A Comparison of the Model-Based & Algebraic Styles of Specification as a Basis for Test Specification

Richard Denney

Landmark Graphics Corporation
220 Foremost Blvd
Austin TX 78745
RDenney@zycor.lgc.com

Abstract

The use of formal specifications as a basis for specifying functional tests has been discussed by a numbers of researchers with most work focusing on one style of specification or another separately. But is any single style an adequate basis for writing functional tests? The strengths, weaknesses and complementary nature of two popular styles of software specification, model-based and algebraic, are examined as a basis for functional test specification.

Keywords: functional testing, formal methods, model-based specifications, algebraic specifications

1 Introduction

The use of formal specifications as a basis for specifying functional (black-box) tests has been discussed by a number of researchers. [Amla & Ammann], [Hall], [Hayes], [Laycock] and [Zweben & Heyn] have looked at the use of model-based specifications while others have advocated algebraic specifications [Doong & Frankl], [Gannon, McMullin & Hamlet] [McMullin & Gannon]. Most work to date has focused on the use of one or the other separately, but to do a thorough job of specifying tests, one may really need aspects of both [Yip & Robson]. In this paper we will look at the strengths, weaknesses and complementary nature of the these two popular styles of specification as a basis for functional test specification.

2 Model-based Specifications as a Basis for Test Specification

The model-based approach to software specification is arguably the most popular style of formal software specification going today, both VDM and Z being based on this style [Hayes, Jones & Nicholis]. A model-based specification provides a model of a system's state in terms of a collection of state variables. Each state variable models some aspect of the system's retained data as a mathematical object; Z for example uses sets, relations, functions and sequences. System operations are then described in terms of how they modify this state model: pre-conditions define valid inputs and start states for an operation; post-conditions define outputs and the state of the system after operation invocation. State invariants define properties, or constraints, of the system state which must always hold and which operations are obliged to preserve.

Let's illustrate these concepts with an example of modeling the operation of allocating devices to users of an operating system. We begin with the state variables which describe the retained data of our system:

Devices is the set of all devices in the operating system, e.g. disks and printers.

Users is the set of users currently logged into the operating system.

FreeDevices is the set of devices which have not yet been allocated to a user.

DeviceTable is a partial (not all devices need be allocated) many/one mapping from **Devices** to **Users**, i.e. many devices can be allocated to a single user, but at most one user can have a given device allocated at a time.

State invariants are described in terms of properties, or constraints, over the state variables:
FreeDevices $\subseteq$ **Devices**
FreeDevices $\cup$ domain (**DeviceTable**) = **Devices**
FreeDevices $\cap$ domain (**DeviceTable**) = $\emptyset$

The first invariant states that **FreeDevices** is a subset (none, some or all) of **Devices**. The second invariant says all devices must be accounted for: the union of free devices and allocated devices must equal all the devices on the system. The third invariant says that a device can either be free, or allocated, but not both at the same time: the intersection of free devices and allocated devices is empty.

The initial state of the system - all devices are free and the device table is empty - is given by describing the initial values of state variables:
FreeDevices = **Devices**
DeviceTable = $\emptyset$

Given the state model we can now define the operation of allocating a device to a user[6]:

Operation: allocate device to user

inputs:

D? is device being requested
U? is user requesting device

pre-conditions:

D? $\in$ **FreeDevices** ! Must be valid, free device
U? $\in$ **Users** ! Must be valid user

post-conditions:
DeviceTable' = **DeviceTable** $\cup$ { (D?,U?) }
FreeDevices' = FreeDevices \ { D? }

The model-based approach has gained much attention as a formal basis for functional test specification. Let's briefly look

[6] Following the style of Z, inputs to the operation (inputs are transient data; not part of the retained data of the system) are decorated with the suffix "?". The suffix prime (') indicates the modified (post-condition) form of a state variable. The operator "$\cup$" denotes set union. The operator "\" denotes set substraction: S1 \ S2 is equal to the set S1 with the elements of S2 removed.

at what makes it attractive from a testing standpoint.

2.1 Testing Inputs

First, as it is generally impossible to test an operation across all possible inputs, techniques such as equivalence partitioning and boundary-value analysis [Myers] are typically used to divide up the input space into test domains from which test cases are drawn. The model- based style of specification is a rich source of deriving test domains for inputs. An input space can be divided into test domains based on:

- properties of the mathematical type an input is modeled as, e.g. a property of sets is their cardinality, or the fact that elements of a set are not ordered
- an input's relation to state variables as documented in the pre-conditions
- constraints placed on the input in the pre-condition

As example, given the pre-condition U? $\in$ **Users** and the fact that **DeviceTable** is a partial mapping from **Devices** to **Users**, one partitioning of the input space for U? is:

1. values of U? where U? $\in$ range(**DeviceTable**), which are users which do not have a device already allocated
2. values of U? where U? $\in$ range(**DeviceTable**), which are users which already have a device allocated

2.2 Error Handling

Pre-conditions also provide a basis for identifying error handling tests for an operation. The pre-conditions of an operation are a *conjunction* of terms describing *valid* inputs and start states; negating the pre-conditions yields a *disjunction* of error cases that an operation should be tested against; a simple application of DeMorgan's Law. As example, given the pre-condition:
D? $\in$ **FreeDevices** error cases for D? are described by the negation of the pre-condition:
D? $\notin$ **FreeDevices**

Given invariant **FreeDevices** $\subseteq$ **Devices** one partitioning of these error values of D? is:

1. values of ?D where D?? $\notin$ **FreeDevices** but where D? $\in$ **Devices**, which is devices which are already allocated
2. values of ?D where D? $\notin$ bf FreeDevices and D? $\notin$ **Devices**, which is to say completely bogus device names

2.3 State Testing

Finally, just as it is typically not possible to test an operation's inputs across all possible values, neither is it usually possible to test an operation in all possible states of the system. There are two aspects of the model-based approach to specification which lend themselves to identifying states that an operation should be tested in.

2.3.1 State Defined in Terms of Independent State Variables.

First, because the system's state is modeled as a set of independent (though related) state variables, an operation's definition need refer to *only* those state variables which affect it, or which it affects. From a testing standpoint this means we have already greatly reduced the number of system states which in theory we need to consider testing the operation in. So, given an entire operating system one is able to describe the operation of allocating devices to users by referring to just those parts of the state model – **FreeDevices, Users** and **DeviceTable** – which affect, or are affected by, the operation.

To appreciate the importance of this feature to test specification, let's contrast it with an algebraic specification. An algebraic specification consists of the **signatures** of operations being defined - the syntax - plus a set of **axioms** which define the semantics of the operations. A signature consists of an operation name, the number and type of its arguments and the type of the operation's result:

Signature of Operations: initial: $\rightarrow$ State
allocate: User x Device x State $\rightarrow$ State
deallocate: User x Device x State $\rightarrow$ State

Axioms:
1. deallocate(**U, D**, initial()) = initial()
2. deallocate(**U, D**, allocate(**U, D, S**)) = **S**
3. deallocate(**U1, D**, allocate(**U2, D, S**)) =
allocate(**U2, D, S**) where **U1** $\neq$ **U2**
4. allocate(**U, D2**, allocate(**U, D1, S**)) =
allocate(**U, D1**, allocate(**U, D2, S**)) where **D1** $\neq$ **D2**.
5. allocate(**U, D**, allocate(**U, D, S**)) = allocate(**U, D, S**).

The first axiom says no device can be deallocated from the initial state; trying to do so affects no change on the system state. In these axioms variables are all *universally* quantified, so in the second axiom we have that for all users **U**, devices **D** and states **S**, operation *deallocate* undoes the state change of a previous operation *allocate*. The third axiom says that a user **U1** cannot deallocate a device **D** which has been previously allocated to another user **U2**. The fourth axiom states that given two devices, **D1** and **D2**, a user **U** can allocate them in any order. The fifth axiom states that a user **U** who attempts to allocate a device **D** which is already allocated to him or her will cause no change of state.

The point to be made is that in these axioms the *entire* state of the operating system is modeled as a *single* universally quantified variable, **S**. As opposed to the model-based approach, in the algebraic approach it is not possible to tell which aspects of the system state are really pertinent to the axiom at hand. The beauty of the algebraic approach — that it abstracts away detail about the state [Gannon et.al.] — presents a problem with respect to its use in specifying functional tests: how do we know what states (values of **S**) are pertinent to test this axiom in? We'll return to this problem later.

2.3.2 Ability to Divide State Space into Test Domains.

So a model-based definition of an operation allows one to

identify *just* those aspects of the system state (state variables) pertinent to testing the operation. But the possible values of a state variable still represent more system states than we are typically able to test an operation in. What we would like to do is divide the state space described by a state variable (or variables) into test domains from which sample states could be selected, just as we did with inputs. This leads us to the second aspect of model-based specifications which lends itself to identifying states that an operation should be tested in: a concrete definition of the state space. Because each state variable is a mathematical description of some retained data, we are able to use the same techniques used with inputs to build test domains for state variables. The state space can be divided into test domains based on:

- properties of the mathematical type a state variable is modeled as
- the relationship between multiple state variables as documented in a pre-condition
- the relationship between multiple state variables as described in the retained data model:
  a) a state variable defined in terms of other state variable(s)
  b) the initial state of a state variable defined in terms of other state variable(s)
  c) state invariants
- test domains based on comparison of a state variable's pre and post-condition form

As example, given the invariant a partition of this relationship between **FreeDevices** and **Devices** is:
1. **FreeDevices** $= \emptyset \wedge$ **Devices** $= \emptyset$
2. **FreeDevices** $= \emptyset \wedge$ **Devices** $\neq \emptyset$
3. **FreeDevices** $\subset$ **Devices** $\wedge$ **Devices** $\neq \emptyset$
4. **FreeDevices** $=$ **Devices** $\wedge$ **Devices** $\neq \emptyset$

And given these invariants:
1. **FreeDevices** $\cup$ domain(**DeviceTable**) $=$ **Devices**
2. **FreeDevices** $\cap$ domain(**DeviceTable**) $= \emptyset$

a partition of this relationship between **FreeDevices** and **DeviceTable** is:
1. **FreeDevices** $= \emptyset \wedge$ domain(**DeviceTable**) $=$ **Devices**
2. **FreeDevices** $\subset$ **Devices** $\wedge$ domain(**DeviceTable**) $\subset$ **Devices**
3. **FreeDevices** $=$ **Devices** $\wedge$ **DeviceTable** $= \emptyset$

3 The Algebraic Approach and Test Specification

The popularity of the model-based approach as a basis for test specification is clear from the work that has been done in the area:[Amla & Ammann], [Hall], [Hayes], [Laycock] [Yip & Robson] and [Zweben & Heyn]. In this section, however, we'll look at aspects of the algebraic approach which can be useful for test specification, but which the model-based approach does not address.

Let's begin with a quick comparison to the model-based approach. The model-based approach advances a model of a system's state in the form of state variables which represent the retained data of the system. All the possible values of these variables, constrained by the invariants over them, define the system's state space. By contrast the algebraic approach offers no model of the retained data of the system; rather state is modeled in terms of combinations of operations required to achieve that state. In the model-based approach *individual* operations are defined in terms of their effects on the state model. In the algebraic approach one cannot define operations individually; rather *combinations* of operations are defined in terms of their equivalence (via equations) to other operations or combinations of operations. As we'll see, these fundamental differences are precisely the source of their complementary nature for use in test specification.

3.1 Specifying How Operations Work in Concert

In terms of state transition testing, [Chow] characterizes 3 types of state transition errors:

- **operation error** — the state transition goes to the next state correctly, but produces the wrong output.
- **transfer error** — the state transition goes to some existing, valid system state, but its just not the right one for that state transition.
- **missing/extra state error** — the system has extra or missing states

[Chow] states that testing each state transition individually — what he calls "0 switch testing" — can catch operation errors, but that for catching transfer errors and missing/extra state errors one must use tests involving longer sequences of transitions. Chow's paper offers theoretical justification for what one might argue is good testing common sense: you can test all the operations of a system individually, but at some point you need to test them in combination. In the model-based approach each operation represents a single, individual state transition from one state (described by the pre-conditions) to another (described by the post-conditions). The model-based approach to specification provides a good mechanism for specifying tests for individual operations. *What it lacks, however, is a mechanism for specifying how operations should be tested in concert with one another.*

Specifying how the operations of a system work in concert is precisely what the algebraic approach to specification is all about. Algebraic specifications are popular as a formalism for defining Abstract Data Types (ADT). The definition of an operation on an ADT is given solely in terms of how it interacts with other operations to manipulate the ADT. One simply cannot define an operation in isolation of other operations on the ADT. If one views the state of a system as an ADT and defines that ADT via an algebraic specification, the resulting axioms amount to properties about the state transitions of the system; properties which we can specify need to be tested.[7] There is, however, a critical piece that the algebraic approach does not supply but which we will need to

[7] This view of ADTs as a model of state and algebraic axioms as a model of state transitions is discussed in [Roberts].

adequately specify state transition tests. Recall this axiom from our earlier operating system example:

deallocate(U, D, allocate(U, D, S) = S

The system's state is modeled as a single universally quantified variable, S. The problem is, we cannot test the system in every possible state, and because the algebraic approach abstracts away detail about the system state it offers nothing to help us in deciding just what states (values of S) are pertinent to test the axiom in. But as discussed previously the model-based approach is quite good at identifying states that a function should be tested in. Combined the algebraic and model-based methods give us the two pieces we need to specify state transition tests; respectively:

- properties (axioms) about state transitions universally quantified over the system state, S
- and test domains for S based on the model-based specification of, in this case, operation *allocate*()

This is a good illustration of how the two styles of specification can be used to complement one another in test specification.

3.2 Constructors and State Transition Testing

Fundamental to state transition testing is knowing what operations are needed to reach all possible states of a system [Beizer]. The algebraic approach makes a useful contribution in this regard. The algebraic approach categorizes operations into those that affect state, and those that do not, the latter being operations which simply **observe** or **query** the state. The model-based method also allows this distinction to be made, e.g. in Z special notation is used to mark operations (*schemas*) as to whether they do, or do not, modify state. The algebraic approach, however, further categorizes state modifying operations using an important concept not found in the model-based approach. From the set of state modifying operations one identifies the set of **constructors**: the set of operations which are required to generate *all* possible states of the system. All other state modifying operations are simply **modifiers**: they modify state, but there is no system state which cannot be reached without them.[8] Notice that this concept of a set of constructors arises from thinking about collections of operations, as the algebraic approach does, rather than thinking about operations individually in isolation from others, as in the model-based method. The significance of the set of constructors to testing is that one has explicitly identified (and reduced the number and combination of) the operations that need be considered to reach all possible states of the system.

Of course the utility of knowing the set of constructors for a system is only as good as one's confidence that the set is valid: that every operation in the set is needed; that all operations that are needed are included. The algebraic approach gives us a method for testing this. Recall that in the algebraic approach a state is defined in terms of the combinations of operations which were used to reach it. By definition then, we should be able to define all states in terms of combinations of constructors. Let's take an example to illustrate how we can use this fact to test a set of constructors.

We have a vending machine with the following operations:

Signature of Operations:
initial: → State
`deposit_coin: Coin x State` → State
`deposit_bill: Bill x State` → State
`select: Product x State` → State
`coin_return:State` → State

We propose a set of constructors for the machine: `{initial, deposit_coin, deposit_bill, select}`. Operation `coin_return` is used to cancel deposits which may have been made, returning the money to the customer. If a dollar bill is deposited into the machine, return change will be given in coins. We tentatively designate it as a modifier as it seems useful only for returning to a previously exiting state. So first, how can we convince ourselves that each of the operations in the set of constructors is really needed? Let's consider the case of `deposit_coin`; we test it as a constructor by trying to write axioms which compare it against each of the other constructors. We begin by comparing it against the initial operation which creates a newly initialized system. If indeed `deposit_coin` were not a constructor, which is to say if it were simply a modifier, we should be able to specify a right hand side for the axiom expressing a state equivalent to the one described on the left hand side, but using only the other constructors:

```
deposit_coin( C, initial() ) =  ?
```

As we are unable to write the right hand side for this axiom, `deposit_coin` must make a unique contribution to defining state of the system: it is indeed a constructor.

It is one thing to err by mistaking a modifier as a constructor: at least we won't miss any states this way. It is more serious to err by mistaking a constructor as a modifier; this will cause us to miss states. In this case we would like to convince ourselves that `coin_return` is simply a modifier, and does not contribute to any unique states. We do this in a fashion similar to above. If `coin_return` is just a modifier, we should be able to write a set of axioms each of which has a left hand side comparing `coin_return` with a constructor; the right hand side of each axiom should express a state equivalent to the one described on the left hand side, but using only the proposed constructors.

Below we have an attempt at such a set of axioms:

```
coin_return( initial()) =  initial()
coin_return( deposit_coin(C, {\bf S})) =
coin_return({\bf S})
coin_return( select(P, {\bf S})) =  select(P, {\bf
S})
```

[8] Various terms have been applied to these categories of operations. [Mallgren] calls these *inquiry, basic generators* and *(non-basic) generators*, respectively. [Roberts] calls them *observation functions, (non- convertible) constructors* and *convertible constructors*. [Guttag & Horning] call them *observers, constructors,* and *extenders*.

`coin_return( deposit_bill(B, {\bf S})) = ?`

Notice with the last axiom we are unable to express the state `coin_return(deposit_bill(B, {\bf S}))` in terms of our initially proposed set of constructors. This is because our problem statement was that return change is always given via coins: if a `coin_return()` operation is performed after a bill has been deposited in the machine, the state will change such that the number of bills in incremented, and the number of coins is decremented. Our inability to express this new state with the initially proposed set of constructors indicates that `coin_return()` is itself a constructor of system state when used in conjunction with `deposit_bill()`.

3.3 Using the System as Its Own Oracle

We now look at a virtue of the algebraic approach for test specification that was first discussed by [Gannon et.al.]. Traditional oracle-based testing relies on supplying a function with inputs then having an "oracle" test to see that the output is correct. Draw-backs to this form of testing are:

- For each test case, expected outputs must be generated by the oracle, and recorded for each input.
- As the "oracle" is usually a human, generating expected outputs is subject to error.

These drawbacks are magnified when oracle-based testing is used as a basis for regression testing: inputs along with expected outputs must be stored in a test library; changes to, or additions of, an input require corresponding re-calculation of expected output and subsequent update of the test library.

[Gannon et.al.] states that testing with algebraic oriented test cases avoids the need for an oracle, and hence all the problems which stem from an oracle's use. An alternate view of this is that with the algebraic oriented test cases, we are letting the system itself act as its own oracle! Either way you view it, the result is that test cases can now be written in which one worries only about inputs.

Let's take an axiom from our earlier operating system example:

allocate(**U**, **D2**, allocate(**U**, **D1**, **S**)) = allocate(**U**, **D1**, allocate(**U**, **D2**, **S**)) where $\mathbf{D1} \neq \mathbf{D2}$

Recall the axiom states that given two devices, **D1** and **D2**, a user **U** can allocate them in any order. From a testing standpoint, the right and left hand sides of the axiom both represent test cases, replete with inputs, with each side serving as the expected output of the other. By letting the system itself compute both the left and right hand sides dynamically at test time, the system acts as its own oracle, and eliminates the need for pre-test-time calculated, and stored, expected outputs.

4 Conclusion

The utility of complementary styles in software specification is not new [Hoare]. "Structured Analysis" has employed the concept of modeling systems from different perspectives since the mid-70s [DeMarco] and more recently multiple, complementary, perspectives have been advocated for object-oriented modeling, e.g. [Coleman, et.al.] [Rumbaugh, et.al.].

In formal test specification, however, most work has focused on the use of one specification method or another. This paper has examined the strengths and weaknesses of two popular specification styles with respect to their use in test specification to motivate the argument that a single method may not always be adequate.

Finally, other methods of axiomatic specification have been proposed which address problems associated with the algebraic style, notably Bartussek & Parnas **trace specifications** with normal forms [Hoffman & Snodgrass] [Lamb]. In the context of testing discussed in this paper, the strengths, weaknesses and complementary nature with respect to the model-based approach of specification are, I believe, equally valid for trace specifications. I chose algebraic specifications for comparison because of their popularity and history of use in testing by [Gannon, McMullin & Hamlet].

Acknowledgments

I'd like to thank Susan Gerhart for our conversations on this topic.

References

Amla & Ammann, Using Z Specification in Category Partition Testing, Proceedings COMPASS 1992

Boris Beizer, Software Testing Techniques, 2nd ed., Van Nostrand Reinhold, 1990

Tsun S. Chow, Testing Software Design Modeled by Finite-State Machines, IEEE Transactions on Software Engineering, 4(3), May 1978

Coleman, Arnold, Bodoff, Dollin, Gilchrist, Hayes & Jeremaes, Object-Oriented Development: The Fusion Method, Prentice-Hall, 1994

Tom Demarco, Controlling Software Projects, Prentice- Hall, 1982

Roong-Ko Doong & Phyllis G. Frankl, The ASTOOT Approach to Testing Object-Oriented Programs, ACM Transactions on Software Engineering and Methodology, 3(2), April 1994

John Gannon, Paul McMullin & Richard Hamlet, Data- Abstraction Implementation, Specification & Testing, ACM Transactions on Programming Languages & Systems, 3(3), July 1981

J.V. Guttag & J.J. Horning, The Algebraic Specification Of Abstract Data Types, in Programming Methodology, David Gries, editor, Springer-Verlage, 1978

P. Hall, Brunel University UK, Towards Testing with Respect to Formal Specifications, 2nd IEE/BCS Conference, Publication #290, Software Engineering 88, July 1988

Ian Hayes, Specification Directed Module Testing, IEEE Transactions on Software Engineering, 12(1), Jan 1986

I. J. Hayes, C.B. Jones and J.E. Nicholls, Understanding the differnces between VDM and Z, Technical Report UMCS-93-8-1, Department of Computer Science, University of Manchester

C.A.R. Hoare, Oxford University Computing Lab, An Overview of Some Formal Methods for Program Design, IEEE COMPUTER, Sept 1987

Daniel Hoffman & Richard Snodgrass, Trace Specifications: Methodology of Models, IEEE Transactions on SE, 14(9), Sept 1988

David Lamb, Software Engineering: Planning for Change, Prentice Hall, 1988

G. Laycock, Formal Specifications and Testing: A Case Study, Journal of Software Testing, Verification and Reliability, 2(3-6), May 1992

William Mallgren, Formal Specifications of Interactive Graphics Programming Languages, MIT Press, 1983

P. McMullen & J. Gannon, Evaluating a Data Abstraction Testing System Based on Formal Specifications, Journal of Systems and Software, Vol 2, 1981

Glenford Myers, The Art of Software Testing, John Wiley & Sons, 1979

Yip & Robson, Conformance Validation of Graphical User Interfaces, IEEE Transactions on Software Engineering, 1991

W.T. Roberts, A formal specification of the QMC Message System: the underlying abstract model, Computer Journal 31(4) 1988

Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, Object-Oriented Modeling & Design, Prentice-Hall, 1991

Zweben & Heym, Systematic testing of data abstractions based on (formal) software specification, Journal of Software Testing, Verification and Reliability, 1(4), 1992