Using Formal Methods To Derive Test Frames In Category-Partition Testing

Paul Ammann * Jeff Offutt †

pammann@isse.gmu.edu ofut@isse.gmu.edu
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030

Abstract

Testing is a standard method of assuring that software performs as intended. In this paper, we extend the category-partition *method, which is a specification-based testing method. An important aspect of category-partition testing is the construction of test specifications as an intermediate between functional specifications and actual tests. We define a minimal coverage criterion for category-partition test specifications, identify a mechanical process to produce a test specification that satisfies the criterion, and discuss the problem of resolving infeasible combinations of choices for categories. Our method uses formal schema-based functional specifications and is shown to be feasible with an example study of a simple file system.*

1 Introduction

Testing software is a standard, though imperfect, method of assuring software quality. The general aim of the research reflected in this paper is to formalize, and mechanize where possible, routine aspects of testing. Such formalization has two benefits. First, it makes it easier to analyze a given test set to ensure that it satisfies a specified coverage criterion. Second, it frees the test engineer to concentrate on less formalizable, and often more interesting tests.

Specification-based testing, or black-box testing, relies on properties of the software that are captured in the functional specification, as opposed to the source code. The category-partition method [BHO89, OB88] is a specification-based test method that has received considerable attention. An important contribution of category-partition testing is the formalization of the notion of a *test specification*, which is an intermediate document designed to bridge the large gap between functional specifications and actual test cases. Some parts of a test specification can be derived mechanically from a functional specification. Other parts require the test engineer to make decisions or rely on experience. We wish to mechanize as many tasks as possible.

In this paper, we address some important aspects of constructing a test specification that are left open in the category-partition method. Specifically, we

- define a minimal coverage criterion for category-partition testing, and supply a general procedure for enumerating tests that satisfy the criterion
- supply a procedure for deriving test cases from test scripts
- supply a method of identifying and eliminating infeasible tests caused by conflicting choices.

A side effect of our method is that we are sometimes able to uncover anomalies in the functional specification. This allows us to, in some cases, detect *unsatisfiable* (as defined by Kemmerer [Kem85]) specifications.

We employ formal methods, in particular Z specifications, as a tool in our investigation of test generation. There are several motivations for using formal methods. First, some of the analysis necessary to produce a test specification has already been done for a formal functional specification, and hence less effort is required to produce a test specification from a formal functional specification. Second, using formal methods makes the determination of whether part of a test specification results from a mechanical process or from the test engineer's judgement more objective. Finally, formal methods are well suited to describing artifacts of the testing process itself. Such artifacts include parts of the test specification and actual test cases.

*Partially supported by the National Aeronautics and Space Administration under grant NAG_1-1123-FDP.

†Partially supported by the National Science Foundation under grant CCR-93-11967.

1.1 Related Work

A variety of researchers have investigated the use of formal methods in test generation. Kemmerer suggested ways to test formal specifications for such problems as being unsatisfiable [Kem85]. In the DAISTS system of Gannon, McMullin, and Hamlet [GMH81], axioms from an algebraic specification and test points specified by a test engineer are used to specify test sets for abstract data types. Hayes [Hay86] exploits the refinement of an abstract Z specification to a (more) concrete specification to specify tests at the design level. Amla and Ammann [AA92] described a technique in which category-partition tests are partially specified by extracting information captured in Z specifications of abstract data types. Laycock [Lay92] independently derived similar results. More recently, Stocks and Carrington [SC93a, SC93b] have used formal methods to describe test artifacts, specifically test frames (sets of test inputs that satisfy a common property) and test heuristics (specific methods for testing software).

1.2 Outline of paper

In section 2, we review the steps in the category-partition method. The Z constructs that we need are described in section 3; further information may be found in the Z reference manual [Spi89] or one of the many Z textbooks [Dil90, PST91, Wor92]. Section 4 defines a minimal coverage criterion for category partition testing, gives a procedure for deriving specifications that satisfy the criterion, and defines a method to produce test scripts from the resulting test specifications. Section 5 presents partial Z specifications, test specifications, test frames, and test case scripts for an example system. Finally, we summarize our results and findings in Section 6.

2 Category-Partition Method

The category-partition method [BHO89, OB88] is a specification-based testing strategy that uses an informal functional specification to produce formal test specifications. The category-partition method offers the test engineer a general procedure for creating test specifications. The test engineer's key job is to develop *categories*, which are defined to be the major characteristics of the input domain of the function under test, and to partition each category into equivalence classes of inputs called *choices*. By definition, choices in each category must be disjoint, and together the choices in each category must cover the input domain.

The steps in the category-partition method that lead to a test specification are as follows.

1. Analyze the specification to identify the individual functional units that can be tested separately.

2. Identify the input domain, that, is the *parameters* and *environment variables* that affect the behavior of a functional unit.

3. Identify the categories, which are the significant characteristics of parameter and environment variables.

4. Partition each category into choices.

5. Specify combinations of choices to be tested, instantiate test cases by specifying actual data values for each choice, and determine the corresponding results and the changes to the environment.

This paper focuses on the last step above. Each specified combination of choices results in a *test frame*. The category-partition method relies on the test engineer to determine constraints among choices to exclude certain test frames. There are two reasons to exclude a test frame from consideration. First, the test engineer may decide that the cost of building a test script for a test frame exceeds the likely benefits of executing that test. Second, a test frame may be infeasible, in that the intersection of the specified choices is the empty set. Recently, Grochtmann and Grimm [GG93] have developed *classification trees*, a hierarchical arrangement of categories that avoids the introduction of infeasible combinations of choices.

The developers of the category-partition method have defined a test specification language called TSL [BHO89]. A test case in TSL is an operation and values for its parameters and relevant environment variables. A *test script* in TSL consists of the operations necessary to create the environmental conditions (called the SETUP portion), the test case operation, whatever command is necessary to observe the affect of the operation (VERIFY in TSL), and any exit command (CLEANUP in TSL). Test specifications written in TSL can be used to automatically generate *test scripts*. The test engineer may optionally give specific representative *values* for any given choice to aid the test generation tool in deriving specific test cases. The category-partition method supplies little explicit guidance as to which combinations of choices are desirable – the task is left mostly to the test engineer's judgement.

In this work, we follow the spirit of the category-partition method, but there are differences in our use of the technique. First, we base our derivation on formal specifications of the software, since, as has been demonstrated in a variety of papers [AA92, Lay92, SC93a, SC93b], the formality of the functional specification helps to simplify and organize the production of a test specification. Second, we do not follow the TSL syntax, but instead format examples as is convenient for our presentation. Specifically, as has been done by others [SC93a, SC93b], we employ the formal specification notation to describe aspects of the tests themselves as well as to describe functional behavior.

3 Z Specifications

Z is a model-based formal specification language based on typed set theory and logic. In this paper, we focus our attention on Z specifications for abstract data types (ADTs). An ADT is characterized by specified states and operations that observe and/or change state.

In Z, both states and operations on states are described with *schemas*. A schema has three parts; a *name*, a *signature* to define variables, and a *predicate* to constrain the variables. A schema describing an ADT state has a signature whose variables define a set-based model for the ADT and a predicate that specifies invariants on the ADT. A schema describing an ADT operation has a signature that defines the inputs, outputs, state variables from the prior state, and state variables from resulting state. The predicate of an operation schema constrains the variables with preconditions and postconditions.

By usual convention in Z, input variables are *decorated* by a trailing "?", and output variables are decorated by a trailing "!". State variables decorated by a trailing "′" indicate the state of the variable after an operation is applied. By way of abbreviation, one schema can be included in another by using the name of the included schema in the signature of the new schema. By convention, if the included schema name is prefixed with a Δ, then the new schema may change the state variables of the included schema. If an included schema name is prefixed with a Ξ, the operation may not change the state variables of the included schema. Instances of schemas are given in the examples of Section 5.

A partial mapping of Z constructs to category-partition test specifications is given by Amla and Ammann [AA92]. We briefly recap major points below:

1. Testable units correspond to operations on the ADT.

2. Parameters (inputs) are explicitly identified with a trailing "?". Environment variables (ADT state components) are the variables of the state schema.

3. Categories have a variety of sources. Some categories are derived from preconditions and type information about the inputs and state components. Typically, there are additional desirable categories that cannot be derived from the specification; the test engineer must derive these from knowledge and experience. Recent work [SC93a, SC93b] points to other sources of categories in formal specifications.

4. Some categories, particularly those that are based on preconditions, partition naturally into normal cases and unusual or illegal cases. Partitions for other categories depend on the semantics of the system, and often require the test engineer's judgement.

5. To determine which combinations of choices to test, there are few general rules to be found in either the Z specifications or in the previous work in category-partition testing [BHO89, OB88]. Supplying a minimal set of rules by defining a coverage criterion is one of the contributions of the current paper. For the verification of outputs, the state invariants and postconditions are helpful.

It has also been observed that Z schemas are ideal constructs for describing test frames [SC93a, SC93b]. In this case, the signature part lists the variables that make up possible test inputs and the predicate part constrains the variables as determined by the reason for the test. For example, a test frame intended to cover a statement in a program has a predicate part that gives the path expression that causes the flow of control to reach that statement. A Z schema used to describe a test frame typically describes a set of possible inputs; a refinement process must be used to select an element from the set before the test can actually be executed. If the set of possible inputs is empty, the test frame is infeasible.

4 Derivation of Test Scripts

The previous category-partition work does not give a prescription for which combinations of choices should

be specified as test frames. Selecting the combinations of choices used is an important problem whose solution affects the strength and efficiency of testing. In general, there is no single solution that applies in all applications, but it is nonetheless clear that some possibilities can be rejected as inadequate.

In TSL specifications [BHO89], a special syntax in the form of conditionals in the RESULTS sections is provided to specify combinations of choices. However, the TSL syntax only supports a way to specify combinations of choices. Although the authors suggest testing certain error conditions only once, deciding which choices to specify is left to the test engineer.

Using all combinations of categories is generally inefficient, and the corresponding tests are repetitious and uninteresting. If we generate all combinations, where there are N categories and the *kth* category has i_k choices, then the number of resulting test frames is:

$$\prod_{k=1}^{N} i_k, \tag{1}$$

which is combinatorial in cost. For example, consider a test specification with two categories X and Y, where X has three choices and Y has two:

Categories:

X	Y
* P_1	* Q_1
* P_2	* Q_2
* P_3	

Let us denote a test frame that satisfies choices P_i and Q_j by $[P_i;\ Q_j]$. The example specification defines six possible test frames: $[P_1;\ Q_1]$, $[P_1;\ Q_2]$, $[P_2;\ Q_1]$, $[P_2;\ Q_2]$, $[P_3;\ Q_1]$, and $[P_3;\ Q_2]$. To construct a test set, one chooses from zero to six of the possible test frames, yielding a total of 2^6 or 64 possible sets of test frames (including the empty set). Although only six decisions need to be made (whether to include each test frame), the test frames are interrelated, and so the six decisions cannot be made in isolation. Thus we are left with the question; which of these test sets should be specified?

We proceed by defining a minimal metric that *any* reasonable selection of choice combinations should satisfy. The basic principle is that each choice should be used at least once, otherwise there is no reason to have distinguished that choice from other choices in a category. Let a set of test frames that incorporates each choice at least once be defined to satisfy the *each-choice-used* criterion.

As it turns out, the each-choice-used criterion is not very useful in practice. Defining successive test frames by selecting unused choices in each category where it is possible to do so results in undesirable test sets.

The reason is that a system typically has some normal mode of operation, and that normal mode corresponds to a particular choice in each category.[1] Call the choice in each category that corresponds to the normal mode the *base* choice. It is useful for a tester to evaluate how a system behaves in modes of operation obtained by varying from the normal mode though the non-base choices in each category. We define the *base-choice-coverage* criterion to describe such a test set.

To satisfy base-choice-coverage, for each choice in a category, we combine that choice with the base choice for all other categories. This causes each non-base choice to be used at least once, and the base choices to be used several times.

The number of test frames generated to satisfy either each-choice-used or base-choice-coverage is linear in the number of choices, rather than the combinatorial number from formula 1. Thus there is no significant cost advantage of each-choice-used over base-choice-coverage, and so we adopt base-choice-coverage as our minimal coverage criterion for category partition test sets.

The exact number of test frames for base-choice-coverage criterion for N categories where the category k has i_k choices is:

$$(\sum_{k=1}^{N} i_k) - N + 1. \tag{2}$$

Note that satisfying base-choice-coverage does not mean that a test set is adequate for a particular application. In general, given a test set that satisfies base-choice coverage, the test engineer would want to add other useful test frames. Mechanically supplying a test set that satisfies base-choice-coverage frees the test engineer to concentrate on these additional tests. A strong argument can be made, however, that a test set that does not satisfy base-choice-coverage *is* inadequate. While this type of argument is not a strong as one would prefer, it is similar to the arguments supporting the use of program-based coverage criteria such as statement coverage or data-flow coverage criteria such as all-uses coverage.

4.1 The Derivation Procedure

The following procedure creates test scripts that are based on the category-partition method. This process

[1] Many systems have multiple normal modes of operation, but we consider only one normal mode for simplicity here.

gives a recipe for the last step of the method, and can be used to generate the TSL's RESULTS section.

1. Create a *combination matrix.*
   A convenient way to organize the values for the inputs and environment variables in test frames is with an N-dimensional *combination matrix*, where N is the number of categories and each dimension represents the choices of a category. The entries in the matrix are constraints that specify a test frame for the intersection of choices. This matrix is intended as a conceptual tool that helps describe our process, rather than something that will actually be constructed in practice.

   An example, explained further below, is the combination matrix for the two categories, X and Y:

		Y	
		Q_1	Q_2
X	P_1	1	
	P_2	2	3
	P_3	4	5

2. Identify a base test frame.
   For each partitioned category and each operation, the test engineer designates one choice to be the *base* choice. This is typically either a default choice, or a "normal" choice as taken from an expected user profile. The *base test frame* is constructed by selecting the base choice from each category. In the above example, we assume that $[P_1;\ Q_1]$ is the base test frame.

3. Choose other combinations as test frames.
   The combination matrix can be used to automatically choose the remaining test frames by varying over choices in each category. In effect, we start at the base cell in the matrix, and successively choose each cell in every linear direction from the base cell. In the above example, if $[P_1;\ Q_1]$ forms the base frame, then we also choose $[P_1;\ Q_2]$, $[P_2;\ Q_1]$, and $[P_3;\ Q_1]$.

4. Identify infeasible combinations.
   In the combination matrix, each cell can be annotated with a number that corresponds to a setup script, or left blank if that combination is impossible. If an impossible choice (a blank cell) is reached, we shift the test frame by varying other choices until we reach a combination of choices that is possible. Since $[P_1;\ Q_2]$ in the above example is blank, we shift the test frame by moving P_1 to P_2 to get test frame $[P_2;\ Q_2]$. Note that when we shift, we shift away from the base frame.
   Deciding whether a combination is infeasible is equivalent to deciding whether the constraints involved can be satisfied. Although satisfiability is a hard problem, tools exist that are capable of resolving many common cases. For example, theorem proving systems can handle propositional and predicate calculus, as well as simple arithmetic properties, and hence could be used as an intelligent assistant to the test engineer.

5. Refine test frames into test cases.
   Each test frame represents a combination of choices, and thus is a set of candidate test inputs for that frame. To actually execute a test, an element from the test set must be chosen. Although we do not concentrate on this aspect of test generation in this paper, we do note that it is a subject of serious inquiry. For example, DeMillo and Offutt [DO91] have developed algorithms to automatically generate test cases from constraints, and Wild *et al.* are developing techniques to employ accumulated knowledge to refine test frames [WZFC92].

6. Write operation commands.
   For each cell in the combination matrix that is chosen, the corresponding operation commands, setup commands, verify commands, and cleanup commands must be written. Although we expect that the test engineer needs to specify the actual commands, not all possible scripts are needed. Thus there is no need to enumerate the entire matrix, which is why this step comes towards the end. For many systems the cleanup command(s) are constant for the entire system and only need to be specified once, and only one verify command will be needed for each operation. Much of this step can be automated, but tools to do so are project specific rather than general purpose.

7. Create test scripts.
   Creating the actual test scripts is a straightforward process. For each chosen cell, the corresponding setup commands are chosen, the command is appended, then the verify and cleanup commands are attached. The commands can be given to an automated tool, which can combine them to automatically generate the actual test scripts, execute them, and provide the results to the tester.

The complete generation of test scripts can be done prior to the design; in fact, any time after the functional specifications are written. By reusing the pro-

cedure, it is also easy to modify the test scripts once they are created.

5 The MiStix Example

We demonstrate our procedure on an example system. `MiStix` is based on the `Unix` file system, and has been used in exercises in graduate software engineering classes at George Mason University. The `Mistix` specification is similar to, although simpler than, the `Unix` file system specification developed as a Z case study in Hayes [Hay93]. There are a total of ten operations defined in `MiStix`:

- Two operations to create and delete directories
- Two operations to create and delete files
- Two operations to copy and move files
- One operation to change the current directory
- One operation to print the full pathname of the current directory
- One operation to list files and directories
- One operation to log off

Because the complete specification is quite lengthy, we focus on one of the ten operations, *CreateDir*. The complete functional and test specifications for `Mistix` are in [AO93]. We specify `MiStix` as an ADT, beginning with a description of the base types needed. There are two types of objects in the system: files and directories. The type *Name* is used to label a simple file or directory name (for example, the `MiStix` file "foo"):

$$[Name]$$

We denote constants of type *Name* with double quoted strings, as in the example above.

Sequences of *Name* are full file or directory names (for example, the `MiStix` file "/usr/bin/foo"):

$$FullName ::= \text{seq}\ Name$$

The representation chosen here has the leaf elements at the tail of the sequence, and so, for example, the representation of "/usr/bin/foo" is the sequence $\langle usr,\ bin,\ foo \rangle$. We use the sequence manipulation functions *front*, which yields a subsequence up to the last element (e.g., $front(\langle usr,\ bin,\ foo \rangle) = \langle usr,\ bin \rangle$) and *last*, which yields the element at the end of the sequence (e.g., $last(\langle usr,\ bin,\ foo \rangle) = foo$).

5.1 State Description

The state of *FileSystem* is represented by the directories in the system (*dirs*), the files (*files*), and a current working directory (*cwd*). The Z schema for *FileSystem* is as follows:

FileSystem

$$files : \mathbb{P}\ FullName$$
$$dirs : \mathbb{P}\ FullName$$
$$cwd : FullName$$

$$\forall f : files \cup dirs \bullet\ f \neq \langle\rangle \Rightarrow front\ f \in dirs$$
$$cwd \in dirs$$

FileSystem has three components in its signature, *files*, *dirs*, and *cwd*, and two invariants in the predicate. The first component, *files*, is the set of files that currently exist in the system. (The $\mathbb{P}$ in $\mathbb{P}\ FullName$ is the powerset constructor and is read "set of".) The second component, *dirs*, is the set of directories that currently exist in the system. The last component, *cwd*, does not record any permanent feature of the file system, but is instead used to mark a user's current working directory. The first invariant states that, with the exception of the root directory $\langle\rangle$, for a file or directory to exist, it must be in a valid directory. (As a note on Z notation, the $\bullet$ in the first invariant may be read, "it is the case that".) The second invariant states that *cwd* must be an existing directory. Note that there is no constraint that prohibits files and directories from sharing the same name, although such a constraint might be desirable and could be easily added, by including a third predicate $files \cap dirs = \emptyset$.

5.2 Example MiStix Operation

By way of example, we give the specifications for one representative operation, *CreateDir*. The English specification for the operation *CreateDir* is as follows:

- CreateDir *n*?
  If the name *n*? is not already in the current directory, create a new directory called *n*? as a subdirectory of the current directory, else print an appropriate error message.

The Z formal specification for *CreateDir* is as follows (the specification for the error message has been omitted here):

CreateDir

$\Delta FileSystem$
$n? : Name$

$cwd \frown \langle n? \rangle \notin dirs$

$dirs' = dirs \cup \{cwd \frown \langle n? \rangle\}$

CreateDir modifies the state of the file system (hence the $\Delta FileSystem$), and takes the new directory name, $n?$, as an input. The first predicate, the precondition for the operation, is that the directory to be created, the concatenation of the current working directory with the new name ($cwd \frown \langle n? \rangle$), does not already exist. The second predicate, the postcondition, adds the new directory to the set $dirs'$. Remember that $dirs'$ denotes the value of the *dirs* environment variable after execution of the operation. Note that we do not need to specify that *cwd* is valid, since the *FileSystem* predicates ensure that.

5.3 Tests For *CreateDir*

In this section we apply the method of Amla and Ammann [AA92] to part of `MiStix`. The remainder of the test specification for `MiStix` is similar, although lengthy, and can be found in the technical report [AO93].

The first step in category-partition testing (as given in section 2) is to identify the testable units, which in this case include *CreateDir*.

The second step is identification of the inputs and environment (state) variables for *CreateDir*. From the syntax of the operation, it is clear that $n?$ of type *Name* is the explicit input and that *dirs* and *cwd* are the state variables of interest. Formally, we can describe the input domain for *CreateDir* with the schema:

CreateDir$_{Input\ Domain}$

FileSystem
$n? : Name$

Note that the schema *FileSystem* includes both the declarations for *dirs* and *cwd* and also constraints on the values *dirs* and *cwd* can take. [2]

The third step is the identification of the categories, or important characteristics of the inputs. One source of categories is preconditions on operations. Preconditions are good sources for categories because they are precisely the predicates on the domain of a testable unit that distinguish normal operation from undefined or erroneous operation. For *CreateDir*, the precondition is that the directory to be created does not already exist. Two choices for a category based on the precondition are that the directory to be created does not yet exist and that it already exists.

Another source of categories is revealed by examining other parts of the `MiStix` specification and noting that variables of type *Name* can assume two special values. One special value, which we denote *PARENT*, is the value of $n?$ used when referring to the parent directory. The special value PARENT corresponds to the ".." in `Unix` filename specifications. The behavior of *CreateDir* with respect to a request to create a file named PARENT is technically allowed by the formal specification, but clearly represents a case that should be disallowed. This is an advantageous side effect of deriving test frames in this manner; deriving test data based on the functional specifications can lead the test engineer to identifying anomalies in the specifications themselves.

Another special value of type *Name*, which we denote *ROOT*, is the value of $n?$ used when referring to the root directory. The special value ROOT corresponds to the empty string.

The schema for *CreateDir* suggests two more categories. One is based on whether the current working directory (*cwd*) is the empty sequence (*i.e.* root). The empty sequence is a typical special case for sequences. The last category we employ is the state of the directories set (*dirs*). The motivation for this category is that if it matters if *cwd* is the empty sequence (root), *cwd* would always be root if the only existing directory is the root directory.

The fourth step is partitioning the categories, some aspects of which have already been discussed. The test specifications for *CreateDir* after the first four steps of the category-partition method are:

Functional Unit:
CreateDir

Inputs:
$n? : Name$

Environment Variables:
$dirs : \mathbb{P}\ FullName$
$cwd : FullName$

[2] Since *files* is neither examined nor changed in *CreateDir*, *files* is not a relevant state variable to the operation. As a technical point, we could capture this fact by using the Z schema hiding operator to hide the variable *files* in the schema *CreateDir*$_{Input\ Domain}$, but we elect not to do so for the remainder of the example.

Categories:

Category – Precondition
- Choice 1 (Base): $cwd \frown \langle n? \rangle \notin dirs$
- Choice 2: $cwd \frown \langle n? \rangle \in dirs$

Category – Type of $n?$
- Choice 1 (Base): $n? \neq ROOT \land n? \neq PARENT$
- Choice 2: $n? = ROOT$
- Choice 3: $n? = PARENT$

Category – Type of cwd
- Choice 1 (Base): $cwd \neq \langle\rangle$
- Choice 2: $cwd = \langle\rangle$

Category – Type of $dirs$
- Choice 1 (Base): $dirs \neq \{\langle\rangle\}$
- Choice 2: $dirs = \{\langle\rangle\}$

5.3.1 Creating The Combination Matrix

The combination matrix is a conceptual tool; only those combinations of choices that are selected need be explicitly enumerated. The combination matrix for *CreateDir* has four dimensions, one for each category. There are $2 * 3 * 2 * 2 = 24$ entries in the combination matrix for *CreateDir*.

5.3.2 Identifying The Base Test Frame

To identify a base test frame, a base choice is selected for each category. The normal case for *CreateDir* is the case where the directory to be created does not yet exist and is neither ROOT nor PARENT, where the current working directory is not the ROOT directory, and where the directory set contains more than just the ROOT directory. The indications of base choices are indicated by the word "Base" in the test specification for *CreateDir*.

For brevity, the choices in the test specification are listed as predicates only, although a more complete description is with a set of variable declarations and predicates on those variables, *i.e.*, with a schema, as done by Stocks and Carrington [SC93a, SC93b]. Note that the invariant from *FileSystem*, which describes the set of valid states for the file system, must hold for every choice. The base test frame is the intersection of the base choice for each category, and this is succinctly expressed with the schema conjunction of the base schema for each category.

For *CreateDir* the base test frame is the schema:[3]

$CreateDir_{Base\ Test\ Frame}$

$FileSystem$

$n? : Name$

$cwd \frown \langle n? \rangle \notin dirs$

$n? \neq ROOT \land n? \neq PARENT$

$cwd \neq \langle\rangle$

$dirs \neq \{\langle\rangle\}$

The source of each of the explicitly listed predicates in $CreateDir_{Base\ Test\ Frame}$ is as follows. The predicate $cwd \frown \langle n? \rangle \notin dirs$ comes from the schema that is Choice 1 (Base) for Category – Precondition. Similarly, the predicate $n? \neq ROOT \land n? \neq PARENT$ comes from the schema that is Choice 1 (Base) for Category – Type of n?, and so on.

5.3.3 Choosing Other Combinations

Applying the heuristic from section 4 to the *CreateDir* operation of `MiStix` gives a total of six test frame schemas, the base test frame schema (shown above in $CreateDir_{Base\ Test\ Frame}$) and five additional variations, one for each non-base choice. In the interest of compactness we omit the declaration part of the test frame schemas and only list the predicate parts from the choices. Note that since each test frame schema includes *FileSystem*, each predicate part of a test frame schema listed below also includes the state invariant from *FileSystem*, even though that predicate is not explicitly listed.

Base Test Frame

Test Frame 1: The Base Test Frame schema, $CreateDir_{Base\ Test\ Frame}$ is shown above

Test Frame From Category – Precondition

Test Frame 2: $cwd \frown \langle n? \rangle \in dirs \land$ $n? \neq ROOT \land n? \neq PARENT \land$ $cwd \neq \langle\rangle \land$ $dirs \neq \{\langle\rangle\}$

Test Frames From Category – Type of $n?$

Test Frame 3: $cwd \frown \langle n? \rangle \notin dirs \land$ $n? = ROOT \land$ $cwd \neq \langle\rangle \land$ $dirs \neq \{\langle\rangle\}$

Test Frame 4: $cwd \frown \langle n? \rangle \notin dirs \land$

[3] Note that the schema inclusion of *FileSystem* in $CreateDir_{Base\ Test\ Frame}$ declares the variables *dirs* and *cwd* and supplies the state invariants on these variables.

$n? = PARENT \wedge$
$cwd \neq \langle\rangle \wedge$
$dirs \neq \{\langle\rangle\}$

Test Frame From Category – Type of *dirs*

Test Frame 5: $cwd \frown \langle n? \rangle \notin dirs \wedge$
$n? \neq ROOT \wedge n? \neq PARENT \wedge$
$cwd \neq \langle\rangle \wedge$
$dirs = \{\langle\rangle\}$

Test Frame From Category – Type of *cwd*

Test Frame 6: $cwd \frown \langle n? \rangle \notin dirs \wedge$
$n? \neq ROOT \wedge n? \neq PARENT \wedge$
$cwd = \langle\rangle \wedge$
$dirs \neq \{\langle\rangle\}$

Note the source of each of the four explicitly listed predicates in a given test frame schema listed above. Predicates are mechanically derived as for those in $CreateDir_{Base\ Test\ Frame}$. Specifically, each predicate is a choice from one of the four categories for *CreateDir*. For any given test frame schema, one predicate corresponds to a non-base choice; remaining predicates correspond to base choices.

5.3.4 Identifying Infeasible Combinations

Test Frame 5, developed above for *CreateDir*, is, in fact, infeasible. The conjuncts

$dirs = \{\langle\rangle\} \wedge$
$cwd \neq \langle\rangle$

along with the invariant relation from *FileSystem*,

$cwd \in dirs$

simplify to **false**. Informally, if the root directory is the only directory, then *cwd* must be set to the root directory.

We demonstrate the utility of the combination matrix by showing the entries for the *Type of dirs* × *Type of cwd* part of the combination matrix:

		Type of dirs	
		Base	Root Only
Type of cwd	Base	1	
	Root	2	3

The empty cell represents the combination of choices for Test Frame 5 where there is only one directory, the root directory, and it contains no subdirectories ($dirs = \{<>\}$, and the current working directory (cwd) is some non-root, *e.g.* `/a`.

We revise Test Frame 5 by shifting to a different cell in the combination matrix, namely the cell labeled 3, where *cwd* is ROOT. As a result, we get a revised Test Frame 5 (listed in full schema form):

$CreateDir_{Revised\ Test\ Frame\ 5}$

FileSystem
$n? : Name$

$cwd \frown \langle n? \rangle \notin dirs$
$n? \neq ROOT \wedge n? \neq PARENT$
$cwd = \langle\rangle$
$dirs = \{\langle\rangle\}$

5.3.5 Refining Test Frames Into Test Cases

Refining the test frames into test cases is the process of selecting a representative input from the set of inputs that satisfy the selected choices. In previous work such as DeMillo and Offutt's [DO91], the refinement may be based purely on the syntax of the constraints and the types of the variables, whereas in a sophisticated system such as the one proposed by Wild *et al.* [WZFC92], such a refinement might be based on a knowledge base incorporating other project specific data. Since refinement is not the focus of our present work, we simply present sample test inputs that satisfy the necessary constraints. Each test input is a triple of ($n?$, $dirs$, cwd).

```
All Base Choice
   Test Case 1:  (b, {<>,<a>}, <a>)
Non-Base Precondition Choice
   Test Case 2:  (b, {<>,<a>,<a,b>}, <a>)
Non-Base Type of n? Choice
   Test Case 3:  (PARENT, {<>,<a>}, <a>)
   Test Case 4:  (ROOT, {<>,<a>}, <a>)
Non-Base Type of dirs Choice
   Test Case 5:  (b, {<>}, <>)
Non-Base Type of cwd Choice
   Test Case 6:  (b, {<>,<a>}, <>)
```

5.3.6 Writing Operation Commands

The actual test case commands must use the parameters specified by the values of the test cases in a syntactically correct command to the system being tested. This is a straightforward process that could be automated by using the formal specification of the command. For cell 1 in the above example, the operation is:

Operation: `CreateDir b`

5.3.7 Creating Test Scripts

A test script is derived from a matrix entry by taking the corresponding setup script, creating the syntax for the test operation, and appending the verify and cleanup scripts. A test script for cell 1 from the above matrix is:

Setup:	`CreateDir a`
	`ChangeDir a`
Operation:	`...`
Verify:	`List`
Cleanup:	`Logoff`

5.4 Results of Testing `MiStix`

To demonstrate the feasibility of our technique, we generated and executed a complete set of test data for the `MiStix` system. The implementation is about 900 lines of C source in three separate modules.

We derived 72 test cases for the ten operations, of which 5 were duplicates. Some of the test cases were also superseded by others in the sense that they were prefixes of the other test cases. We did not eliminate these tests (although an automated tool to support this process could easily do so). The `MiStix` system contained 10 known faults, of which 7 were detected.

6 Conclusions

Test specifications are an important intermediate representation between functional specifications and functional tests. In general, it is desirable to know the extent to which a test specification can be derived from the functional specification and the extent to which the tester must rely on information external to the specification. In this paper we have helped to answer this question by presenting a procedure that will guide the tester when creating complete functional tests from functional specifications.

It is unreasonable to expect to derive a test specification completely from the functional specifications. For example, the test engineer might know that programmers often make the mistake of using fixed-sized structures when dynamically-sized structures are required. For example, in the specification for `MiStix`, the depth of the directory tree is not constrained by the Z specifications. The lack of an explicit statement makes it difficult to derive an appropriate category (e.g., directory tree depth). However, to a typical human test engineer, limits on directory depth is a relatively obvious property to check (and indeed such a check can lead to tests that discover faults not found by the generated test specification). Thus the role of mechanically generated test specifications as identified here is to relieve the burden of routine tasks and free the test engineer to concentrate on more creative tasks.

Previous category-partition papers have offered relatively little guidance as to which combinations of choices should be tested. In this paper we have defined the base-choice coverage criterion, and demonstrated the feasibility of applying the criterion to an example specification. Some choices are designated as "base" choices. A base test frame uses all the base choices. Other test frames are generated by systematic enumeration over all non-base choices. This technique is relatively inexpensive (linear in the number of choices) and ensures that each choice will be used in at least one test case (if feasible). In future work we intend to investigate the feasibility of the base-choice coverage criterion for categories derived from other sources than those investigated here.

Acknowledgements

It is a pleasure to acknowledge Tom Ostrand for his helpful comments and encouragement on the application of formal methods to category-partition testing. We are also grateful to Steve Zeil for explaining the utility of Z schemas as test frame descriptors.

References

[AA92] N. Amla and P. Ammann. Using Z Specifications in Category Partition Testing. In *Proceedings of the Seventh Annual Conference on Computer Assurance (COMPASS 92)*, Gaithersburg MD, June 1992. IEEE Computer Society Press.

[AO93] P. Ammann and A. J. Offutt. Functional and test specifications for the MiStix file system. Technical report ISSE-TR-93-100, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1993.

[BHO89] M. Balcer, W. Hasling, and T. Ostrand. Automatic Generation of Test Scripts from Formal Test Specifications. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.

[Dil90] A. Diller. *An Introduction to Formal Methods*. Wiley Publishing Company Inc., 1990.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[GG93] M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. *Journal of Software Testing, Verification, and Reliability*, 3(1), 1993. To appear.

[GMH81] J. Gannon, P. McMullin, and R. Hamlet. Data-Abstraction Implementation, Specification, and Testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[Hay86] I. J. Hayes. Specification Directed Module Testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.

[Hay93] I. Hayes. *Specification Case Studies*. Prentice Hall Publishing Company Inc., 1993.

[Kem85] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[Lay92] G. Laycock. Formal Specification and Testing: a Case Study. *Journal of Software Testing, Verification, and Reliability*, 2:7–23, 1992.

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

[PST91] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall Publishing Company Inc., 1991.

[SC93a] P. Stocks and D. Carrington. Test Template Framework: A Specification-Based Testing Case Study. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis*, pages 11–18, Cambridge, MA, June 1993.

[SC93b] P. Stocks and D. Carrington. Test Templates: A Specification-Based Testing Framework. In *Proceedings of the 15th International Conference on Software Engineering*, pages 405–414, Baltimore, MD, May 1993.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall Publishing Company Inc., 1989.

[Wor92] J.B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

[WZFC92] C. Wild, S. Zeil, G. Feng, and J. Chen. Employing Accumulated Knowledge to Refine Test Descriptions. *Journal of Software Testing, Verification, and Reliability*, 2(2):53–68, August 1992.