Appeared in Eivind Coward (ed.): *Norsk Informatikkonferanse 2005*,
Tapir, Trondheim Norway 2005, pp. 43--54.
http://www.nik.no/2005/

Structured testing in Sophus*

Magne Haveraaen Enida Brkic
Institutt for informatikk, Universitetet i Bergen

Abstract

Testing is very important for the validation of software, but tests are all too often developed on an ad hoc basis. Here we show a more systematic approach with a basis in structured specifications. The approach will be demonstrated on Sophus, a medium-sized software library developed using (informal) algebraic specifications.

1 Introduction

Testing is one of the oldest and most used approaches for the validation of software. In spite of its importance, it is often performed as an add-on to the development of the software. In some approaches, e.g., extreme programming [Bec99], testing, has been given a prominent role. Still, even in those approaches, tests are normally created in an ad hoc fashion.

Here we investigate the systematic exploitation of specifications for testing, specifically utilising algebraic specifications of a library as a means of systematically structuring and re-using tests for the library components. We will demonstrate this on selected specifications and implementations from the Sophus library, see figure 1.

Algebraic specifications [BM03] represent a high-level, abstract approach to specifications, completely void of implementation considerations. The idea of testing based on algebraic specifications is not a new one, see e.g., [GMH81, Gau95, Mac00]. We follow the technique of [GMH81], extending it to structured specifications.

Sophus [HFJ99] is a medium-sized C++ software library developed for solving coordinate-free partial differential equations. The aspect of the library that is relevant to us is the fact that it was developed using extensive, but informal, algebraic specifications. The specifications are small and build upon each other, and every implementation is related to several specifications. We want to exploit this structure in the development of tests, so that we may re-use the same tests for many different implementations.

This paper is organised as follows. Next we describe the structure of the Sophus library. Then we look closer at the notion of specification and give some sample Sophus specifications. Section four discusses the notion of implementation and sketches some Sophus implementations. Section five is about testing the Sophus library. And finally we sum up our approach and conclude.

*This presentation builds on Enida Brkic' master's thesis [Brk05].

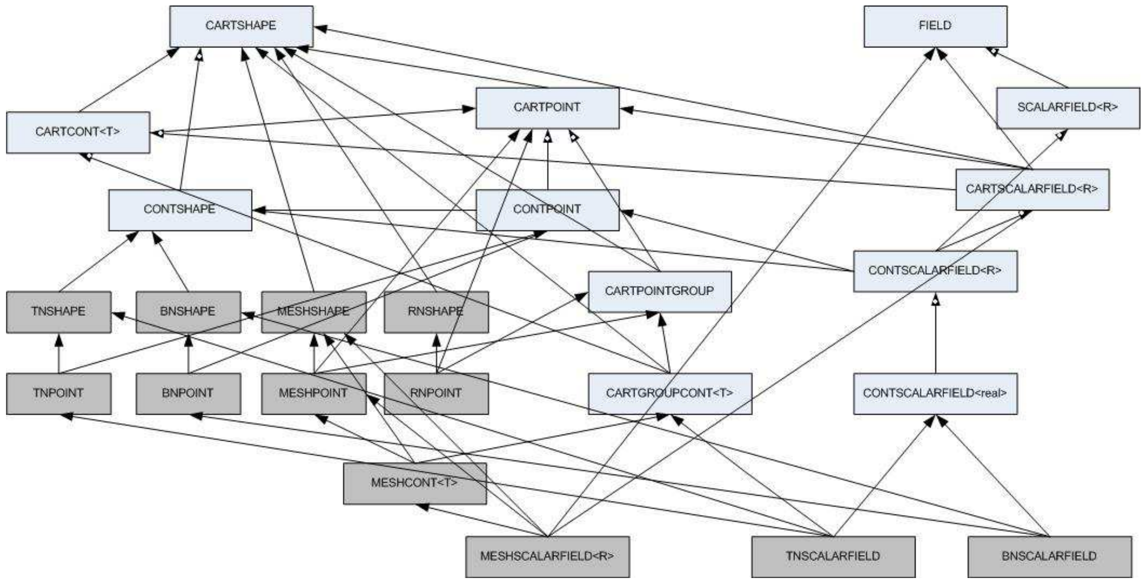

Figure 1: The arrows relate implementations (shaded dark) to specifications (lightly tinted) in the Sophus library. A specification *extends* another specification (punctured arrowhead) or *uses* another specification (normal arrowhead). An implementation *uses* another implementation or *satisfies* a specification (and all its subspecifications).

2 The structure of the Sophus Library

The Sophus library was developed with a focus on making reusable components. In order to do this, algebraic specifications were used as a domain analysis and engineering tool [Hav00], and many small specifications were written to capture the central concepts.

Implementations were also targeted to be as general as possible, trying to establish a high level of reuse even within the library itself.

One effect of this is that each specification may relate to several implementations, and that each implementation relates to several specifications. Figure 1 shows a small part of this structure. The specification `CartPoint` uses the specification `CartShape`, while `ContShape` extends the `CartShape` specification. The implementation `BnShape` satisfies the `ContShape` specification, and hence its subspecification `CartShape`. The implementation `BnPoint` uses the `BnShape` implementation, while it satisfies the `ContPoint` specification, and hence the subspecification `CartPoint`.

The fragment of Sophus shown defines three main kinds of concepts:

- *shapes*: descriptions of index sets,
- *points*: elements of an index set of a given shape,
- *containers* and *scalar-fields*: indexed structures with given shapes.

This setup is to ensure the correct typing when a scalar-field is indexed by a point: only indices (points) that are relevant for the container will be applied. So the indexing operator has declaration

`_ [ _ ] :  C⟨E⟩, Point → E, shape(c)==shape(p)` **guards** `c[p]`

where `C⟨E⟩` is a (template) container/scalar-field type with elements of type `E`, and `Point` is a point type. The guard requires the condition that the shape (allowed indices) for the scalar-field and the shape of the point (the actual index) must match for the indexing to be valid.

The shapes are Cartesian, as in being Cartesian products of sets. So each point will have several, independent values, i.e., the points themselves are small arrays indexed by *directions*. The dimension of a shape tells how many directions the corresponding points have. The shape encodes the legal index bounds for each direction of a point. The index sets given by a shape may be the often used intervals of integers, but they may also define continuous index sets, as needed for the application domain of partial differential equations.

Each of the abstractions have additional properties beyond that of just being indices and arrays. We will illustrate this by a simplified (and slightly adapted) presentation of a small part of Sophus. For this reason, not all specifications we present are part of figure 1. We will sketch specifications `CartShape`, `BoundedShape`, `CartPoint`, `UnboundedPoint` and `BoundedPoint`, and implementations `RnShape`, `RnPoint`, `MeshShape` and `MeshPoint`. This will be sufficient to show the reuse of the same (sub)specification in several contexts, and that library implementations relate to more than one specification unit. The goal is to exploit this structure when building tests.

3 Specification

Here we present some standard theory about algebraic specifications, see for example [LEW96]. Then we show selected parts of Sophus specifications.

Specification theory

Given a declaration of *sorts* (also known as types or classes) and *operations* (also known as functions or methods). Then we may declare variables of the sorts, and build terms (expressions) from the variables and the declared operations.

An equational axiom is the statement that two such expressions should be equal. They are typically written as an equation, with an expression on each side of an equality sign. Here we use the C/C++/Java == symbol as our equality sign. We assume that the only declared variables are the variables visible in the axiom, i.e., the *free variables* of the axiom.

To check that an equation *holds*, we must provide a *model* for the sorts and operations. In computer science speak, this means we need to provide an implementation. The implementation will for each sort define a data-structure, a place-holder for the values, and for each operation an algorithm. We also need to give the variables values. This is called an assignment, and can be thought of as putting data in the data structure for each variable. It is then possible to evaluate the expressions in an axiom, and compare them to check that they are equal. Denote the model by M, the variables by V, the axiom by `ax` and the assignment by $a : V \to M$, seeing assignments as a function that gives each variable a value from the model. We can then write that axiom `ax` *holds* in the model M for the assignment a by

$$M \models_a \mathtt{ax}. \tag{1}$$

If any of the operations are guarded, only assignments which are accepted by the guard can be used.

A model M *satisfies* an axiom ax, if ax holds in M for all possible assignments a, or, in a mathematical notation,

$$M \models \mathtt{ax} \quad \Leftrightarrow \quad \forall (a : V \to M) : M \models_a \mathtt{ax}. \tag{2}$$

A specification is a collection of axioms, and a model M satisfies a specification if all of the axioms are satisfied.

Some shape and point specifications from Sophus

Shape specifications

We will focus on two shape specifications, CartShape which is general and applies to any finite-dimensional shape, and BoundedShape which is for bounded, finite-dimensional shapes.

The CartShape specification establishes the basic notion of a Cartesian shape. A Cartesian shape has a finite set of *directions*, the number of directions is the *dimension* of the shape. Dimensions and directions are given as integers (natural numbers).

```
specification CartShape⟨sorts Shape⟩
// Cartesian index set, each index has n directions
operations
  setDimensions :  int → Shape, n≥0 guards setDimensions(n)
  // set the dimension, i.e., the number of directions
  getDimensions :  Shape → int
  // get the dimension, i.e., the number of directions
  legalDirection :  Shape, int → bool
  // check that a direction index is within bounds
axioms
  getDimensions(setDimensions(n)) == n
  legalDirection(s,d) == (0≤d and d<getDimensions(s))
```

This is a loose specification with respect to Shape. We have not given enough axioms to pin down the meaning of the sort, in fact we have not provided enough operations to make it possible to pin this down. This looseness (and incompleteness) is deliberate. It gives the specification a higher degree of reusability.

The BoundedShape specification is more specific than CartShape, as it in addition to the finite Cartesian assumption defines that the extent of each dimension is bounded.

```
specification BoundedShape⟨sorts Shape, R⟩
// Bounded Cartesian index set
includes CartShape⟨Shape⟩
operations
  setBounds :  Shape, int, R, R → Shape,
     legalDirection(s,d) and rl≤ru guards setBounds(s,d,rl,ru)
  getLower :  Shape, int → R, // lower bound for a direction
     legalDirection(s,d) guards getLower(s,d)
  getUpper :  Shape, int → R, // upper bound for a direction
     legalDirection(s,d) guards getUpper(s,d)
axioms
  getLower(setBounds(s,d,rl,ru),d) == rl
  k≠d ⇒ getLower(setBounds(s,k,rl,ru),d) == getLower(s,d)
  getUpper(setBounds(s,d,rl,ru),d) == ru
  k≠d ⇒ getUpper(setBounds(s,k,rl,ru),d) == getUpper(s,d)
```

The get-functions are supposed to return the corresponding values that have been set, and the set operation is supposed to only change the value of one direction. We make the obvious restriction that the upper bound should be larger than the lower bound in the guard to the `setBounds` operation.

Point specifications

The members of the index set defined by a given shape are called points. Cartesian points have an index value (of sort `R`) for every direction defined by its shape. The operation `_[_]` returns this index value. These are used to index the actual scalar field data, and to move around the data, movements given by the + operation.

```
specification CartPoint⟨sorts Shape, Point, R⟩
// Cartesian point set, each point is a Cartesian shape
includes CartShape⟨Shape⟩
operations
  setShape :  Shape → Point
  getShape :  Point → Shape
  getDimensions :  Point → int
  legalDirection :  Point, int → bool
  setDirection :  Point, int, real → point,
     legalDirection(p,d) guards setDirection(p,d,r)
  _ [ _ ] :  Point, int → R, legalDirection(p,d) guards p[d]
  _ + _ :  Point, Point → Point
axioms
  getShape(setShape(s)) == s
  getDimensions(p) == getDimensions(getShape(p))
  legalDirection(p,d) == legalDirection(getShape(p),d)
  setDirection(p,d,r)[d] == r
  k≠d ⇒ setDirection(p,k,r)[d] == p[d]
  p1+(p2+p3) == (p1+p2)+p3 // associativity
implies
  legalDirection(p,d) == (0≤d and d<getDimensions(p))
```

The implies clause is derived from the meaning of `legalDirection` on the shape.

A Cartesian point specialises to an unbounded point type, where we know that addition on the parameter type is to be consistent with addition on the point type.

```
specification UnboundedPoint⟨sorts Shape, Point, R⟩
includes CartShape⟨Shape⟩, CartPoint⟨Shape,Point,R⟩
axioms
  (p1+p2)[d] = p1[d] + p2[d] // lifted _+_:  R, R → R
```

In the bounded version we know more about the point values, but we do not have a general specification of addition, as there is no general way to further restrict point addition for arbitrary bounds.

```
specification BoundedPoint⟨sorts Shape, Point, R⟩
includes CartShape, CartPoint⟨R⟩, BoundedShape⟨R⟩
axioms
  getLower(getShape(p),d) ≤ p[d]
  p[d] ≤ getUpper(getShape(p),d)
```

Strictly speaking these two axioms are not equations, just positive assertions on the data.

4 Implementation

The implementation theory is based on [Mor73, LZ74], which provides the basic insight into why class-based programming is so successful. Important aspects of this insight seems largely ignored in modern computer literature, but is slowly creeping back into consciousness.

Implementation theory

Given a declaration of sorts (also known as types or classes) and operations. An implementation provides a data structure for each of the sorts, and an algorithm for each of the operations.

The sort then derives its meaning from the values we may insert into the data structure. But we may not (want to) be using all possible data values. There may be natural limits within the data, e.g., an integer representing months will only have values between 1 and 12. Or we may have declared some redundancy, e.g., setting aside a variable to track the size of a linked list so we do not have to traverse it whenever we need to know how long it is. Such restrictions are captured by the *data invariant*, a guard which states what data values are accepted into the data structure. Only data values that makes the data invariant hold are accepted in the data structure. We assume that every implementation alongside the data structure for a sort also provides a data invariant checking operation `DI`.

Another question which arises is when we should consider two values for a data structure to be the same. Intuitively we may think that the values must be identical, but this is not always the case. We normally consider the rational numbers $\frac{1}{2} = \frac{3}{6}$, even though the data set $(1,2)$ is different from the data set $(3,6)$. To overcome this, an implementation should define the == operator (or `equals` method in Java) for every sort. This is standard practice in C++. In Java/C++ the compiler in any case provides one for you unless you define it yourself. This equality operation is the one used to check the equations.

For an implementation to be consistent, it needs to satisfy two basic requirements.

- Every algorithm must preserve the data invariants: if the input data satisfies the data invariant, so must the output data.
- Every algorithm must preserve equality: given two sets of arguments for an algorithm, such that every corresponding pair of input data structure values (possibly different data sets) being equal according to the equality operator, then the two (possibly different) resulting data structure values produced by the algorithm must be equal according to the equality operator.

The first requirement makes certain every algorithm takes legal data to legal data. The second requirement maintains the "illusion" of equality according to the user-defined equality operator. The former property may be seen as a guard on the data structure. The latter property may be formalised as a set of conditional equational axioms, one for each operation being declared.

This approach eliminates the *oracle problem* of [Gau95], as we now are treating the equality operator as any other operation of the specification with regards to its correctness and implementability.

Some shape and point implementations from Sophus

We will first sketch the implementation of the unbounded n-dimensional real number shape.

- `RnShape` is a shape type with only the number of dimensions in its data structure. The data invariant asserts that this number is at least 0, and the equality operator on `RnShape` checks that the number of dimensions are the same. `setDimensions` sets this value, `getDimensions` reads it, and `legalDirection(s,d)` checks that 0≤`getDimensions(s)`, exactly as in the `CartShape` axioms given that `RnShape` is the `Shape` parameter.

- The related points, `RnPoint`, for an *n*-dimensional `RnShape`, is a list of *n* real values, spanning the *n*-dimensional Euclidean space.

We then sketch the implementation of the bounded mesh-type. These represent a Cartesian product of integer intervals.

- `MeshShape` is a *n*-dimensional shape for integer intervals. Its data structure contains the number of dimensions `n:int` and two arrays `L,U`, each containing `n` integers. The data invariant asserts that the number `n` is at least 0, and that for all `i:int`, 0≤`i`<`n`, `L[i]`≤`U[i]`. The equality operator on `MeshShape` checks that the number of dimensions are the same and that the corresponding elements of the corresponding arrays of the two shapes contain the same values. `setDimensions(n)` sets the attribute `n` and initialises all array elements to 0. `getDimensions` reads the attribute `n`, and `legalDirection(s,d)` checks that 0≤`getDimensions(s)`. The function `setBounds` sets the corresponding elements of the arrays `L` and `U`, while `getLower` and `getUpper` access the corresponding elements of `L` and `U`, respectively. This should ensure `BoundedShape`⟨`MeshShape,int,`≤`:int,int`→ `int`⟩.

- An *n*-dimensional `MeshPoint` is a list of *n* integers, for each direction bound by the corresponding upper and lower bounds of the associated `MeshShape`.

Note that `RnShape` and `MeshShape` represent different abstractions, Euclidean continuous space versus lists of integer intervals, and that their data structures are significantly dissimilar, a natural number versus a natural number and two lists of integers with some constraints. Yet they are both to satisfy the `CartShape` (sub)specification.

5 Testing

The theory on testing we present is loosely based on [Gau95], but extended in the direction of [GMH81] to also handle other specification styles than conditional equations.

Testing theory

Testing is a form of validation where we run the algorithms on selected data sets in order to increase our belief in their correctness. The decision procedure that decides if a test is successful is called a *test oracle*.

We have three different correctness criteria for our algorithms.

- Preservation of the data invariant: every operation can check the data invariant on the return value.

  This is sufficient to guarantee the preservation of the data invariant, provided we are certain only the algorithms of our model are allowed to modify the data in a data structure. Checking every algorithm's return value will then be the earliest possible point of detecting a breach of the data invariant[1] as pointed out by [Mor73].

- Preservation of equality: whenever we have two alternative sets of argument data to an algorithm, we need to verify that the algorithm returns "equal" data if the data sets are "equal"[2].

- Checking the axioms: whenever we have a set of data values corresponding to the free variables of an axiom, we should check that the axiom holds. Note that even though this is an absolute criterion of correctness, an error may be in any one of the operations in the axiom, even in the equality operator itself.

The first two of these test criteria relate to the consistency of the implementation. Only the last criterion is related to the specification per se. Also note that the first criterion does not need any specific data, the check can be performed whenever an algorithm is executed. The latter two criteria needs us to device data for the checks.

As noted in the section on implementations, the second correctness criterion may be encoded as a collection of conditional, equational axioms. In the following we will treat these "equality preservation axioms" together with the "normal" axioms, avoiding the need to discuss them specifically in any way. We also check that our "equality" is an equivalence by adding these equations to the "normal" axioms.

From specification theory we know that an implementation M is correct with respect to (a combined user-defined and equality preservation) specification if it satisfies all the axioms. The implementation M satisfies an axiom $\mathtt{ax}$ if it holds for all assignments $a : V \to M$ of (data invariant and declared guarded) assignments to the data structures, $M \models_a \mathtt{ax}$. But normally the set $A(\mathtt{ax}, M) = \{a : V \to M \mid V \text{ the free variables in } \mathtt{ax}\}$ of possible assignments is too large. It will not be possible to check that the model holds for every assignment.

In testing we then choose some smaller set $T \subseteq A(\mathtt{ax}, M)$ so that we get a manageable amount of checks to perform.

$$M \models_T \mathtt{ax} \quad \Leftrightarrow \quad \forall a \in T : M \models_a \mathtt{ax}. \tag{3}$$

Such a set T is called a *test set*. A test set is *exhaustive* if it guarantees correctness. Normally we will not have an exhaustive test set. But a *test reduction hypothesis* allows us to reduce the size of the test set, yet guarantee *correctness relative* to the hypothesis. The more powerful the reduction hypothesis, the smaller data set we can use for exhaustive testing.

One test reduction hypothesis is the *random value selection hypothesis* [DN84]. The idea here is to randomly select values from the entire domain $A(\mathtt{ax}, M)$. Every such value has a certain probability of discovering an error. The more values selected the better, up until a certain threshold, where the probability of detecting an hitherto undetected error rapidly decreases. Since the selection is random, there is no bias in the data towards ignoring specific errors in the implementation. The random selection function need not

[1] If it is possible to modify data by other means, we must either do a data invariant check when data has been modified by such means, or check the data invariant for the input arguments as well. An algorithm is not required to satisfy any specific property if it is supplied data not secured by the guards, and a data invariant breach would also invalidate the premise for the algorithm.

[2] Note that this is a relative relation between the equality operator and the algorithm, as there is no formal definition that makes one of them more fundamentally correct than the other.

be a uniform distribution, but could be distributed according to normal usage patterns, towards especially safety critical parts, etc.

Another class of test reduction hypothesis are the *domain partitioning hypothesis*. These try to split the test universe $A(\mathtt{ax}, M)$ into (possibly overlapping) subsets. Then a few representative test values may be chosen within each partition. The idea being that these test values will detect any problem within the partition they belong to.

Partitioning can be combined with random test value selection. Then we choose a random selection of values in each partitioning. The chosen values will then not be distributed evenly across the whole domain, but rather in each subdomain.

A common domain partitioning hypothesis is the *discontinuity hypothesis*. The idea being that in subdomains where an operation behaves "continuously" it will behave in the same manner for any value, but at the borders of discontinuity we risk irregular changes in behaviour. So we need to select "normal" test values and "border" test values to best discover any errors in the implementation. There are two main approaches to generate such partitionings: specification based and implementation based. Thinking of the implementation as a box, these are also referred to as black box and clear box, respectively, referring to whether we can look inside to the implemented code or not.

In the following we will look at specification based (black box) testing. This implies that we look at the specification to find the discontinuities.

If this partitioning gives specific terms which replace some variables in an axiom, then we call the axioms specialised with these terms for *test cases*. If all values for a model are generated, i.e., there exists syntactic expressions representing each value, then we may reduce all testing to test cases [Gau95], avoiding the need to administer external data.

Some tests for the Sophus Library

We now need to derive test oracles and test data partitionings from the loose (Sophus) specifications. Since our starting point is loose specifications, we are in general unable to pin down any concrete test values. But any conditional in the specification (guard or axiom) normally indicates a discontinuity of some sort. We will then (randomly) choose a single value at each such border, and a single value in the continuous parts.

Building reusable test oracles

We take every axiom in a specification and turn it into a reusable test oracle operation by the following steps.

1. the operations have the free variables of the axiom as parameters
2. the template arguments of the specification become template arguments of the operation, if they appear as sorts for the free variables of the axiom
3. the body of the operation is the axiom itself

Following this procedure[3], we turn the two `CartShape` axioms into the following two test oracles.

```
bool CartShapeAxiom1(int n)
{ return getDimensions(setDimensions(n)) == n; }
template⟨sorts Shape⟩
```

[3] At the moment we have no automatic tool for this, but work has started to provide a tool taking formal specifications to test oracle operations.

```
bool CartShapeAxiom2(Shape s, int d)
{ return legalDirection(s,d) == (0≤d and d<getDimensions(s)); }
```

All implementation satisfying `CartShape` will, for any data assignment to the variables, pass the tests above.

For the Cartesian point classes, we generate test oracles similarly, e.g., for the 6th axiom which gives the associativity of +.

```
template⟨sorts Point⟩
bool CartPointAxiom6(Point p1, p2, p3)
{ return p1+(p2+p3) == (p1+p2)+p3; }
```

An interesting observations, which is possible to explore in some cases, is that `CartShapeAxiom1` has the same form as the (only) implication given for `CartPoint`. We may then actually reuse this shape oracle in the point context.

The test oracle operations we have showed so far are valid for all shapes / points. A test oracle created from, e.g., a `BoundedShape` axiom checking for the correctness of bounds, is not relevant for `RnShape` since this sort has no bound concept.

Some test data cases

We will here apply the discontinuity domain splitting hypothesis to a few selected axioms. Our first example is `CartShapeAxiom1`. The `setDimensions` operation requires that the parameter is at least 0. This gives a natural splitting into two test cases: the value 0 and some (arbitrary) value greater than 0. So we reduce the test oracle into two parameterless test-cases.

```
bool CartShapeAxiom1Case0() { return CartShapeAxiom1(0); }
bool CartShapeAxiom1Case1() { return CartShapeAxiom1(4); }
```

The number 4 is a random value greater than 0. These two tests oracles are all we need to validate any shape implementation given the test hypothesis we have chosen.

A similar analysis for `CartShapeAxiom2` gives us the following 6 test cases.

```
bool CartShapeAxiom2Case1()
{ return CartShapeAxiom2(setDimensions(7),0); }
bool CartShapeAxiom2Case2()
{ return CartShapeAxiom2(setDimensions(5),4); }
bool CartShapeAxiom2Case3()
{ return CartShapeAxiom2(setDimensions(9),2); }
template⟨sorts Shape⟩
bool CartShapeAxiom2Case4(Shape s) 0<getDimensions(s) guards
{ return CartShapeAxiom2(s,0); }
template⟨sorts Shape⟩
bool CartShapeAxiom2Case5(Shape s) 0<getDimensions(s) guards
{ return CartShapeAxiom2(s,getShape(s)-1); }
template⟨sorts Shape⟩
bool CartShapeAxiom2Case6(Shape s, int d)
0<d and d<getDimensions(s)-1 guards
{ return CartShapeAxiom2(s,d); }
```

The 0'th test case, with the number of dimensions equal 0, does not exist since there are no direction values `d` such that `0≤d<0`. Note how the latter test cases put restrictions on the test sets based on what already has been tested. Also see that we are unable to provide any further splitting of the test cases, as the information we have about discontinuities from declarations do not allow this. Further, the operations we have in `CartShape` do not allow us to construct any test values for the latter three test cases.

Some test data sets

Constructing the remaining test data sets for `CartShapeAxiom2` requires us to know which implementation to test.

Given `RnShape`, we observe that the only constructor for shape data is `setDimensions`. Then the first three test cases for `CartShapeAxiom2` covers all relevant cases for the discontinuity domain splitting hypothesis.

For `MeshShape` we are able to construct further test data. Using the same hypothesis we first create a test data set with 3 test data values. These can be used for `CartShapeAxiom1Case4` and `CartShapeAxiom1Case5`.

```
setBounds(setBounds(setDimensions(2),0,3,3),1,-5,-5);
setBounds(setBounds(setBounds(setDimensions(3),0,3,7),1,-5,7),2,3,7);
setBounds(setBounds(setDimensions(2),0,3,7),1,-5,7);
```

For `CartShapeAxiom1Case6` we need data sets for dimensions $n \geq 3$, as $n = 3$ dimensions is the smallest size we can use in order to satisfy the guard `0<d<`n`-1`. Note how these sets need to cover the cases where `rl==ru` and `rl<ru`, and we need this when all directions in the shape belong to the first (==) of these two cases, when the directions represent a mix of cases, and where all directions represent the latter (<) case. The hypothesis does not give us any hints about this being better (or worse) than reusing the same data values where applicable.

Even though the values used as inputs to the tests only can be constructed from the implementation, the discontinuities defining the test cases arise from the analysis of the specification, maintaining the black box nature of this testing discipline.

6 Conclusion

We have briefly presented theories for algebraic specification, implementation and testing, and showed how these apply to a fragment of the Sophus library. The approach systematically derived reusable test oracles from axioms, and the process was supplemented with test case and test data generation related to the specifications. This approach is now being used for the systematic validation of Sophus, and tool support for this is under way.

This validation exploits the specifications already written as part of the development of Sophus. It also exploits the structure of these specifications, allowing the reuse of tests for all implementations that satisfy the same (sub)specification. Such reuse of test cases across a wide range of implementations does not seem to have been reported earlier.

The formulation we have chosen for test satisfaction, see equation 3, seems to easily extend to any institution where the formulas have variables, whether free or explicitly quantified. This improves over earlier attempts to integrate testing in the institution framework [LGA96, DW00], although it remains to be seen whether there exists a general characterisation of institutions with variables.

References

[Bec99] Kent Beck. Extreme programming: A discipline of software development. In Oscar Nierstrasz and M. Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, page 1. Springer, 1999.

[BM03] Michel Bidoit and Peter D. Mosses, editors. *CASL Casl User Manual - Introduction to Using the Common Algebraic Specification Language*, volume 2900 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Brk05] Enida Brkic. Systematic specification-based testing of coordinate-free numerics. Master's thesis, Universitetet i Bergen, P.O.Box 7800, N–5020 Bergen, Norway, Spring 2005.

[DN84] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.

[DW00] M. Doche and V. Wiels. Extended institutions for testing. In T. Rus, editor, *Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 514–528. Springer Verlag, 2000.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995.

[GMH81] John D. Gannon, Paul R. McMullin, and Richard G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[Hav00] Magne Haveraaen. Case study on algebraic software methodologies for scientific computing. *Scientific Programming*, 8(4):261–273, 2000.

[HFJ99] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modelling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

[LEW96] J. Loeckx, H.-D. Ehrich, and B. Wolf. *Specification of Abstract Data Types*. Wiley, 1996.

[LGA96] P. Le Gall and A. Arnould. Formal specification and testing: correctness and oracle. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Algebraic Development Techniques*, volume 1130 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.

[Mac00] Patrícia D.L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, University of Edinburgh, 2000.

[Mor73] J.H. Morris. Types are not sets. In *Proceedings of the ACM Symposium on the Principles of Programming Lanugages*, pages 120–124, October 1973.