# ROLE OF TEST TEMPLATE IN COLLECTION OF DATA: A CASE STUDY

**Amit Singh[1] and Dr. Pardeep Goel[2]**

1Research Scholar, CMJ University, Shillong, Meghalaya

2Associate Professor, Fatehabad, Haryana

---

**Abstract:** It offers little in the way of defining classes of input which we believe to have similar error-detecting ability. In fact, the valid input space on its own is suitable only for deriving a suite of random tests, each a member of the valid input space. Nevertheless, the valid input space is a useful template to define and has an important role to play in the framework. As mentioned at the end of chapter 3, the valid input space of an operation must be the source of all specification-based tests for the operation. This means that any test is an element of the valid input space. It also means that any test template must be a subset of the valid input space. So, we can define a Z type for test templates for a certain operation, Op:
TT Op == p VIS Op
Note the subscripted use of the operation name. This is a practice we will adopt for the remainder of the thesis. This definition defines TT Op to be the type of all test templates for Op.
Schemas vs sets: The significance of bindings It has already been noted that templates describe sets of test data and that Z schemas are used to define templates. It may seem strange not to use sets to define templates. As mentioned in section 4.1, defining test data for an operation involves assigning values to the input components (both state and parameter) of the operation, that is, defining a binding between input component identifiers and values. Thus, a template intuitively defines a set of bindings, which is exactly what a Z schema defines.
Key word: errordetecting ability, operation, assigning.

## INTRODUCTION

Despite the major limitation of testing that it can only show the presence of errors and never their absence1, it will always be a necessary verification technique. Lucid arguments to this eject can be found in [Tan76]. The community is also aware of the usefulness of formal methods for specifying and designing software. The accepted role of formal specifications in program verifications as the basis for proofs of correctness and rigorous transformation methodologies.

The central concept of the framework is the Test Template (TT), which is the basic unit for defining data. The art of designing test data is determining the particular aspects of the implementation that are to be tested, and determining the distinguishing characteristics of input data that test these aspects. Once these classes of requirements are defined, any actual input satisfying them is appropriate test data. Most important is defining the classes of requirements that test data must satisfy. A test template is a formal statement of a constrained data space, and thus can be a description of test data as input meeting certain

requirements. The key features of a test template are that it is

- generic, i.e., it represents a class of input,
- abstract, i.e., it has the same level of implementation detail as the specification,
- insatiable, i.e., there is some representation of a single element of the defined class of input,
- derivable from a formal specification.

Test templates constrain important features of data without placing unnecessary restrictions. That is, test templates can be expressed by constraints over the input variables defined in the specification. In this sense, test templates define sets of bindings of input variables to acceptable values. As with the various data spaces discussed in chapter 3, we use Z schemas to model test templates. For example,

A template b x ;y : N | x < y]

Defines a set of tests having two values, x and y, such that x is less than y. This template can represent the input for a single test case, though it defines an infinite set of possible bindings. The point is that each binding satisfying the template is an acceptable test input exercising the requirements of the single test case. We stress that a test template only defines sets of data. We use templates to represent test data, but there is nothing intrinsic in their definition that indicates they are defining test data for an operation using some criteria. This is done to preserve flexibility and structure in our framework. Later, we define a hierarchy of test templates, and this is where the connection between templates and test cases is made.

**Review of literature**

The most important element of testing is the actual tests themselves, if for no other reason than they are the basis of almost every other testing concern. Accordingly, by far the most prominent use of formal methods in testing is for the derivation and generation of comprehensive black box test sets. The specification is an authoritative description of functionality and is an obvious source for black-box tests. The three different styles of formal specification support different test derivation methods. The key concepts of each approach are somewhat complementary. However, we are considering testing using informal specifications. Informal specifications Any specification is useful in software testing, and most specifications are informal, presented using natural language and sometimes augmented with diagrams and structure charts. Some work has been done on directly using such specifications in testing.

These methods focus on identifying key elements in the specification. The realization when considering deriving tests from informal specifications is the size and impreciseness of such specifications, along with (usually) poor ability tolerate components of the specification. Clearly, tool support is a major consideration. Ostrand et al. [OSW86] describe a tool for managing specification-based testing from

informal specifications. The major functions of the tool for to annotate parts of the specification for record keeping purposes (for example, highlighting functional units), and maintain relationships between parts of the specification and any test information derived from them. Category partitioning [OB88, BHO89] is a more advanced method for natural language specification-based testing. Specifications are analyses to determine the various functional units. For each functional unit, the relevant characteristics of the parameters and environment objects are classed in categories. Then, using experience, the tester decides significant choices of input for the categories. This information is the basis of the test suites. The strength of the method is the definition of a test specification language, TSL, used in automatic construction of test suites and test execution. This is described in more detail in section 2.3.2. It is clear that most of the effort in these approaches is extracting information from informal specifications that is trivial to extract from formal specifications. An example is the work on category partitioning using Z specifications discussed in the section on test derivation from model-based specifications below.

**Material and Method**

We see that test templates and valid input spaces have similar definitions as bindings of input variables to appropriate data values. Our definition of a test template is deliberately flexible, and clearly the valid input space of an operation is a test template for that operation. As a test template, the valid input space of an operation is very coarse. It offers little in the way of defining classes of input which we believe to have similar error- detecting ability. In fact, the valid input space on its own is suitable only for deriving a suite of random tests, each a member of the valid input space. Nevertheless, the valid input space is a useful template to define and has an important role to play in the framework. As mentioned at the end of chapter 3, the valid input space of an operation must be the source of all specification-based tests for the operation. This means that any test is an element of the valid input space. It also means that any test template must be a subset of the valid input space. So, we can define a Z type for test templates for a certain operation, Op:

TT Op == p VIS Op

Note the subscripted use of the operation name. This is a practice we will adopt for the remainder of the thesis. This definition defines TT Op to be the type of all test templates for Op.

Schemas vs sets: The significance of bindings It has already been noted that templates describe sets of test data and that Z schemas are used to define templates. It may seem strange not to use sets to define templates. As mentioned in section 4.1, defining test data for an operation involves assigning values to the input components (both state and parameter) of the operation, that is, defining a binding between input component identifiers and values. Thus, a template intuitively defines a set of bindings, which is exactly what a Z schema defines. The set of bindings can be constrained by predicates in the same way as

sets are defined using set comprehension. Schema types define generalized tuples, where ordering of components is not significant, and individual components can be referenced. Consider these alternatives

Schema T [x ; y : N | x < y]

Set T = {x ;y : N | x < y}

Used as a template, Set T defines a set of ordered pairs, where individual components cannot be referenced. Schema T defines a set of bindings of values to the identifiers x and y. If B were such a binding (B : Schema T ), then B :x and B :y reference the components of the binding. The descriptive power of schemas fits the idea of describing test data. What does using a schema to represent a template mean? As a test template, Schema T describes the set of test data consisting of two components, x and y, both natural numbers, satisfying the condition that x is less than y. Templates are, of course, types in the Z notation. An instance of a template is a particular binding of values to components, and represents an actual test.

The particulars of the Z syntax and semantics raise two points in the usage of schemas as test templates. These do not restrict the use of templates, but must be made clear. A useful concept in the framework is reasoning with sets of templates, that is, sets of Z schemas. Both points relate to this usage.

Bindings are described using the notation from [Spi92]

Schema [x; y :] type : p | x : N ; y : N|

Schema Set ==Schema type : p(p | x :N; y :N)

S : Schema type : x :N; y :N

S Set: Schema Set type : p | x :N; y :N

SS : S Set type : x : N; y :N

Both Schema and S Set define sets of bindings, and instances of each are as expected.

However, they are not exactly the same. Despite the similarity, Schema is a Z schema, and S Set is only a set of bindings. This means that operations of the schema calculus cannot be applied to S Set : it is a set, not a schema. In every other regard Schema and S Set are identical. Instances of both are bindings (with no ordering of elements and component reference). Because the types of schemas and sets of bindings are so similar, schemas can be used in set expressions. Set operations require all sets in the expression to have the same signature. The resulting type of a set expression involving schemas is a set of bindings.

We use a structured approach to build a hierarchy of test templates. Coarser templates are iteratively divided into smaller templates using testing strategies. Test data derivation is simplified by this structured approach involving the systematic application of various testing strategies.

Since all tests for an operation must be derived from the operation's valid input space, the valid input space is the starting point of a hierarchy. Once the valid input space of the functional unit is determined,

the next step is to subdivide the valid input space into the desired subsets, or partitions, called domains. Choice of domains is not determined by the test template framework. Rather, testing strategies and heuristics are used to subdivide the valid input space. The goal is to derive domains which are equivalence classes of error-detecting ability for the function under test, and which cover the valid input space. That is, the goal is to choose domains so that each element of a domain has the same error-detecting ability. Some, but not all, strategies assume every element of a domain is equivalent to all the others for this purpose and so only one need be chosen. However, this assumption is often invalid. To preserve the flexibility to choose tests for domains selectively, the domain derivation step is used repeatedly, dividing domains into further sub-domains, until the tester is satisfied that the domains represent desired equivalence classes.

This derivation results in a collection of test templates, related to each other by their derivation and the strategies used in their derivation. We construct a graph where nodes are templates and edges represent application of testing strategies. The edges are directed from parent templates to child templates. Typically, a template hierarchy looks something like Figure 4.1. A hierarchy can be considered as a tree of tests, with the valid input space at the root. In fact, in the general case, a hierarchy is a directed graph, because it is possible to derive the same template using different strategies (and hence different links in the graph). The significance of a template in the hierarchy is that it can be used as the source of test data. If it is too coarse for this, there should be sub-templates derived representing finer divisions of the parent template. The terminal nodes in a hierarchy represent the final input classes. Some strategies do not advocate domain partitioning (e.g., random testing), in which final tests are derived directly from the valid input space. Some partitioning strategies assume each member of a domain is equivalent to all others, in which case only one level of derivation is required. Some strategies may advocate further subdividing of already derived templates. The framework is merely a defining structure, and doesn't enforce particular derivation approaches on the tester. Figure 4.1 shows a common hierarchy structure.

The hierarchy of templates for each operation is a directed graph. Notational, all elements of the hierarchy relating directly to the particular operation or functional unit under test are subscripted with the operation's name. All templates in the hierarchy are sub-schemas of the valid input space. The hierarchy shows the derivation structure of the templates as a relationship between sets of templates derived from some other template using some testing strategy. The generic set of strategies is introduced and deliberately left abstract:

[STRATEGY ]

The Test Template Hierarchy (TTH) graph for an operation is a set of mappings from parent template/strategy tuples to the set of child templates derived from the parent using the strategy:

$TTH_{Op} : TT_{Op} \times STRATEGY \rightarrow TT_{Op}$

Templates are defined in terms of their parents and additional constraints. For example, a template, T 1, derived from $VIS_{Op}$ with the additional constraint is defined. If the strategy used in this derivation was strat, then its position in the hierarchy can be described by

strat : STRATEGY T1 € $TTH_{OP}$ ($VIS_{OP}$, strat)

If T 1 is the only template derived from the valid input space using strat, then this section of the hierarchy can be completely defined by

{(T 1)} = $TTH_{OP}$ ($VIS_{OP}$,strat )

Useful relationships among templates, based on the structure of the hierarchy, can be defined. We define two standard functions over templates in a hierarchy: children $_{op}$ and descendants op .

Children $_{Op}$ : $TT_{op} \rightarrow P (TT_{op})$

Children $_{op}$ = ($\lambda$ T : $TT_{op}$ ♦$\cup${s: STRATEGY ♦$TTH_{op}$ (T ; s )})

Descendants $_{op}$ : $TT_{op} \rightarrow (TT_{op})$

Descendants $_{op}$ = ($\lambda$T : $TT_{op}$) The function children $_{op}$ determines the set of templates directly derived from some template using any strategy. For example, given the hierarchy in figure 4.1

childrenFig1 (VIS) = {$Ta_1$,..,T am ,..,Tb1 ,..,T bn}

The function descendant $_{op}$ determines the set of templates directly or indirectly derived from some template using any strategy. That is, the descendant templates from some template are all the templates in the sub-graph extending from that template. For example, given the hierarchy in figure 4.1

descendantsFig1 (VIS) ={$Ta_1$,.., T am ,..,T b1 ,..,T bn,

Tc1,..,Tco,.., Td1 ,..,T dp ,..,Te1T,..,eq ,..,T f1 ,..,T fr}

After applying all the desired strategies to derive test templates, the template hierarchy is considered complete. Instances of the templates in the hierarchy represent test data. If no further subdivision of templates is to be undertaken, each instance of a terminal template in the hierarchy graph is considered equivalent to all other instances of this template for testing purposes. For a complete description of the test data, the only remaining task is to instantiate the terminal templates in the hierarchy. There are two ways to view the instantiation of templates. Before discussing these, however, it must be noted that an instance of a template is a precisely defined object, but it is still abstract. That is, it exists at the same level of abstraction as the templates. An instance of a template will most likely not serve as final test data because it probably has some data reification to undergo. For example, suppose one input class identified by a test template for queue operations involves a two element queue (of natural numbers, say) with duplicate elements. In Z, the queue would be represented by a sequence, so this template would be

QT1=[q : seq N | # q = 2 ^ #(ran q) = 1]

Any instances of this template expressed in Z describe specific Z sequences (e.g., h1; 1 ), but if the final implementation refined the sequence representation of the queue to a linked list, the instances of templates would also have to be refined to suitable linked list equivalents. The most straightforward way to describe instances of templates is to use schema instantiation. If QT 1 is a template, then

Q : QT 1

is an instance of the template it is (abstract) test data. This form of instantiation is no more useful than the original template because no new information is presented.

Constraints can be defined on instances, so this approach could be used to describe the test datum mentioned above:

Q : QT 1

Q : q = (1, 1)

**Conclusion**

The first point is one of Z syntax. Schemas are Z types. Defining objects with schema types (bindings) has the syntax inst: Schema. However, this is a short hand for the syntax inst: {Schema}, which states that inst: is a member of the set of bindings defined by Schema. Because of this shorthand, a singleton set of schemas, containing only schema S, can not be declared $\{_S\}$, since this is merely the schema type set of all bindings defined by S ." Rather the singleton set is placed in parenthesis to unambiguously describe the correct set: {(S)}. Non-singleton sets of templates can be defined normally, since there is no ambiguity.

There is a subtle difference between schemas and schema types in Z, best illustrated with an example. Consider the following definitions with the type of the defined entity shown at its side.

The preferred approach to describing instances is to define instance templates. These are merely templates (schemas) with only one possible instantiation. This approach is more attractive in three ways. Firstly, it presents more information, as in the second example of using schema instantiation above. Secondly, uniform use of schemas and templates is made in the hierarchy, which is important when we consider making general expressions about all templates in a hierarchy. Thirdly, some templates derived using strategies may have only one instantiation so, again, the uniformity of the model is preserved. The instance template corresponding to the instance of QT 1 described above is simply defined as

Q =[QT 1 | q = (1,1) ]

Again, the final translation of instance templates to concrete test data is implementation dependent. Instance templates are incorporated into the hierarchy. The strategy to derive instance templates is assumed in the framework: instantiation : STRATEGY

References

1. Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, Inc., Reading, MA, 1999.
2. M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218. ACM Press, 1989.
3. T. Ball. The limit of control flow analysis for regression test selection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 134–142. ACM Press, March 1998.
4. Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.
5. Boris Beizer. *Software Testing Techniques*. Van Nostrong Reinhold, New York, NY, 1990.
6. Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, Boston, MA, 1999.
7. R. Biyani and P. Santhanam. TOFU: Test optimizer for functional usage. *Software Engineering Technical Brief*, 2(1),1997.
8. Jonathan P. Bowen and Michael G. Hinchley. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995.
9. Bill Brykczynski. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes*, 24(1):82 1999.
10. T.A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1980.