner uses two preprocessors to capture this information: Initial State Generator and Goal Generator. The initial state generator randomly creates an initial state vector for the planning system. This state vector uses information about parameter constraints to make sure the state is valid. 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 7 shows an example of an initial state.

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

```
;;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 6: Example planning representation of the command preconditions, postconditions and intracommand rules: move a volume

```
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 7: Example planning representation of objects, object elements and parameter constraints: initial state list and goal list

or to mount-dismount a single tape. If more than one move or mount-dismount is needed, a conjunctive goal is created. An example of a compound goal is shown in Figure 7. 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])
```

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 Results

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

6.1 Model Development and Test Generation

At the most basic level, we were able to use the AI planning system to generate a variety of test cases in the experimental subdomain similar to that of the existing domain based testing system. Additionally, the AI planning system is able to configure a test subdomain similar to the hybrid representation. By "configure", we mean that the test domain can be subdivided or changed to generate constrained (e.g., illegal) or restricted (e.g., just use one type of command) types of tests. For instance, script rules and script parameter binding can be altered by removing ("mutilating") precondition lists and by changing relational operators, respectively. Intracommand rules can be eliminated by removing parameter binding preconditions or by altering relational operators. Parameter value selection can be adjusted by modifying the preprocessors. Finally, command syntax can be altered by changing the postprocessor.

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 a sophisticated user interface. Despite the small code size in the planner, computation time for a test case could take from minutes to hours. This was not a surprise since the planner was developed as a prototype and no efforts were made to improve its efficiency. Additionally, UCPOP was designed for easy-of-use and theoretical soundness.

Let's look at the results of running UCPOP. Figure 8 shows the results from running the UCPOP Planner given the goal of moving a tape from one LSM to another. The initial state shows that the tape is located in LSM 000 (`(IN EVT280 0 UNKNOWN UNKNOWN UNKNOWN)`) and both the source and destination LSMs are offline (`(OFF 0) (OFF 1)`). In the StorageTek system, it may not be possible to know all of the state information for particular entities. This is indicated with the `UNKNOWN` place holder.

UCPOP's solution is listed in step 1-3, and the postprocessor output is listed last. The postprocessor examines each step and generates the the correct syntax for the StorageTek Robot Tape Library.

6.2 Domain Coverage Comparison

We compared the domain coverage of the planning and hybrid (i.e., *Sleuth*) implementations by considering how each represented the seven levels of the domain model (as appear in Table 1). Both the hybrid and planner representations employed similar mechanisms to implement Script Classes. The hybrid 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 hybrid 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. The final test case is generated using a postprocessor to translate the UCPOP output into the appropriate StorageTek syntax. The hybrid approach stores each command as a syntax diagram. *Sleuth* takes a random-walk through the syntax diagram to create a command template.

Command Preconditions and Postconditions are not explicitly represented in the hybrid representation. Instead, testers must issue set up commands or execute a list of commands to put the system in the correct state. The planner was able to represent pre/post conditions in each planning system operator. The trade off between these two representation is in the amount of state information required during test generation.

Initial:	((FULL SLEV) (LOC 0) (OFF 0) (CAP 0 DRAINED) (LOC 1) (OFF 1) (CAP 1 ENTERING) (LOC 10) (ON 10) (CAP 10 DRAINED) (CONNECT 0 1) (LOC ACS) (TAPE EVT280) (IN EVT280 0 UNKNOWN UNKNOWN UNKNOWN))
Step 1:	(MODIFYTOON 0) Created 3
	0 -> (FULL SLEV)
	0 -> (OFF 0)
Step 2:	(MODIFYTOON 1) Created 2
	0 -> (FULL SLEV)
	0 -> (OFF 1)
Step 3:	(MOVEONE EVT280 1 0 UNKNOWN UNKNOWN UNKNOWN UNKNOWN UNKNOWN UNKNOWN) Created 1
	3 -> (ON 0)
	2 -> (ON 1)
	0 -> (FULL SLEV)
	0 -> (CONNECT 0 1)
	0 -> (IN EVT280 0 UNKNOWN UNKNOWN UNKNOWN)
	0 -> (TAPE EVT280)
	0 -> (LOC 1)
	0 -> (LOC 0)
Goal:	(FROM EVT280 0 UNKNOWN UNKNOWN UNKNOWN 1 UNKNOWN UNKNOWN UNKNOWN)
Postprocessor:	
step 1 is:	(MODIFY 000 ONLINE)
step 2 is:	(MODIFY 001 ONLINE)
step 3 is:	(MOVE VOLUME(EVT280) TLSM(001))

Figure 8: Example Results from UCPOP : Goal = Move Tape EVT280 from LSM 000 to LSM 001

Parameter value sets and parameter constraints are handled differently in both domain model representations. The hybrid approach 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 two preprocessors: Initial State Generator and Goal Generator. Both use information about default parameter values and parameter constraint rules to define initial states and goals that do not violate the parameter selection rules.

6.3 Nature of Tests

One of the most interesting aspects of these results were the differences between the planning approach and the hybrid approach to test data generation. Testers take different views of the problem during test generation. Using the hybrid approach, 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.

To demonstrate the differences consider the test in Figure 4. The objective was to move a single tape. *Sleuth* was directed to randomly generate four commands; the test case was selected when it included 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 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.

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

	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`	

Table 4: Comparing *Sleuth* and UCPOP Tests

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 Conclusions

An AI planning system has been shown to be an effective approach to test case generation. To extend the results, the current prototype must be enhanced in several ways. First, a more efficient implementation of the planner can be created in one of two ways. Domain specific search rules can be added to control UCPOP's traversal of the search space of test cases, or we can replace UCPOP with a more efficient planner.

Second, we plan to continue this work by extending the test domain for the planning system. The complete StorageTek command language can be encoded as planning system operators. We can extend the current experimental subdomain to create "illegal" or "invalid" test cases by automating the "mutilation" process described above. Additionally, the planner system can be designed to create tests in a three stages by representing operators hierarchically. This will give us an opportunity to more fully compare staged test case generation in the planning system with the hybrid method.

Finally, the differences in the two approaches highlights an open research question. How much state information is needed by an offline test generator? At one extreme, *Sleuth* does not represent state information. At the other extreme, the planning system needs as much state as possible. Test cases generated from the planning system are provably correct based on the initial state, the goal, and the operators. The planner representation can also capture hardware connections in the system under test through state information. Some AI planning systems, such as UCPOP, assume that all effects of operators are known. Unfortunately, in a systems like the StorageTek tape library, the effects may be conditional on unknown factors, probabilistic, or simply unpredictable. For example, in the StorageTek `MOVE` command, the initial state of a tape may be completely defined by its location in an LSM, panel, row, and column; However, some instances of the `MOVE` command identify only the destination LSM, but not the destination panel, row, or column. So, the effect of such `MOVE` commands is unknown state information. We will investigate methods for planning under uncertainty to address this problem.

Based on our comparison, an AI planning system appears to be a natural alternative for generating certain types of test cases. There are trade offs between the planner and the hybrid representation. *Sleuth* does not use state information. Its three staged test generation sequence is useful for testers, test case recall, and test case reuse. When using *Sleuth*, testers focus on what commands are included in the test case. In contrast, the planner guarantees correct test cases, creates innovative and unusual test sequences, focuses testers on the goals of test case generation leaving the planner to choose the appropriate commands. Consequently, AI planning based test generation may be most appropriate for generating innovative, goal oriented test cases in command language systems.

Acknowledgements

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.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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. American Association for Artificial Intelligence.

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

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

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

[12] 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.

[13] Anneliese von Mayrhauser, Richard T. Mraz, and Jeff Walls. "Domain Based Regression Testing," (To be published) *Proceedings of the International Conference on Software Maintenance*, Sept 1994.

[14] Anneliese von Mayrhauser, Richard Mraz, Jeff Walls, and Pete Ocken. "Domain Based Testing: Increasing Test Case Reuse," (To be published) *Proc. of the International Conference on Computer Design*, October 1994.

[15] Anneliese von Mayrhauser, Jeff Walls, and Richard Mraz, "Testing Applications Using Domain Based Testing and *Sleuth*," (To be published) *Proceedings of the Fifth International Software Reliability Engineering Conference*, November 1994.

[16] Anneliese von Mayrhauser, Jeff Walls, and Richard Mraz. "*Sleuth*: A Domain Based Testing Tool," (To be published) *Proc. of the International Conference on Computer Design*, October 1994.