The Axioms Strike Back: Testing with Concepts and Axioms in C++

Anya Helene Bagge Valentin David Magne Haveraaen
University of Bergen
http://www.ii.uib.no/~{anya,valentin,magne}/

Abstract

Modern development practises encourage extensive testing of code while it is still under development, using unit tests to check individual code units in isolation. Such tests are typically case-based, checking a likely error scenario or an error that has previously been identified and fixed. Coming up with good test cases is challenging, and focusing on individual tests can distract from creating tests that cover the full functionality.

Axioms, known from program specification, allow for an alternative way of generating test cases, where the intended functionality is described as rules or equations that can be checked automatically. Axioms are proposed as part of the *concept* feature of the upcoming C++0x standard.

In this paper, we describe how tests may be generated automatically from axioms in C++ concepts, and supplied with appropriate test data to form effective automated unit tests.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing Tools; D.1.5 [*Programming Techniques*]: Object-oriented Programming

General Terms Reliability, Languages, Design

Keywords Algebraic Specification, Axiom-Based Testing Axioms, C++, C++0x Concepts, Generative Programming, Mouldable Programming, Program Transformation, Test Generation, Unit Testing,

1. Introduction

Modern software engineering practises encourage the use of unit testing to increase software reliability. Test-driven development (TDD) [7] dictates that software should be extended by writing tests for a new feature *first*, before implementing the feature. The tests provide a specification of the behaviour of the new feature, and provide an easy way to check the implementation throughout development and refactoring.

Less extreme methods call for tests for all program units, and for regression tests to be written to ward off the reappearance of known bugs. Such methods may be practised rigorously, or in an ad hoc manner. Common to all is that they rely on the programmer to invent good test cases that cover both likely, unlikely and even 'impossible' errors. The programmer must also be careful that the tests exercise the full expected feature set, or the implementation will seem OK when all it does is implement the bare minimum to pass the tests (as is common in TDD, where the tests are the actual specification of expected behaviour).

1.1 Axiom-Based Testing

We suggest writing tests based on *axioms* that formally specify expected behaviour, rather than relying on ad hoc test cases. Axiom-based testing was introduced in the early eighties in the DAISTS [18] system, which used formal algebraic specifications as a basis for unit testing. In DAISTS, a test consists of axioms in the form of conditional equations, which serve as a test oracle, an implementation with an equality operator; and a set of test data. A simple coverage analysis is done of the test runs to ensure that all the axioms and program code are exercised by the tests.

ASTOOT [17] applied the ideas of axiom-based testing to object-orientation, with automated testing for Eiffel. Axioms were specified in an OO-like style, rather than the functional notation used in DAISTS.

The Daistish system [34] brought these ideas to C++. Unlike ASTOOT, Daistish used a functional notation for axioms, giving a notational gap between the conditional equational form of the axioms and the methods of C++.

Traditional unit testing, as popularised by agile methods in the last decades, is practically oriented, and does not rely on formal methods. Mainstream software engineers have focused on development methods like TDD and extreme programming [6], while much formal methods research has focused on formal specification and verification – which have been difficult to apply to mainstream languages and mainstream development.

Research on axiom-based testing has continued, however, and axioms are have been introduced as part of the new *concept* proposal for the upcoming C++ standard [9, 23] – giving a mainstream language built-in syntactic support for axioms. The CASCAT system [47] provides a tool for testing Java components based on algebraic specification. Axiom-based testing has been employed in the Sophus numerical C++ library [31], and also in the JAX [44] (Java Axioms) testing approach and JAxT [33] tool. Axiom-like features have also been added to recent versions of JUnit [40]. Gaudel and Le Gall [20] provide a survey of the use of algebraic specification in testing.

Axiom-based testing from concepts has two main parts that instrument the implementation being tested:

- axioms, in the form of conditional equations, and
- suitable test data points.

Running an axiom-based test consists of evaluating the condition and (if it succeeds) the two sides of the equation using the test data, and comparing the results, typically with the help of the equality operator. For example, to test the following commutativity axiom $x + y = y + x$, we may substitute 4 for x and 5 for y, evaluate 4 + 5 and 5 + 4, and then verify that 9 = 9. A good test data set for this case would also include negative numbers and zero.

If the results are to be reliable, the axiom must correctly express the desired feature. Earlier it was considered crucial that the code for the equality operator had to be correct [19]. We have discussed this previously [31], concluding that with testing the equivalence and congruence properties of the equality operator, it can be treated alongside any other function being tested. Another problem appears if the equality operator used in a concept axiom is not implemented. This is known as the oracle problem, and can be handled by techniques based on behavioural equivalence [19, 12], i.e., two values are considered equal if they cannot be distinguished by any operation in the system. The ASTOOT [17] system is based on behavioural equivalence, though in practise the user must still define equivalence, either through an axiom, or by an equals operator in the implementation. Chen et al. [12] describe a system for testing object-oriented programs, and provide a technique for determining behavioural equivalence based on white-box heuristics. In this paper, though, we will assume that an equality operator has been implemented for every type that occurs in the left- or right-hand side of an axiom.

Concepts and axioms are still a work in progress as far as C++ standardisation is concerned[1]. Previous work on C++ axioms has mainly focused on their use for optimisation [4, 45]. Our contributions in this article include:

- a technique for using C++ axioms for testing, and
- a tool to support this technique.

The rest of the paper is organised as follows. In the next two sections we introduce C++ concepts and axioms, and show how to generate test oracles and test code from them. In Section 4, we discuss how to generate test data, both random and user-selected. We finish off with a discussion and conclusion in Sections 5 and 6.

2. Concepts

Concepts [23, 9] allow restrictions to be placed on template arguments. A concept describes a specification for types. It lists the members (functions, associated types, etc.) that are required for some types to model the concept, and the axioms that apply to those members. For example, the following *Monoid*[2] concept requires the existence of an `identity_element` and an operator, and gives an *Identity* axiom (adapted from [9]):

```
concept Monoid<typename T>
    : Semigroup<T> {
  T op(T, T);
  T identity_element();
  axiom Identity(T x) {
    op(x, identity_element()) == x;
    op(identity_element(), x) == x;
} }
```

Axioms are simple conditional equations (or inequalities), universally quantified over the axiom parameters. Multiple equations may be given inside an axiom – they are combined by logical *and*. More complicated axioms, e.g., with existential quantification, cannot be expressed directly. The sides of the equations are full C++ expressions, allowing use of things like the comma operator and calls to any accessible function.

[1] And, in fact, have recently been dropped from the final proposal.

[2] A *monoid* is an algebraic class with an operator $\oplus$ and an identity element e, such that $x \oplus e = e \oplus x = x$. For example, $\langle \texttt{int}, +, 0\rangle$ and $\langle \texttt{int}, *, 1\rangle$ are monoids.

```
concept Indexable<typename A, typename I,
                  typename E>
  : std::EqualityComparable<A,A>,
    std::EqualityComparable<E,E> {
  requires SameShape<A, I>;
  const E& operator[](const A&, const I&);
  E& operator[](A&, const I&);

  axiom ArrayEqual(A a, A b, I i) {
    if (a == b)
      a[i] == b[i];
  }
}
```

Figure 1. The concept *Indexable* has indexing operators and an axiom *ArrayEqual* that states that two if Indexables are equal, then their elements are equal. `A` is an indexable type, `I` is the index type, and `E` is the element type. `A` and `I` are required to be of the same shape, i.e., the values of type `I` are the allowable indices for the type `A`. *SameShape* is a trivial concept used to state that the indexable and index type are of compatible shapes/dimensions.

To state that a set of types model a concept, we use a *concept map*. The concept map can specify a mapping between the implementation names (from the class) and the names used in the concept, and can also be used to add extra code necessary to model the concept. Any functions not mentioned explicitly in the concept map is taken from the context – in many cases the body of a concept map is quite short, or empty. In the concept map below, we state that the `FiniteInt` class of bounded integers models the Monoid concept, and give an operator that returns the addition of elements and an `identity_element` function that returns the `FiniteInt::zero` identity element.

```
template<int size>
concept_map Monoid<FiniteInt<size> >{
  FiniteInt<size> op(const FiniteInt<size>& a,
                     const FiniteInt<size>& b) {
    return a+b;
  }
  FiniteInt<size> identity_element() {
    return FiniteInt<size>(0);
} }
```

Without the concept map body, we would have to provide `op` and `identity_element` for `FiniteInt`s directly.

Concepts may also be declared `auto`, in which case an implicit concept map is provided for any set of types that have the relevant functions declared. We feel it is best to avoid axioms in auto concepts – since they may end up specifying behaviour for functions without the programmer being aware of it (though, a few standard cases like having equality, comparison or assignment operators can probably safely be made auto). We will therefore only generate tests for the cases where the programmer has explicitly used a concept map to declare that the implementation models a concept.

3. From Axioms to Test Code

There are two steps involved in generating tests from concepts. First, we generate a *test oracle* for each axiom in each concept. The test oracle is a function having the same parameters as an axiom, and returning true or false depending on whether the axiom holds for the given arguments.

For example, consider the *Indexable* concept in Figure 1, intended for data structures such as arrays. It has the usual indexing operators you would expect, and an axiom *ArrayEqual*. The axiom

```
template <typename A,typename I,typename E>
requires Indexable<A, I, E>
struct Indexable_oracle
{
  static bool ArrayEqual(A a, A b, I i)
  {
    if (a == b)
      if (!(a[i] == b[i]))
        return false;
    return true;
  }
};
```

Figure 2. Oracle code from the ArrayEqual axiom. The oracle returns immediatly upon failure, otherwise we continue, as there may be more than one equation in the axiom.

can be transformed into callable code by creating a normal C++ template class for the concept (`Indexable_oracle`), and making the axiom a boolean function within that class – see Figure 2.

The second step is to generate test cases for each type that models a concept. This is done by finding all the concept maps within the program, and generating code for each of them. The test case will use data iterators (see Section 4) to iterate through a set of data values for each argument to the axioms, and then call the test oracle for each combination of data values. Success or failure of the oracle test is then reported to the testing framework.

For example, consider an `ArrayFI` class – an array indexed by finite (bounded) integers. A simplified version of the class is shown in Figure 4. It is supplied with two concept maps, relating the implementation to the *SameShape* and *Indexable* concepts. The first stating that any `ArrayFI` of size `size` has the same shape as a `FiniteInt` of size `size` – this is needed to fulfil the *SameShape* requirement of the *Indexable* concept. The second states that `ArrayFI` is Indexable with index type `FiniteInt` and element type `E`. Note that the concept maps are templated, working on any integer `size` and element type `E`.

The test case (seen in Figure 3) consists of an `Indexable_testCase` class specialised for `ArrayFI<size, E>`, `FiniteInt<size>` and `E`. The class contains a test function, `ArrayEqual`, which iterates over the data generators and calls the generic test oracle derived from the axiom. The two outer loops generate arrays (`*a_0` and `*c_0`), while the inner loop generates indexes (`*d_0`). The test oracle (from Figure 2) will check that the array code behaves as expected for an Indexable structure. The `HasDataSet` provides a mapping from a type to a data generator for that type (reasonable defaults for this are generated automatically – see Section 4).

3.1 Reusable Tests

A convenient effect of having concepts and their axioms separate from the classes that implement them is that they can be freely reused for testing new types that model the same concepts. If you already have a *Stack* concept with carefully selected axioms, you get the tests for free when you implement a new stack class.

Having libraries of standard concepts for things such as algebraic classes [22] (including *monoid*, *ring*, *group* and others that apply to numeric data types), containers (indexable, searchable, sorted, ...) as well as common type behaviors [24] (*CopyAssignable*, *EqualityComparable*, ...) cuts down on the work needed to implement tests. A well thought-out library is also far less likely to have flawed or too-weak axioms than axioms or tests written by a programmer in the middle of a busy project.

```
template<int size, typename E>
class ArrayFI {
private:
  E data[size];
public:
  E& operator[](const FiniteInt<size>& i) {
    return data[i];
  }
  bool operator==(const ArrayFI& a){
    for(int i = 0; i<size; ++i)
      if (data[i] != a.data[i])
        return false;
    return true;
  }
  int getSize() const {
    return size;
  }
};

// Any ArrayFI has the same shape as
// a FiniteInt index type if the sizes match
template<int size, typename E>
concept_map SameShape<ArrayFI<size, E>,
                      FiniteInt<size> > { }

// ArrayFI<size,E> is Indexable, with index
// type FiniteInt<size> and element type E
template<int size, typename E>
concept_map Indexable<ArrayFI<size, E>,
                     FiniteInt<size>, E> { }
```

Figure 4. The *ArrayFI* class, parameterised with a size and an element type.

3.2 Concept Combinations

Some combinations of classes can create interesting interactions between concepts. For example, the `FiniteInt` type we used in the implementation of `ArrayFI` satisfies the *Monoid* concept from Section 2 (as well as several other algebraic concepts that are too lengthy to include in this paper). If we extend our `ArrayFI` with element-wise operations, an instance `ArrayFI<FiniteInt>` can 'inherit' the *Monoid* concept from the `FiniteInt`. For this to work, we need to provide a concept map

```
template<typename A>
  requires DefaultIndexable<A>,
         Monoid<DefaultIndexable<A>
              ::element_type>,
         std::CopyConstructible<A>
concept_map Monoid<A> {
  A op(const A& a, const A& b) {
    return Shape<A>::map(
      Monoid<DefaultIndexable<A>
           ::element_type>::op,
      a, b);
  }
  A identity_element() {
    return Shape<A>::build(
      Monoid<DefaultIndexable<A>
           ::element_type>
      ::identity_element());
  }
}
```

```
template <int size,typename E>
requires Indexable<ArrayFI<size, E>, FiniteInt<size>, E>
struct Indexable_testCase<ArrayFI<size, E>, FiniteInt<size>, E>
{
  static void ArrayEqual() {
    typedef HasDataSet<ArrayFI<size, E>>::dataset_type dt_0;
    dt_0 b_0 = HasDataSet<ArrayFI<size, E>>::get_dataset();
    for (DataSet<dt_0>::iterator_type a_0 = DataSet<dt_0>:: begin(b_0)
       ; a_0 != DataSet<dt_0>::end(b_0); ++a_0) {
      typedef HasDataSet<ArrayFI<size, E>>::dataset_type dt_1;
      dt_1 d_0 = HasDataSet<ArrayFI<size, E>>::get_dataset();
      for (DataSet<dt_1>::iterator_type c_0 = DataSet<dt_1>:: begin(d_0)
         ; c_0 != DataSet<dt1>::end(d_0); ++c_0) {
        typedef HasDataSet<FiniteInt<size>>::dataset_type dt_2;
        dt_2 f_0 = HasDataSet<FiniteInt<size>>::get_dataset();
        for (DataSet<dt_2>::iterator_type e_0 = DataSet<dt_2>:: begin(f_0)
           ; e_0 != DataSet<dt_2>::end(f_0); ++e_0)
        check(Indexable_oracle<ArrayFI<size, E>,
                           FiniteInt<size>, E>::ArrayEqual(*a_0,*c_0, *e_0),
             "Indexable", "ArrayEqual");
} } } };
```

Figure 3. Concrete test code generated from a concept map. *HasDataSet* is used to select an appropriate data set for each data type. `check` is a hook for reporting results to a testing framework.

The *Indexable* concept may be used in several different ways on the same array type with different index and element types. As we want the compilation process to automatically deduce which way to index our data structure, we need to provide a default pair of index type and element type to each *Indexable* through the following concept *DefaultIndexable*:

```
concept DefaultIndexable<typename A> {
  typename index_type;
  typename element_type;
  requires Indexable<A, index_type,
                  element_type>;
}
```

Then, for example, `ArrayFI` would only need a small concept map like the one below to inherit all the axioms.

```
template <int size, typename E>
concept_map DefaultIndexable<
               ArrayFI<size, E>> {
  typedef FiniteInt<size>
    index_type;
  typedef E element_type;
}
```

Based on the above concepts and the concept maps, an `ArrayFI<size, FiniteInt>` would have test code for the ArrayEqual axiom (instantiated from the template code in Figure 3), and for the `Monoid::Identity` axiom. And, as `ArrayFI<size, FiniteInt>` is itself a *Monoid*, we can use it as the element type for a new *Indexable Monoid* `ArrayFI<size1, ArrayFI<size2, FiniteInt> >`, and so on[3]. Such constructions are important in some problem domains [32] and allow us to do some simple integration testing with axioms as well.

[3] Which raises the question – how many of these do we generate code for? None and all. Since the generated test code uses templates, the basic test case functions also handle the nested combinations. Only the basic variants are called automatically by our tool's code, though.

3.3 Test Drivers / Suites

So far we have generated test oracles from axioms, and test cases that generate test data and call the oracles. To actually perform the testing, we need to call the test cases as well. There are three ways to do this: we may call the code manually, we may generate code that calls all known test functions, or we may use a combined approach.

By default, our tool will generate a `main` function filled with calls to all non-template test functions. Guessing at sensible template parameters is difficult in the case of unconstrained template parameters and when there is a large or infinite number of choices (as in the case of the nested arrays above). We therefore rely on the user to choose which templated tests to run, as explained in Section 4.

If we want fully automatic test program generation, we could analyse existing application code and find suitable template instantiation arguments there. Or, in cases where template parameters are constrained by concepts, we could generate calls with all classes that fulfil the concept constraint (with a cut-off in place to avoid infinite nesting). This would allow quite exhaustive exercising of code, including combinations that a programmer would likely never think of.

Even if the tool does not automatically generate full test suites, it could help the programmer by generating code templates. With integration into an IDE – such as Eclipse – test suite building can be done in a guided manner.

3.4 Axioms for Object-Oriented Code

The axiom examples we have used so far have mostly been in a functional style where results are returned and there are no side-effects on arguments. Realistic C++ code will often be written in a more object-oriented style.

Object-oriented style favors side-effects on the first argument. To capture side-effects in concepts, some functions will have to have reference type arguments. If the first argument is a reference, then the function can be defined in the concept map as a method defined in a class. If the first argument is not a reference, or it is a `const` reference, then the function is defined as `const` method.

```
concept BoundedStack<typename S> {
  requires std::DefaultConstructible<S>,
    std::EqualityComparable<S>;
  std::EqualityComparable E;
  E top(S);
  E pop(S&);
  void push(S&, E);
  bool full(S);
  bool empty(S);

  axiom PushTop(S s, E e) {
    if(!full(s))
      (push(s,e), top(s)) == e; }
  axiom PushPop(S s, E e) {
    if(!full(s))
      (push(s,e), pop(s)) == e;
    if(full(s))
      (push(s,e), pop(s), s) == s; }
  axiom Empty1() {
    empty(S()) == true; }
  axiom Empty2(S s, E e) {
   if(!full(s))
    (push(s, e), empty(s)) == false; }
  axiom Equal1() {
    S() == S(); }
  axiom Equal2(S s, E e) {
   if(!full(s))
    (push(s, e), s) != S(); }
  axiom Equal3(S s1, S s2) {
   if(!empty(s1) && (s1 == s2))
      s1.top() == s2.top();
   if(!empty(s1) && (s1 == s2))
      (pop(s1), s1) == (pop(s2), s2);
  }
}
```

Figure 5. An example of a bounded stack concept capturing side effect of OO programming style, with a selection of axioms. The comma operator (`,`) is used to first evaluate a call for the side effect (left side), then choosing the value we're interested in (right side). `S()` constructs a new stack.

Non-`const` methods – methods that may change the object – is the norm in C++ programming, which means that function declarations in concepts will often have reference parameters. The side effect of those functions have to be captured somehow by the axiom. The comma operator can be useful for testing side effects. An example is the following axiom:

```
axiom CopyPreservation(T x, U y) {
  (x = y, x) == y;
}
```

This axiom states that after assigning `y` to `x`, the value of `x` should be equal to `y`. The comma operator has the effect of first assigning `y` to `x`, and then yielding the value of `x`.

Figure 5 shows the traditional bounded stack example also used for DAISTS [18], Daistish [34] and JAX [44]. The *BoundedStack* concept is written in a functional syntactic style, but the reference of the first parameter on `pop` and `push` captures the object-oriented style. These two stack operations modify the current object, rather than return a new modified stack.

In our stack axioms, we have intentionally not specified what happens when we attempt to push onto a full stack or pop an empty stack. In a traditional bounded stack, pushing onto a full stack has no effect. By leaving this behaviour undefined, we leave the door open for alternative solutions (handled by non-axiom test cases, for example).

```
  ...
  bool PushFull_help(S s, E e) {
    try { push(s, e); return false; }
    catch(...) { return true; }
  }
  axiom PushFull(S s, E e) {
    if(full(s))
      PushFull_help(s, e) == true;
  }
```

Figure 6. Checking for exceptions. The function `PushFull_help` will return true if pushing onto a full stack throws an exception, and false otherwise.

However, if we wish to specify that an exception should be thrown when attempting to push onto a full stack, we would need a small helper function to do the push, catch the exception and return true or false – see Figure 6. With some small changes [4] to the proposed C++ syntax, we could avoid the use of the helper function.

This state-modifying style of axioms has some consequences for test code generation, since the test oracles will modify the test data. For this reason, the test oracles avoid reference arguments, ensuring that the data is copied into the oracle function. This may not be sufficient for all data structures, though. We are still unsure of the best way to handle this, as we would like to keep data generation as simple and efficient as possible. Fortunately the `const`/non-`const` status of parameters will give a clue as to when this may be a problem – for example, the equals operator is safe, since *EqualityComparable* specifies that it has `const` arguments. We could then try to force copying of test data which is passed as non-`const` arguments in axioms. Alternatively, one could simply expect the test driver to generate fresh data for every oracle invocation.

4. Generating Test Data

Creating a test oracle from the concept axioms and a concept map is straightforward, as described in the previous section. Such a test oracle will normally have parameter variables (free variables) that need to be instantiated by suitable values in order to actually perform testing.

We have three cases to consider when we want to provide data for a free variable:

- The parameter has a known, primitive C++ type.
- The parameter has a known, user-defined type. In this exposition we will not investigate the issues arising if the known type can be subclassed.
- The parameter type is a template argument to the test oracle. In this case, the template may have additional constraints, e.g., that a parameter models a given concept, see Section 3.3.

For the last case we will rely on concept maps to identify candidate types. Though some authors [14] claim that fixing the test data set for one such candidate will be sufficient, we believe test data sets should exercise several of these in order to check that the stated requirements are sufficient constraints on the template arguments.

We provide test data through associating test data generators with each class. For the primitive types, we can use a random generator library, to obtain an arbitrarily large selection of test data. User defined classes should provide a test data generation interface, allowing our testing tool to feed generated test data to the test oracles. A test data generator for a class template may call upon the data generators for the argument classes.

For a known type, whether primitive or user-defined, we see several strategies for providing test data.

1. User selected data sets.
2. Randomly chosen generator terms.
3. Randomly chosen data structure values.

The first is the classical approach to testing and the one (implicitly) favored by test driven development. Here the tester decides, e.g., that integer values -1, 0, 1 and 3 are of prime importance, or that stacks `S()`, `S().push(1)` and `S().push(1).pop()` are specifically important. Such selected data sets are useful for regression testing, where specific data sets that have exposed problems in the past are rechecked with each revision of the code. The data sets can also be targeted for other purposes, e.g., path coverage of the implemented algorithms.

The second is favored by Claessen & Hughes in their QuickCheck system [14] and by Prasetya et al. for their Java-based testing system [39]. The idea is to let random expressions or sequences of (public) methods compute data values of the appropriate type. By choosing a suitable enumeration of terms this will always be possible and give good data coverage. For example, testing integer-like types (with axioms such as associativity, commutativity, distributivity) we may use expressions $0, 0 + 1, (-1 + 0) * 2, \ldots$ and for stacks sequences like `S s(); s.push(1+-2); s.push(3|4); s.pop();` may be used.

The third approach requires the tester to have access to the data representation (data field attributes) of a type. For primitive types such as floats, this means setting the bit patterns of a floating point number directly. For a user-defined class this implies that each data field is given a random value of the appropriate type, subject to the constraints of the implementation. For instance, having a rational number class where we represent rational numbers as pairs of integers (a nominator and a denominator, the denominator different from zero), we may choose random pairs of integers for the attributes, discarding any pair where the denominator part would be equal to 0. Such direct setting of attribute values may give access to a larger range of test values than allowed by method 2, and is needed if all or some of the data fields are publicly available. Setting data attributes directly requires a filtering mechanism that identifies all bad data combinations, i.e., a complete data (class) invariant. If the data invariant has narrow requirements on the data, e.g., that the stack has a length field required to be equal to the length of the linked list representing the data on the stack, independently generating random integers and random linked lists will probably turn up too few good combinations for this technique to be worthwhile.

Harvesting the data produced by an application program is related to the second method, in that it provides values computed by the public methods of the classes, though harvesting ensures a statistical distribution of data much closer to those that appear in practice. One way of harvesting application data would be to insert the test oracles directly as assertions into an application, using the available data values as parameter arguments to the oracle. This would only be safe for stateless data types or copy-assignable data types, otherwise we risk that the oracle itself modifies the state of the application.

Currently, random test data generation seems to be favored by the literature [25, 29, 30]. Studies of testing efficiency seem to indicate that random testing outperforms most other test set designs. For any fixed data set size, a carefully chosen data set will normally be better than a random data set, but a slightly larger, often cited as 20% larger, random data set is often just as good [30]. Random data generation offers an easy route to expand the data set to any reasonable size.

Similarly to the data invariant, a conditional axiom itself represents a filtering mechanism. A conditional axiom contains an if-statement, and only those data combinations that satisfy the condition will really be tried. Assume that we want to test the transitivity axiom for equality on a user-defined rational number type.

```
if (a == b && b == c) a == c;
```

With the representation of rationals as pairs of integers sketched above, we may compute the equality of $\frac{n_1}{d_1}$ and $\frac{n_2}{d_2}$ by the Boolean expression `n1*d2 == n2*d1` involving integer equality. Choosing arbitrary combinations of integers for nominator and denominators, chances are rather slim we ever will get to the truth part in the transitivity axiom. As in QuickCheck we will provide a warning in such cases, encouraging the user to provide data sets where a significant amount of data reaches the body of the condition. On the other hand, only choosing obviously equal nominator and denominator pairs, skews the data set towards trivially satisfying an axiom, and not providing good tests for the algorithms in general.

Claessen & Hughes also point out that different uses of a data type may benefit from different data distributions. The observation being that the data set of integers which best checks that the integers form a monoid, may not be the ideal data set for array sizes when generating finite array test sets. We see this observation on targeted generation of data sets as very important, and expect the locality we have by associating the data generators with each class will provide this flexibility.

Once the test oracles and the test data machinery are in place, it is easy to run the tests by iterating through the corresponding data set for each of the free variables of each test oracle. However, this easily leads to a combinatorial explosion in the testing size. A test set of 100 elements is quite reasonable, but when we test axioms with several free variables this may become a problem. Take the transitivity axiom. It has three free variables, hence we will test it for one million elements altogether. This may be OK for integers, but what about one million finite arrays? We can deal with this by providing the data generators with a parameter related to the number of arguments in an axiom. Our test generator tool can then fill in this parameter automatically based on the number of free variables in an axiom.

4.1 Associating a Data Set with a Type

Any type that is part of a universal quantification on an axiom needs to have a test data set associated with it. Our tool expects the user to configure the data generation with a concept map, where relevant types model a concept *HasDataSet*.

For any type, the user must then provide a concept map for *HasDataSet* with a function `get_dataset`, which provides the iterators needed to obtain values of the type. The `for`-loops in Figure 3 show how `get_dataset` is used to obtain test data. The exact mechanics of iterators and data source (predefined values, random or generated by some other scheme) is up to the user, but the library provided with the testing tool provides a general implementation which can be used as a basis for generating predefined and random values.

5. Discussion

There is no reason to believe that writing axioms (or test cases) is any less error-prone than programming in general. Failure of a test can just as well indicate a problem with the axioms or the equals operator as a problem in the implementation. It is important to be aware of this while programming, so that bug-hunting is not exclusively focused on implementation code. The same issue arises with hand-written tests, though, so this is not specific to axiom-based testing. Also, since axioms have a different form than implementation code (equation versus algorithm), it is unlikely that a bug in an axiom and in the implementation will 'cover' for each other so that neither are detected. It is still possible, though; having several axioms covering related behaviour will make this less likely.

Building libraries of well-tested concepts with axioms will increase confidence in the completeness and correctness of the axioms, and reduces the training needed to make effective use of axioms. Not everyone can be expected to know all the laws governing integer arithmetic – but using an existing axiom library and simply stating that "my class should behave like an integer" is easy.

5.1 Equality Testing

Axiom-based testing (at least with equations) relies on a correct implementation of equality. In many cases, problems with equality will be uncovered in testing, but it is possible to write an implementation of equality that tries to hide most errors – for example, by simply returning true for all arguments (which may be detected when testing inequalities, unless a `!=` operator has been provided with the same problem).

We expect the equals operator to be a *congruence relation* – an equality relation that is preserved by all functions. This means that it has the usual reflexivity, symmetry and transitivity expected of an equivalence relation, with the additional requirement that all equal objects are treated the same by all functions, i.e. $f(a) = f(b)$ if $a = b$ for all a, b, and f.[4] A straightforward bitwise comparison of two objects will often lack this property. In some cases, such as with floating-point numbers, a usable equals operator will not be truly transitive (due to a small amount of 'fuzz' when comparing, to cover up round-off errors) – this has little impact on our use, however.

The *EqualityComparable* concept in the standard library provides axioms for the equivalence relation of the equality operator and also ensures that inequality operator is the negation of the equality.

It may not always be desirable that the equality operator is a congruence. In the cases we want this property, the relevant axioms should be tool generated, since they will involve every method belonging to the class being tested.

A 'bad' equality operator, returning arbitrary results, will almost certainly be caught during testing since it is basically tested by every axiom in the system relevant for the particular type. Trivial cases like equality always returning true is easily caught by testing based on equality axioms, while more subtle bugs may only show up in general testing, and will be more difficult to trace to the equality operator.

Note that having an equality operator is not strictly necessary. Any type that is *EqualityComparable* is observable in our test oracles, *i.e.*, can be tested on equality. But any type that can be projected on an observable type becomes observable. A projection or context is a term with placeholder for a variable [48]. This kind of test oracle generation has not been developed in our tool yet and we for the moment require tested types to be *EqualityComparable*.

Note, though, that even if equality is not generally available for a type, it can be provided in a concept map, thus making it available in any template context where the type is constrained to *EqualityComparable*.

5.2 Algebraic Axioms and Imperative Code

As discussed in Section 3.4, a particular problem occurs for code written in an object-oriented or imperative style, relying on side-effects on arguments. Although this is a poor fit for algebraic-style axioms, side-effects can be captured by using the comma operator. Another issue is that the concept itself must specify whether side-effects occur or not, through the use of non-`const` reference arguments. If an implementation has chosen a different approach, a mapping between the two styles may be given in a concept map, possibly at the expense of an extra temporary. A solution to this problem is provided by *mutification* [5], which automatically maps between algebraic and imperative/OO-style code.

In the ASTOOT [17] system, algebraic specification of object-oriented programs is done in the LOBAS formalism which supports OO syntax. Each axiom relates object states or values that are computed through a sequence of method calls; optionally, observer functions may be called at the end each sequence to inspect the objects. The system is purely algebraic, allowing no side-effects in operations, except for modifying object state in methods – though a relaxation of this is described by Doong and Frankl [15, 16]. ASTOOT will automatically generate test drivers from class interfaces, and also generates test cases from a LOBAS algebraic specification. Automated tests can be augmented by manual test generation.

As the C++ axiom proposal allows arbitrary expressions, the ASTOOT / LOBAS-style can easily be used with C++ axioms; though, without disciplined use within same restrictions, there is a danger that side-effects will interfere with testing, as discussed in Section 3.4.

The ideas of ASTOOT have been developed further by Chen et al., and applied to axiom-based testing of object-oriented code at the level of class clusters and components [12, 13].

5.3 Axiom Selection and Algebraic Specification

Early work by Liskov and Zilles [36] discuss techniques for formal specification of abstract data types. They point out that specification should be done by relating the various operations of the abstract data type, rather than directly specifying the input / output of each operation. The latter leads to over-specification, providing many unnecessary details and hiding the essential properties of the data type – for example, by enforcing some order on the elements of an unordered set. Specifying operations in terms of each other avoids bias towards particular representations or implementations. In traditional unit testing, there is always a temptation to over-specify by focusing on testing the input and output of every operation, though a disciplined developer can still avoid over-specification.

In the context of C++ concepts, the concept is separate from the implementation and should avoid putting undue constraints on how the concept may be implemented. Hence, axiom expressions should be limited to using the operations provided in the concept (together with C++'s primitive operations – on booleans, for example – these can be considered implicitly defined in every concept).

Among the techniques discussed by Liskov and Zilles, algebraic specification [21, 27, 28, 36] shows the most promise in terms of usability and in avoiding over-specification. An algebraic specification consists of a syntax description and a set of axioms; this maps to the C++ idea of concepts, which provide axioms together with a syntax description in the form of associated types and operations.

To ensure that the behaviour of the abstract data type is fully specified (or *sufficiently complete*) one can divide the operations into *constructors* (the set of which can generate all possible values), *transformers* (which can be defined in terms of constructors) and *observers* (which yield values of another type). Left-hand sides for the axioms of a sufficiently complete specification can then be constructed from the combination of each constructor with every non-constructor. Further guidelines for constructing specifications are discussed by Guttag [26] and Antoy [1].

Many of the existing axiom based testing approaches, such as JAX and Daistish, rely on sufficiently complete specifications, provided by complete axiomatisations or initial specifications. This

[4] Informally speaking, since C++ functions may have side-effects or rely on global data.

gives extra properties on which to base tools. For example, the approach of Antoy and Hamlet [2] uses initial specifications, which are evaluated alongside the implementation, as a *direct implementation* [28] of the specification. All objects in the system contain both a concrete value and an abstract value (in the form of a normalised term over constructors in the specification), and the equations from the specification can be evaluated by treating them as rewrite rules on the abstract value terms. A *representation mapping* translates between the abstractions of the specification and the concrete data structures of the implementation. Self-checking functions are made by doing an additional abstract evaluation according to the specification, and – using the representation mapping – comparing the result of normal execution and evaluating the specification. In this way, a whole program can be described and evaluated in two distinct ways – using program code and algebraic specification – providing good protection against programming errors. This is also the disadvantage of the approach – the implementation work must basically be done twice. The overhead of the abstract evaluation and comparison can probably be lowered by running the testing code in a separate thread on a multicore system.

Axioms written in C++ concepts will normally be loose and incomplete, making many of these testing techniques void. The approach described in this paper will work equally well with an incomplete specification (though, it will of course not be able to test unspecified behaviour). Our experience with developing and testing Sophus [31, 32] shows that such axioms are very useful.

5.4 Experiences with Axiom-Based Testing

There is currently no large body of code around that uses C++ axioms, since the standard proposal is not yet finished and compiler support is still not mature.[5] A version of the Matrix Template Library [42] (MTL) with concepts and axioms is in development and we plan to apply our tool to it as soon as it is ready.

We have experience with axiom-based testing from the Sophus numerical software library [31]. This predates C++ axioms, so the tests were written by hand, based on a formal algebraic specification. In our experience, the tests have been useful in uncovering flaws in both the implementation and the specification, though we expect to be able to do more rigorous testing with tool support.

The JAxT tool [33, 35] provides axiom-based testing for Java, by generating tests from algebraic specifications. The axioms are written as static methods and are related to implementation classes through inheritance and interfaces. For any class with axioms, the JAxT tool will generate code that calls the associated axioms. A team of undergraduate students successfully wrote JAxT axioms for parts of the Java collection classes, discovering some weaknesses in the interface specifications in the process [38].

The JAX [44] method of combining axioms with the JUnit [8, 37] testing framework has provided some valuable insight into the usefulness of axiom-based testing. The JAX developers conducted several informal trials where programmers wrote code and tests using basic JUnit test cases and axiom testing, and found that the axioms uncovered a number of errors that the basic test cases did not detect.

Initial experiences with DAISTS [18] were positive and indicated that it helped users to develop effective tests, avoid weak tests, and the use of insufficient test data. With Daistish [34], the authors did trials similar to those done with JAX, with programming teams reporting that their axioms found errors in code that had already been subjected to traditional unit testing. Testing also uncovered numerous incomplete and erroneous axioms – the Daistish team note that this is to be expected since the programmers were students learning algebraic specification. This is probably a factor, but some axiom errors can be expected even from trained programmers.

Further experiences and case studies are summarised by Gaudel and Le Gall [20].

5.5 Tool Implementation

Our implementation is based on the Transformers C++ parsing toolkit [10, 46] and the Stratego program transformation language [11]. We have extended Transformers with the new syntax for concepts and axioms, and written a tool, `extract-tests`, that reads C++ with concepts and generates testing code from the concepts and concept maps in the code [3].

As part of our concepts extension to Transformers, we also have an embedding of the Concept C++ grammar into Stratego, so that Stratego transformation rules can be written using concrete C++ syntax. This makes it easy to modify the code templates for the generated code, for instance, changing the test oracles to report success / failure to a testing framework. As an example we use a back-end for test oracles that instead of returning a boolean, throws an exception for the CUTE library [43] with the line number of the axiom, so we get test results reported within the Eclipse IDE.

Together with the tool, we have a utility library with basic data generation support, and hooks into a testing framework. This library provides a concept for test data generators. Each type to be tested is expected to have an associated data generator specified through a concept map. This allows the user to specify which generator to use, to create any new kind of generator, and finally to combine streams of generated data.

Since compiling Concept C++ is usually slow, and since generating code directly for pure C++ is complex, the tool is delivered with a Concept C++ to C++ tool translation. Though this tool is not complete, it can still give a sufficient translation to be able to work on a big part of Concept C++ with a standard pure C++ compiler.

5.6 Future Work

We have identified several areas for improvement throughout this paper. Areas of particular research interest are:

- Perform proper trials to gauge the effectiveness of axiom-based testing and its impact on development.
- Testing of multi-threaded applications is notoriously difficult [41], and it would be interesting to see if axiom-based testing could be applied here.
- As discussed in Section 4, there are many open issues with data generation. These will likely only be resolved once we apply the method to realistic-sized projects (like MTL).

There are also much engineering work to be done (in no particular order):

- A library of common concepts with axioms should be written. There has been some work on this already [22]. Such concepts should eventually make their way into the C++ standard, for consistency and interoperability.
- Our tool is still experimental, and would need many improvements to be ready for production use. In particular, the underlying framework needs to be developed to handle the kind full-featured C++ code found in mainstream application.
- The tool should be extended with the ability to generate meta-axioms for testing, e.g., congruence axioms for the equality operator or axioms checking the preservation of class invariants in all methods.
- Generate oracles that can test equality on observable types that have no direct equality comparison operator.

[5] The prototype ConceptGCC compiler works well in some cases, but is not complete yet.

6. Conclusion

The use of axioms and "informal formal methods" has seen a surge in popularity recently. We have presented a method for doing axiom-based testing in the context of proposed concept and axiom features for C++, along with a tool to make generation of such tests automatic.

Both the C++ standard, and programming tools such as compilers are still in development and should be considered 'unstable'. However, our initial experiments with simple test cases show promise, and experiences with axiom-based testing from other languages (both our own and others') encourage us to push forward with tool development and larger-scale experiments.

Acknowledgments

This research was funded in part by the Research Council of Norway. We thank the anonymous reviewers and May-Lill Sande for providing us with useful feedback, corrections and references.

References

[1] S. Antoy. Systematic design of algebraic specifications. In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design*, pages 278–280. ACM, New York, NY, USA, 1989. ISBN 0-89791-305-1.

[2] S. Antoy and R. G. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Trans. Software Eng.*, 26(1):55–69, 2000.

[3] A. H. Bagge, V. David, and M. Haveraaen. Axiom-based testing for C++. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 721–722. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-220-7.

[4] A. H. Bagge and M. Haveraaen. Axiom-based transformations: Optimisation and testing. In J. J. Vinju and A. Johnstone, editors, *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier, Budapest, Hungary, 2008.

[5] A. H. Bagge and M. Haveraaen. Interfacing concepts: Why declaration style shouldn't matter. In T. Ekman and J. J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, Electronic Notes in Theoretical Computer Science. Elsevier, York, UK, March 2009.

[6] K. Beck. Extreme programming: A humanistic discipline of software development. In *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE'98)*, pages 1–6. Springer Berlin / Heidelberg, 1998.

[7] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002. ISBN 0321146530.

[8] K. Beck and E. Gamma. JUnit – Java Unit testing. `http://www.junit.org` and `http://junit.sourceforge.net/` per 2007-03-15.

[9] P. Becker. Working draft, standard for programming language C++. Technical Report N2857=09-0047, JTC1/SC22/WG21 – The C++ Standards Committee, March 23rd, 2009.

[10] A. Borghi, V. David, and A. Demaille. C-Transformers: A framework to write C program transformations. *ACM Crossroads*, 12(3), April 2006.

[11] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: Components for transformation systems. In F. Tip and J. Hatcliff, editors, *PEPM'06: Workshop on Partial Evaluation and Program Manipulation*. ACM Press, January 2006.

[12] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 7(3):250–295, 1998. ISSN 1049-331X.

[13] H. Y. Chen, T. H. Tse, and T. Y. Chen. Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Trans. Softw. Eng. Methodol.*, 10(1):56–109, 2001. ISSN 1049-331X.

[14] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM, New York, NY, USA, 2000. ISBN 1-58113-202-6.

[15] R.-K. Doong. *An approach to testing object-oriented programs*. PhD thesis, Polytechnic University, Brooklyn, NY, USA, 1993.

[16] R.-K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 165–177. ACM Press, New York, NY, USA, 1991. ISBN 0-89791-449-X.

[17] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2): 101–130, 1994. ISSN 1049-331X.

[18] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981. ISSN 0164-0925.

[19] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995. ISBN 3-540-59293-8.

[20] M.-C. Gaudel and P. L. Gall. Testing data types implementations from algebraic specifications. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 209–239. Springer, 2008.

[21] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978.

[22] P. Gottschling. Fundamental algebraic concepts in concept-enabled C++. Technical Report TR639, Department of Computer Science, Indiana University, 2006.

[23] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310. ACM, New York, NY, USA, 2006. ISBN 1-59593-348-4.

[24] D. Gregor and A. Lumsdaine. Core concepts for the C++0x standard library (revision 2). Technical Report N2621=08-0131, JTC1/SC22/WG21 – The C++ Standards Committee, May 19th, 2008.

[25] W. J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5): 661–674, 1999. ISSN 0098-5589.

[26] J. V. Guttag. Notes on type abstraction (version 2). *IEEE Trans. Softw. Eng.*, 6(1):13–23, 1980. ISSN 0098-5589.

[27] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10:27–52, 1978.

[28] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12):1048–1064, 1978. ISSN 0001-0782.

[29] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, 1990. ISSN 0098-5589.

[30] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[31] M. Haveraaen and E. Brkic. Structured testing in Sophus. In E. Coward, editor, *Norsk informatikkonferanse NIK'2005*, pages 43–54. Tapir akademisk forlag, Trondheim, Norway, 2005. URL `http://www.nik.no/2005/`.

[32] M. Haveraaen, H. A. Friis, and H. Munthe-Kaas. Computable scalar fields: a basis for PDE software. *Journal of Logic and Algebraic Programming*, 65(1):36–49, September-October 2005.

[33] M. Haveraaen and K. T. Kalleberg. JAxT and JDI: the simplicity of JUnit applied to axioms and data invariants. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 731–732. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-220-7.

[34] M. Hughes and D. Stotts. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 53–61. ACM Press, New York, NY, USA, 1996. ISBN 0-89791-787-1.

[35] K. T. Kalleberg. *Abstractions for Language-Independent Program Transformations*, chapter 11. University of Bergen, Norway, Postboks 7800, 5020 Bergen, Norway, June 2007. ISBN 978-82-308-0441-4.

[36] B. Liskov and S. Zilles. Specification techniques for data abstractions. In *Proceedings of the international conference on Reliable software*, pages 72–87. ACM, New York, NY, USA, 1975.

[37] P. Louridas. JUnit: Unit testing and coding in tandem. *IEEE Softw.*, 22(4):12–15, 2005. ISSN 0740-7459.

[38] M. Masood, E. Birkenes, K. T. Kalleberg, M. Haveraaen, and A. H. Bagge. Axiom-based testing of Java collections with JAxT. Technical Report 388, Department of Informatics, University of Bergen, P.O.Box 7803, N-5020 Bergen, Norway, August 2009. URL `http://www.ii.uib.no/publikasjoner/texrap/`.

[39] I. S. W. B. Prasetya, T. E. J. Vos, and A. Baars. Trace-based reflexive testing of OO programs. Technical Report UU-CS-2007-037, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2007. URL `http://www.cs.uu.nl/research/techreps/repo/CS-2007/2007-037.pdf`.

[40] D. Saff. Theory-infected: or how I learned to stop worrying and love universal quantification. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 846–847. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-865-7.

[41] K. Sen. Effective random testing of concurrent programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-882-4.

[42] J. G. Siek and A. Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 466–467. Springer-Verlag, London, UK, 1998. ISBN 3-540-65460-7.

[43] P. Sommerlad. *C++ Unit Testing Easier*, 2009. URL `http://r2.ifs.hsr.ch/cute`.

[44] P. D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In D. Wells and L. A. Williams, editors, *XP/Agile Universe*, volume 2418 of *Lecture Notes in Computer Science*, pages 131–143. Springer, 2002. ISBN 3-540-44024-0.

[45] X. Tang and J. Järvi. Concept-based optimization. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 97–108. ACM, New York, NY, USA, 2007. ISBN 978-1-60558-086-9.

[46] The Transformers Group, LRDE, EPITA. *Transformers*, 2008. URL `http://www.lrde.epita.fr/cgi-bin/twiki/view/Transformers/Transformers`.

[47] B. Yu, L. Kong, Y. Zhang, and H. Zhu. Testing java components based on algebraic specifications. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 190–199. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3127-4.

[48] H. Zhu. A note on test oracles and semantics of algebraic specifications. In *QSIC '03: Proceedings of the Third International Conference on Quality Software*, pages 91–98. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-2015-4.