Using Formal Methods To Mechanize 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

Draft – Submitted
September, 1993

Abstract

We extend the category-partition method, a specification-based method for testing software. Previous work in category-partition has focused on developing structured test specifications that describe software tests. We offer guidance in making the important decisions involved in transforming test specifications to actual test cases. We present a structured approach to making those decisions, including a mechanical procedure for test derivation. With this procedure, we suggest a heuristic for choosing a minimal coverage of the categories identified in the test specifications, suggest parts of the process that can be automated, and offer a solution to the problem of identifying infeasible combinations of test case values. Our method uses formal schema-based functional specifications and is illustrated with an example of a simple file system. We conclude that our approach eases test case creation and leads to effective tests of software. We also note that by basing this procedure on formal specifications, we can identify anomalies in the functional specifications.

*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 Introduction

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 [3, 12] is a specification-based test method that has received considerable attention. An important aspect 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 in a mechanical way from a functional specification. Other parts require the test engineer to make decisions or rely on experience. The work reported on in this paper aims to isolate those tasks of producing a test specification that are mechanical, thereby freeing the test engineer to focus on the remaining, more intellectually demanding tasks.

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

- refine the latter steps of the category-partition method by supplying a mechanical procedure for deriving test cases from test scripts
- argue that every category-partition test set should include certain tests and supply a general procedure for enumerating these tests,
- supply a method of resolving infeasible tests caused by conflicting choices.

An important 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 [10]) 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 for producing a test specification is already present in a formal functional specification, and hence less effort is required to produce a test specification from a formal functional specification. Second, the use of 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 manipulating artifacts of the testing process itself. Such artifacts include parts of the test specification and actual test cases.

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 [10]. In the DAISTS system of Gannon, McMullin, and Hamlet [6], axioms from an algebraic specification, in conjunction with test points specified by a test engineer, are used to specify test sets for abstract data types. Hayes [9] exploits the refinement of an abstract Z specification to a (more) concrete specification to specify tests at the design level. Amla and Ammann [1] described a technique in which category-partition tests are partially specified by extracting information captured in Z

specifications of abstract data types. Laycock [11] independently derived similar results. More recently, Stocks and Carrington [15, 16] have used formal methods for describing test artifacts, specifically test frames, or sets of test inputs that satisfy a common property, and test heuristics, or specific methods for testing software.

1.2 Outline of paper

The paper begins by reviewing the steps in the category-partition method in section 2. As an in depth discussion of the Z notation is beyond the scope of this paper, the Z constructs that we require are described in section 3; further information may be found in the Z reference manual [14] or one of the many Z textbooks [5, 13, 18]. Our mechanical procedure for deriving test scripts is given in section 4, and 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 [3, 12] 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 may be summarized 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 and the corresponding results and the changes to the environment.

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 reward 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 [7] have developed *classification trees*, a hierarchical arrangement of categories that avoids the introduction of infeasible combinations of choices.

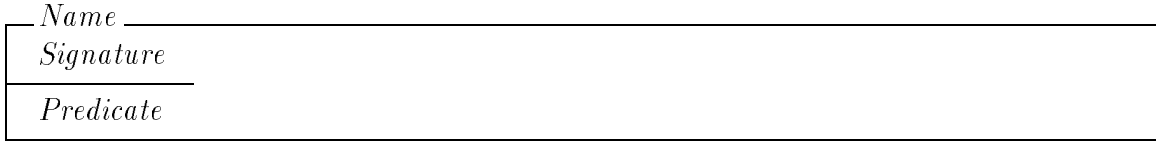
The developers of the category-partition method have defined a test specification language called TSL [3]. A test case in TSL is an operation and values for its parameters and relevant environment variables. A *test script* in TSL consists of the operations necessary to create the environmental conditions (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 [1, 11, 15, 16], 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 [15, 16], we employ the formal specification notation to describe aspects of the tests themselves as well as to describe functional behavior. Third, we identify categories with the function under test instead of a parameter or environment variable of the function. Amla and Ammann present a more elaborate discussion of this point [1].

3 Z Specifications

The formal specification language Z is a mathematical notation based on typed set theory and predicate calculus. Here we restrict our overview to the specific aspects that are relevant to this discussion. In particular, we focus our attention on Z specifications for abstract data types (ADTs). An ADT is characterized by specified states on a data structure and operations that observe and/or change the state.

A Z specification is organized around named objects, called *schemas*, which can be combined to form larger specifications. A schema has three parts; the object *name*, the *signature*, which defines the components of the object, and the *predicate*, which constrains the components. For an ADT operation, the signature typically defines the inputs, outputs, and state variables. The predicate typically defines preconditions and postconditions of the schema. A schema representation is shown below.



For the ADT-based systems we consider here, a Z schema defines either static or dynamic aspects of the behavior of the system.[1] A *static* schema describes the states that the system can have, and it introduces the components (data items) of the state, and invariant relations on the state. A *dynamic* schema describes an operation, or relation between states, that can occur in the system, and gives the preconditions and postconditions for the operation. The preconditions must be satisfied before the operation can be applied, and the postconditions define the relationship between the inputs, the outputs, the prior state, and the resulting state.

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.

A partial mapping of Z constructs to category-partition test specifications is given by Amla and Ammann [1]. We briefly recap major points below:

1. Testable units correspond to "dynamic" schemas, *i.e.*, operations on the ADT.

2. Parameters (inputs) are explicitly identified with a trailing "?". Environment variables (ADT state components) are the components of the "static" 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 [15, 16] 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 [3, 12]. 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 [16, 15]. In this case, the signature part lists the variables that make up possible test inputs

[1] Z schemas are often interpreted in a more general way, as described in the various Z references [5, 13, 14].

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.

4 Mechanical Derivation of Test Scripts

The category-partition work has left a considerable amount of detail in the last two steps to the discretion of the tester. In particular, which combinations of categories to use is an important problem whose solution affects the strength and efficiency of testing. One problem that remains is that some of the combinations of categories are impossible, because they have conflicting requirements. Thus these combinations must be recognized and avoided. Another problem is that the number of combinations of categories can be quite large and repetitious. 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
- P_1
- P_2
- P_3

Y
- Q_1
- Q_2

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 (classes of) test cases, $[P_1;\ Q_1]$, $[P_1;\ Q_2]$, $[P_2;\ Q_1]$, $[P_2;\ Q_2]$, $[P_3;\ Q_1]$, 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). On the surface, only six decisions need to be made (whether to include each test frame), but since the test frames are interrelated, the six decisions cannot be made in isolation. Thus we are left with the question; which of these test sets should be specified?

In TSL specifications [3], 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.

4.1 The Derivation Procedure

As a significant refinement of the category-partition method, we present the following mechanical procedure to create test scripts. This process addresses open issues in the last two steps 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. 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.
   Without some knowledge of the application, one cannot say, in general, which of the many possible test sets are preferable. Some test sets, such as the empty set, are clearly undesirable. The test set with all combinations of choices is the most comprehensive of the possibilities, but is usually repetitious, and can result in an unmanageable number of test frames when there are a large number of choices.

   In our work, we wish to mechanically derive the parts of the test specification that the test engineer must always specify anyway. Therefore we adopt the following minimal goal: For each choice in a category, we combine that choice with the base choice for all other relevant categories. This causes each non-base choice to be used at least once, and the base choices to be used several times. We leave more extensive specification of combinations of choices to the test engineer to choose as the application warrants.

   Once the test engineer chooses a base test frame, 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 $[P_3;\ Q_1]$.

An advantage of this approach is that the number of test frames generated is linear in the number of choices, rather than the combinatorial number from formula 1. The exact number of test frames is:

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

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 assistants 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 the subject of serious inquiry. For example, DeMillo and Offutt [4] 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 [17].

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.

7. Create test scripts.
   Given this information, creating the actual test scripts is a straightforward process. For each chosen cell, the corresponding setup script is chosen, the command is appended, then the verify and cleanup commands are attached. The commands can be given to an automated tool, which can 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 procedure, it is also easy to modify the test scripts once they are created.

5 The `MiStix` File System Example

We demonstrate our mechanized procedure on an example system. `MiStix` is based on the `Unix` file system, and is 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 [8]. 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*. 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 the predicate $files \cap dirs = \emptyset$.

5.2 Example `MiStix` Operation

By way of example, we give the specifications for one representative operation, *CreateDir*. The full specifications for `MiStix` can be found in the technical report [2]. 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 Category-Partition Tests For *CreateDir*

In this section we apply the method of Amla and Ammann [1] to part of `MiStix`. The remainder of the test specification for `MiStix` is similar and can be found in the technical report [2].

The first step in category-partition testing (as given in section 2) is to identify the testable units. Since *FileSystem* is a static schema giving a state description, it is not a testable unit. Since *CreateDir* is a dynamic schema that describes an operation, it is a testable unit.

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. 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.

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 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 an unusual case. 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: $dirs : \mathbb{P}\, FullName$
$cwd : FullName$

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): $n? \neq ROOT \wedge n? \neq PARENT$
- 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 selection of a base choice is somewhat arbitrary, but a good selection is the choice that corresponds to normal or typical system activity. 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 [15, 16]. 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:[2]

$$
\begin{array}{|l}
\hline
\textit{CreateDir}_{\textit{Base Test Frame}} \\
\textit{FileSystem} \\
n? : \textit{Name} \\
\hline
\textit{cwd} \frown \langle n? \rangle \notin \textit{dirs} \\
n? \neq \textit{ROOT} \land n? \neq \textit{PARENT} \\
\textit{cwd} \neq \langle\rangle \\
\textit{dirs} \neq \{\langle\rangle\} \\
\hline
\end{array}
$$

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

5.3.3 Choosing Other Combinations as Test Frames

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 $\textit{CreateDir}_{\textit{Base Test Frame}}$) 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.

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

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:
$$cwd \frown \langle n? \rangle \notin dirs \land$$
$$n? = PARENT \land$$
$$cwd \neq \langle\rangle \land$$
$$dirs \neq \{\langle\rangle\}$$

Test Frame From Category – Type of $dirs$

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

Test Frame From Category – Type of cwd

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

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

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

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 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 one where $cwd = \langle\rangle$. As a result, we get a revised Test Frame 5 (listed in full schema form):

CreateDir$_{Revised\ Test\ Frame\ 5}$

$FileSystem$
$n? : Name$

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

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

$cwd = \langle\rangle$

$dirs = \{\langle\rangle\}$

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 [4], 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.* [17], 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 Choices
   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 example above, 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 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 mechanical 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 be aware that typical (incorrect) implementations for a given problem employ fixed sized structures in cases where dynamically sized structures are required. In the specification for `MiStix`, the depth of the directory tree is not constrained in any way by any explicit statement in the Z specifications. The lack of an explicit statement makes it difficult to mechanically 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 mechanically 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 other areas.

A problem with category-partition testing that has not received much attention is guidance for specifying which combinations of choices should be tested. In this paper we have outlined a technique that designates some choices as "base" choices. A base test frame uses all the base choices, and only the base choices. Other test frames are generated by systematic enumeration over all non-base choices. This technique has the desirable properties that it 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). Another advantage of the technique is that the test engineer's determination of expected results for a test case can be aided by examining the category over which choices are being enumerated. Such an approach reveals inconsistencies in the functional specification if a given test input is interpreted as simultaneously leading to two inconsistent outputs.

Thus, our results support the thesis that test specifications can be useful early in the development process to identify ambiguous and inconsistent parts of the functional specification. An important, but often overlooked point, is that the mere use of formal methods, such as Z, does not eliminate problems with specifications. Testing the specification itself is as valid a role for test specifications as is testing the implementation against the specification.

Our eventual goal with this research is twofold. First we want to find new ways to integrate testing activities into early phases of the lifecycle. Creating test specifications and test scripts immediately after functional specifications is one such activity. Second, we hope to provide test engineers with as much automated support as possible to derive consistently effective test specifications and test cases for system level testing. The mechanical procedure reported here is a step towards this goal; in future work we hope to augment this procedure with automation whenever possible.

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

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

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

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

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

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

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

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

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