Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Testing with Axioms in C++ 2011

Anya Helene Bagge[a] Valentin David[a] Magne Haveraaen[a]

a. Bergen Language Design Laboratory, Department of Informatics, University of Bergen, Norway, http://bldl.ii.uib.no/

Abstract Unit testing is an important part of modern software development, where individual code units are tested 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, particularly when testing generic code, and focusing on individual tests can distract from creating tests that cover the full functionality.

Concepts provide a generic way of describing code interfaces for generic code. Together with *axioms*, program behaviour can be specified algebraically in a formal or semi-formal way.

In this paper we show how concepts and axioms can be expressed in standard C++ 2011, and explore how to generate generic unit tests, by treating the axioms as code to be evaluated with test data. We also show a generic way to generate test data for axiom-based testing in C++ 2011.

Keywords Algebraic Specification; Axiom-Based Testing; Axioms; C++; C++0x; C++11; Concepts; Generative Programming; Mouldable Programming; Test Generation; Unit Testing

1 Introduction

Modern software engineering practises encourage the use of unit testing to increase software reliability. Test-driven development (TDD) [Bec02] 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 and unlikely 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.

Anya Helene Bagge, Valentin David, Magne Haveraaen. *Testing with Axioms in C++ 2011.* In Journal of Object Technology, vol. 10, 2011, pages 10:1–32. doi:10.5381/jot.2011.10.1.a10

1.1 Testing with Concepts and Axioms

We suggest writing tests based on axioms that formally specify expected behaviour, rather than relying on *ad hoc* test cases. We integrate axioms with concepts that describe code interfaces – basically, the types and operations that a program module supports. Axiom-based testing provides reusable tests for implementations of concepts. Also, by stating the requirements of generic code in terms of concepts, we know what behaviour we must test for in arguments given to generic code.

Testing an implementation consists of stating (and checking) that it conforms to the interface of the given concept, then generating suitable test data, and finally evaluating the axioms for the generated test data values using the implementation. If any of the axioms evaluate to false, the implementation does not satisfy the concept specification – which may mean that the implementation is buggy, or that the specification is wrong; if the results are to be reliable, the axiom must correctly express the desired feature.

1.2 Concepts in C++ 2011

Concepts were proposed for inclusion in the C++11[1] standard as a modularisation feature for generic programming. Concepts were designed to solve the problem of incomprehensible error messages from deep inside C++ template code, providing a sort of bounded polymorphism to the otherwise duck typed (at compile time) template system, similar to how type classes work in Haskell [HHPW96]. The proposal also allowed for axioms in concepts, opening the way to axiom-based optimisation [TJ07, BH09] and testing [BDH09]. However, concepts were dropped [Str09] from the draft standard [B+11b] before finalisation.

Since the proposed C++ standard no longer provides for concepts and axioms, our approach has changed since our original implementation of axiom-based testing for C++ [BDH09]. We now provide a library, CATSFOOT, which adds alternative concept support to C++11 as well as support for generating axiom tests, all using template meta-programming. While we previously had to rely on an external tool to support test generation, with limited C++ support, this is no longer necessary.

Our contributions in this article include:

- a technique and library for testing with concepts and axioms in C++11,
- a library for using concepts in C++11, and
- a generic test data generation library to be used with axiom-based testing.

Our CATSFOOT concept library supports most of the features of the previous C++0x draft [GSSW08] (with the exception of archetypes), though in this paper we will focus mainly on using it for axiom-based testing, leaving the rest for a future paper. Further information on the concept library, as well as the full source code released under the GNU LGPL license, is available at `http://catsfoot.sourceforge.net/` [Dav11].

We have tested all code examples in this paper with GCC 4.5.2 and 4.6.0 (with the exception of one example, which does not run on current compiler releases). The file containing the examples can be obtained from the website.

[1] At the time of writing, the new C++ Final Draft International Standard [B+11a] was awaiting review and approval by ISO before publication.

The rest of the paper is organised as follows. In the next section, we introduce and define concepts and axioms, and in Section 3 we show how to perform axiom-based tests. Data generators are explained in Section 4, and in Section 5 we discuss specifications, different aspects of axiom-based testing and related work. The conclusion is given in Section 6. There are also two appendices, Appendix A containing code and examples referenced in the paper, and Appendix B which gives a quick overview of the C++ definitions used in the paper.

2 Concepts

We will now introduce definitions of the terms we will use throughout the remainder of the paper.

2.1 Concepts

A *concept* $C\langle p_1, p_2, \ldots, p_n\rangle = (R, \Phi)$ consists of a set of *parameters* $p_1, p_2, \ldots, p_n$, a set of *requirements* R and a set of *axioms* Φ.

The concept parameters may be either types or operations, both encoded in the C++ type system. For example, the *monoid<T,Op,Id>* concept has three parameters: a type `T`, an operation `Op` and an identity element `Id`, expressed as a nullary operation. In C++, we write concepts as a template class inheriting from `concept`:

```
1  template <typename T, typename Op, typename Id>
2  struct monoid: public concept {
```

We will use the concept `monoid` as a continuing example throughout this section.

2.2 Requirements

A *requirement* on one or more concept parameters may be either a *predicate* or another concept. A predicate is a template meta-function which checks some property of a type or function. For example, the predicate `is_callable` checks whether a function accepts the given parameter list. This is compatible with all the predicates defined in `<type_traits>` of the standard library, such as `std::is_convertible` and `std::is_constructible`.

The requirement list is defined as a `typedef requirements`, with a `concept_list` listing the requirements:

```
3    typedef concept_list<
4      // operations are callable with the given parameter types
5      is_callable<Op(T, T)>,
6      is_callable<Id()>,
7      // results are convertible to T
8      std::is_convertible<typename is_callable<Op(T, T)>
9                          ::result_type, T>,
10     std::is_convertible<typename is_callable<Id()>
11                         ::result_type, T>
12   > requirements;
```

Encoding the requirements as a type in this way is a standard C++ meta-programming trick, making them accessible to the template system at compile time. The member `result_type` gives the return type of a `is_callable` operation.

New concepts can be built from simpler ones by using concepts as requirements. For example, `monoid`s for addition and multiplication could be part of a larger `ring` concept (see Listing 2, page 10).

2.3 Axioms

An *axiom* performs some test on the relationship between the operations in the concept. Each axiom contains one or more expressions or *assertions* with universally quantified variables of the concept parameter types. For example, the *monoid* concept has axioms for associativity (`op(x, op(y, z)) == op(op(x, y), z)`) and identity (`op(x,id()) == x` and `op(id(),x) == x`) for all values `T x, y, z`, operators `Op op` and `Id id`.

Our axioms are represented as static member functions, with the universally quantified variables as parameters. When running tests, we can then supply the axiom functions with concrete test values of the appropriate types.

```
13   static void associativity(const Op& op, const T& a,
14                             const T& b, const T& c) {
15     axiom_assert(op(a, op(b, c)) == op(op(a, b), c));
16   }
17
18   static void identity(const Op& op, const T& a, const Id& id) {
19     axiom_assert((op(id(), a) == a) && (op(a, id()) == a));
20   }
```

The `axiom_assert` call will take care of recording success or failure, and computing test coverage. The `axiom_assert` argument is the actual axiom. Multiple such assertions can be put into the same function, and the programmer is free to use `if` to create conditional axioms or, in fact, use arbitrary C++ code to compute the axiom result.[2] Hence, we are not restricted to a particular form of logic (*e.g.*, conditional equational logic, as used in the C++0x concept proposal).

The equality operator used in the axiom is the normal C++ equality defined for the type (if it exists). In some cases it may be a good idea to make the equality operator itself a parameter of the concept, so that the programmer has more control over the semantics of equality. Similarly, other comparison operators should be parameters to the concept, since they may not be defined for all types. See Section 5.6 for more about defining and using equality.

The list of all axioms is obtainable through the function `get_axioms()`, which the concept programmer must implement. For convenience, the `AXIOMS` macro is used to implement `get_axioms()` correctly, given a list of axioms:

```
21   AXIOMS(associativity, identity);
22 }; // end of concept monoid
```

[2]Note that `axiom_assert` is currently defined as a macro, and in order to correctly determine axiom coverage, it should not be placed inside a `{ }` block.

2.4 Models

A *model* or *implementation* is a list of actual concept arguments $\langle a_1, a_2, \ldots, a_n \rangle$ that fulfil all the requirements of a concept and satisfy all the axioms. A concept expression $C\langle a_1, a_2, \ldots, a_n \rangle$ is *verified* if $\langle a_1, a_2, \ldots, a_n \rangle$ models C, with the actual arguments substituted for the formal concept parameters. For example: $\langle \mathbb{Z}, +, 0 \rangle$ models *Monoid*. In C++, we express this using a type trait `verified`:

```
template <>
struct verified<monoid<int, op_plus, constant<int,0> > >
 : public std::true_type
{};
```

The programmer must explicitly state that the model relationship holds.

The actual code implementing a model can – and often will – be generic template code. All operations are wrapped as types – classes implementing `operator()` – when they are used in concepts. This makes it easier to deal with them at the template level. Our library contains wrappers for all standard C++ operators, such as `op_plus` for `+`, `op_lt` for `<`; and `constant<T,V>` for constants; as well as convenience macros for wrapping both member and non-member functions.

2.5 Axiom-Based Testing

We are now ready to define axiom-based testing: *Axiom-based testing* attempts to demonstrate that an implementation $\langle a_1, a_2, \ldots, a_n \rangle$ models a concept C (or, equivalently, that `verified<`$C\langle a_1, a_2, \ldots, a_n \rangle$`>` is true), by experimentally searching for counter-examples using the axioms of the concept and the data values and operations of the model.

Testing is controlled separately from the model relationship, as we shall see in Section 3, but each `verified` declaration implies an obligation on the programmer to verify by testing (or proof) that the model relationship holds.

2.6 Predicates

Predicates can be used as part of the requirement list for a concept. A predicate is implemented as a type which has a static member `::value` as a Boolean constant, the value of which is defined based on whether certain syntactic requirements are fulfilled. For instance, a predicate can tell whether a type is callable with a certain signature.

The header `<type_traits>`, introduced in the C++11 standard, provides predicates which can be used with our system. For example, `is_lvalue_reference` is a predicate that tests whether a type represents an lvalue. It is implemented with a default case, returning `false`:

```
template <typename T>
struct is_lvalue_reference: public std::false_type {
};
```

and a specialised case for references, returning `true`:

```
template <typename T>
struct is_lvalue_reference<T&>: public std::true_type {
};
```

An *automatic concept* (or auto concept) has no axioms, and is automatically verified for all models that match the requirements. An auto concept must inherit from `auto_concept`, and differs from a predicate only in how errors are reported in concept checking; for a predicate, the error is reported directly, for an auto concept, the message will refer to the specific requirement that failed. Auto concepts and predicates have no role in testing, they are only used in concept checking. An example can be seen in Listing A.5.

2.7 Concept Checking

Concept checking is the act of determining whether an implementation satisfies the requirements of a concept [SL00]. Syntactic requirements, such as which operations should be defined, are expressed through predicates and can be checked by seeing if the predicate has a true value. For semantic requirements (axioms), we must rely on the programmer promising that the requirements are fulfilled, using the *verified* trait. Concepts and auto concepts in the requirement list are checked recursively.

Concept checking is performed prior to testing, and whenever requested by the programmer using `assert_concept` (for function templates) and `class_assert_concept` (for class templates) to check that a set of template arguments model a given concept, or when using `ENABLE_IF` to perform concept based overloading. The concept asserts will trigger an error if the given concept expression fails the requirements. `ENABLE_IF` is used for concept-controlled polymorphism [JWL03]; if inserted into a function template argument list, it will disqualify the function from overload resolution if concept checking fails. An example of `ENABLE_IF` can be seen in Listing A.5; more information on non-testing use of the concept library can be found in the Catsfoot tutorial and source distribution [Dav11].

2.8 Long Parameter Lists

Compared to the previous concept proposal [GSSW08], we make wider use of parameters; basically every type and operation in a concept is a parameter. This allows for far more flexibility in matching a wide range of implementation styles with the interface defined in a concept (*i.e.*, by providing *signature morphisms*), eliminating the need for the `concept_map` construct of the previous proposal. The cost of this is somewhat more unwieldy parameter list, particularly for large concepts. We can reduce this problem by making specialised version of concepts that bind some of the arguments. For example, the `plus_monoid` in the following example binds the operator to `op_plus` and the identity element to the default constructor:

```
1  template <typename T>
2  struct plus_monoid: public concept {
3    typedef
4      monoid<T, op_plus, wrapped_constructor<T()> >
5      requirements;
6  };
7
8  template <typename T>
9  struct verified<monoid<T, op_plus, wrapped_constructor<T()> > >
10  : public verified<plus_monoid<T> >
11 {};
```

The `verified` clause then states that if the user claim a type to be model of `plus_monoid` it is also a `monoid`.

3 Testing Axioms

As we have seen in Section 2.3, an axiom is just a function that calls the macro `axiom_assert`. This macro will throw an exception back to the test driver if the axiom test fails, and do nothing on success. The user may redefine the macro to fit the programming environment.

Our previous axiom-based testing scheme [BDH09] would generate test drivers by parsing the concepts with external tools. However, C++11 provides a way to make drivers using pure template meta-programming, with no external tool. Using *variadic templates*, introduced in the standard, it is now possible to match a function with arbitrary number of parameters and go through all its parameter types and select the right data to pass as arguments.

We can test a single axiom in the following way:

```
test(generator,
     monoid<int, op_plus, constant<int, 0>>::associativity);
```

where `associativity` is an axiom of the `monoid` concept (instantiated for $\langle \mathbb{Z}, +, 0 \rangle$), and `generator` is a data generator (described in Section 4). The `test` function will go through all the parameters of the axiom and get suitable argument values from the generator, then call the axiom and report the outcome.

A simplified version of the test driver is shown in Listing 1. It shows how it is possible to parse a parameter list of a function at compile-time, without the use of reflection.

3.1 Testing Concepts

Testing just a single axiom at a time would be cumbersome. Using the `test_all` function, we may automatically test all the axioms of a concept – including the axioms of any recursively required concept:

```
test_all<monoid<int, op_plus, constant<int, 0>>>(generator);
```

The `test_all` function obtains a list of all axioms in a concept from the `get_axioms` function (defined by the user using the `AXIOMS` macro). This list also contains the name of each axiom as a string, used to print a short report on success. Each axiom is tested in turn using `test`. After all axioms have been tested, `test_all` is called recursively on all the requirements of the concept.

Support for concept testing is the reason for the `AXIOMS` macro. There is no way to find the members of a class using template meta-programming, so the programmer must supply the list of axioms manually. This also implies that we can define axioms outside of concepts (*i.e.*, as non-member functions), should we want to.

Such free-floating axioms can either be added to a concept by naming them in the `AXIOMS` argument list, or be tested directly with `test`, as in the previous section. This can be an easy way to add axioms to a project, without reorganising the code around concepts.

```
1  template <>
2  struct tester<> {
3    template <typename Generator, typename Fun, typename... Params>
4    static bool call(Generator, Fun f, Params... values) {
5      try {
6        f(values...);
7        return true;
8      } catch (axiom_failure af) {
9        return false;
10     }
11   }
12 };
13
14 template <typename T, typename... U>
15 struct tester<T, U...> {
16   template <typename Generator, typename Fun, typename... Params>
17   static bool call(Generator g, Fun f, Params... values) {
18     auto container = g.get(selector<T>());
19     for (auto i = container.begin();
20          i != container.end(); ++i) {
21       if (!tester<U...>::call(g, f, values..., *i))
22         return false;
23     }
24     return true;
25   }
26 };
27
28 template <typename Generator,
29           typename... T>
30 bool test(Generator g, void f(T...)) {
31   return tester<T...>::call(g, f);
32 }
```

Listing 1 – The function `test` starts by calling `tester<T...>::call` where `T...` represents the list of the parameters of the axiom. Then this function `tester<T...>::call` will call itself recursively, removing the first parameter until it reaches an empty list. In the mean time values of each type will stack up, building an argument list. At the end, the `call` static member will receive a value of each type and can then call the axiom.

```
code.cc:447: Axiom static void monoid<T, Op, Id>::identity(const Op&,
    const T&, const Id&) [with T = int, Op = op_plus, Id = constant<
    int, 1>] failed.

Expression was: (op(id(), a) == a) && (op(a, id()) == a)

Values were:
 * Values of type op_plus are not printable
 * -1
 * Values of type constant<int, 1> are not printable
```

Figure 1 – Axiom failed report for $monoid\langle\mathbb{Z}, +, 1\rangle$. The library gives similar reports for the other monoid axioms and test values.

3.2 Test Reports

On successful completion of a test, the default test driver will write '`passed`' to the console. An alternative implementation can easily provide feedback to a unit testing framework or integrated editor. Any failures will result in a detailed problem report. For example, if we try to see if $\langle\mathbb{Z}, +, 1\rangle$ is a monoid (which it isn't, since $x + 1 \neq x$):

```
test_all<monoid<int, op_plus, constant<int, 1>>>(generator);
```

we get an axiom failed message, as shown in Figure 1.

As explained in Section 2.3, evaluation of axioms is done by `axiom_assert`. If the axiom test succeeds, the axiom function will proceed and eventually return, otherwise `axiom_assert` will throw an exception. This exception contains the argument values and all other necessary information to generate the report – including the line number and full name of the axiom, obtained using the compiler's predefined macros. This detailed information is available only on failure; the success message will simply report the name of the axiom.

The printing of values depends on the `<<` operator being defined for the value and the error stream. This is handled using concept-based overloading; the code for this is shown in Listing A.5.

3.3 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 [Got06] (including *monoid*, *ring*, *group* and others that apply to numeric data types), containers (indexable, searchable, sorted, ...) as well as common type behaviour predicates [GMWL08] (defined in `<type_traits>` in C++11) 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 compared to axioms or tests written by a programmer in the middle of a busy project.

Another typical form of reuse is building more complex concepts on top of simpler ones. For example, we may build a *ring* from *monoid* and *group*, with the operators

```
1  template <typename T, typename MOp, typename AOp,
2            typename Minus, typename Zero, typename One>
3  struct ring: public concept {
4    typedef monoid<T, AOp, Zero> add_monoid;
5    typedef group<T, AOp, Minus, Zero> add_group;
6    typedef monoid<T, MOp, One> mul_monoid;
7
8    typedef concept_list<
9      mul_monoid,
10     add_group, // implies add_monoid
11     distributive<T, MOp, AOp>,
12     commutative<T, AOp>
13     > requirements;
14
15   // check that we also have add_monoid
16   class_assert_concept<add_monoid> check;
17 };
```

Listing 2 – An algebraic *ring* concept, built from *monoid* and *group*. The requirement `group<T, AOp, Minus, Zero>` should imply `monoid<T, AOp, Zero>`; we check that this is really the case with the assert at the end of the concept.

satisfying the distributive law (*i.e.*, $a \times (b + c) = a \times b + a \times c$) – see example in Listing 2. This may be an interesting property to test; for instance, integers are rings, but the `int` data type may not behave like a ring on architectures where numbers do not wrap around on overflow.

3.4 Axioms for Object-Oriented (OO) code

To be as generic as possible and to support all programming paradigms in C++, we have stayed away from object orientation in our approach. In general, we recommend that axioms are written as algebraic expressions. OO programming can be handled by wrapping object member functions into types, and imperative functions can also be wrapped to avoid side-effects on arguments.

Listing 3 shows two examples of how member functions are wrapped into types. In the first case, the class `foo_wrapped` provides a wrapper around any member function `Ret T::foo(Args...)`, as a call with signature `Ret (T&, Args...)`; *i.e.*, with the object as the first parameter. Similarly, the constant member function `Ret T::foo(Args...) const` is wrapped as `Ret (const T& Args...)`, with a `const` first parameters.

We can generate such wrappers using the `DEF_MEMBER_WRAPPER` macro. For example:

```
DEF_MEMBER_WRAPPER(foo);
```

The macro only needs to be called once per function name, as it will wrap any member by that name, in any class, with any argument list. Unfortunately, there does not seem to be a way to trigger the generation of such wrappers automatically.

Axioms often use functional style programming, potentially causing trouble with

```
1  struct foo_wrapped {
2    template<typename T, typename... Args,
3             typename Ret = decltype(std::declval<T>()
4                           .foo(std::declval<Args>()...))>
5    Ret operator()(T&& object, Args&&... args) const {
6      return std::forward<T>(object).foo(std::forward<Args>(args)...);
7    }
8  };
9
10 struct foo_functionalized {
11   template<typename T, typename... Args,
12            typename = decltype(std::declval<T>()
13                        .foo(std::declval<Args>()...))>
14   T operator()(const T& object, Args&&... args) const {
15     T ret(object);
16     ret.foo(std::forward<Args>(args)...);
17     return ret;
18   }
19 };
```

Listing 3 – Examples of method wrappers. The type parameter with a `decltype` expression will both be used to infer the return type and to enable the call only when the argument list is legal for the original member function.

member functions that update their object. Members with side effect on the object can be converted into functions – *functionalised* – returning the modified copy and removing the side effect. In the example in Listing 3, `foo_functionalized` provides a functionalised wrapper for `foo`.

Similarly, if a function has side-effects on its arguments, we can wrap it to avoid the side effects. The wrapper will proceed through the argument list, try to make each argument `const`, and simply copy it if that fails (this requires, of course, that a copy constructor is defined for the type). Functions with arbitrary global side effects cannot be wrapped; but such behaviour would be difficult to describe with axioms anyway.

3.4.1 Inheritance

According to the Liskov substitution principle, functions expecting an object of a base class should behave consistently when presented with an object of a subclass. Hence, the subclass must satisfy the behaviour specification of the base class(es); *i.e.*, it must model at least the same concepts, and we should test it with (at least) the same axioms. The subclass may also model additional concepts, with more axioms to test.

During testing, one may treat subclasses just like any other class, and test with the applicable concepts (the base class concepts and the concepts of the subclass itself). If one wishes to specifically test the inheritance and virtual dispatch behaviour, it is possible to run the test on references instead of values:

```
test_all<base_concept<base_class&>>()(generator);
```

and then use a data generator that provides object of both the base class and the subclasses. Such a generator can be built using the random term generator (Section 4.2) by supplying operations that construct objects of the subclasses (using the term generators for the subclasses).

3.5 Exceptions

Depending on the approach taken when writing specifications, it may be interesting to test exception behaviour. Lack of proper error detection and reporting is a common and serious problem, and we may want to test that improper use of operations results in an exception being thrown – for example, when popping an empty stack or creating a rational number with a zero denominator.

The function `throwing<E>` can be used to test if an operation throws a particular exception. It is instantiated with the exception type, and takes the operation and arguments as arguments. For example:

```
axiom_assert(throwing<Exception>(fun, 0));
```

Of course, this is only necessary if we write axioms in a purely functional style; otherwise we may use `try`/`catch` directly in the axiom.

Under the substitution principle, it is usually considered allowable for an implementation or an implementation subclass to accept a wider range of parameter values than stated in the specification (in accordance with the principle of the previous section), so exception testing in this manner should be used with care. A possibility is to separate the specification of exception behaviour into another concept, or specify preconditions in some other way.

3.6 Test Coverage

We can get some idea of the effectiveness of our axiom tests by measuring axiom coverage and implementation code coverage. Axiom coverage tells us how much the axioms have actually been tested, and can uncover problems like axioms that are untested or under-tested because their conditions are never true on the given test data. Code coverage of the implementation gives an indication of how good the axioms and the test data are at exercising all parts of the code. Ideally we want test sets that cover the entire implementation, including seldom used branches of `if` statements and error handling code – possibly using other tests in addition to the axiom based tests.

Our testing library measures axiom coverage and will warn about unused axioms. We can measure code coverage using an external tool like `gcov`(1) or one of the various commercial tools that are available.

Full axiom and code coverage is not necessarily sufficient to give confidence in the implementation; if we have only used a few data points, a trivial faulty implementation may still pass the test, even though it would easily break when exposed to a more diverse selection of test data. We have no way to detect this directly, though we could give warnings if tests are done with few data points.

Lack of full axiom coverage indicates a lack of diverse test data. Low code coverage may also stem from a lack of test data or a deficiency in how the data is generated, but is more likely due to not covering the full behaviour with axioms. This may not be a problem in itself; it may be perfectly acceptable to cover some parts of the code with other tests.

4 Data Generation

The second main component of axiom-based testing is *data generation*.

There are several possible strategies for generating test data: We may use user selected data sets, randomly chosen generator terms, randomly chosen data structure values, or data values harvested from an application.

The first option is the classical approach to testing and the one (implicitly) favoured by test driven development. It relies on picking representative values that exercise the full behaviour of the code. We may do this by guessing (*e.g.*, pick common boundary values like $-1, 0, 1$ and a few other values), reasoning (*e.g.*, this function does something different for positive and negative numbers, so pick one positive and one negative number, plus zero to test the boundary – path coverage analysis can give useful input here), and by using values that have caused problems in the past, to expose regressions.

There are two main approaches to random testing: generate random terms or expressions and use their values, or generating random data for each field (member variable) of a class. QuickCheck [CH00] uses term generation, as does Prasetya *et al.* for their Java-based testing system [PVB08]. Term generation assumes that all data values can be generated by some sequence of the available operations – which is true for algebraic data types, but not necessarily for all C++ classes (*i.e.*, if the data structure fields can be manipulated directly).

Random field value generation has its own problems. There is often a particular relationship between the fields of a class, so that only some combinations are valid (satisfies the data or class invariant). In practice, we may have to implement a specific data generator for each class, to ensure that sensible and valid data is generated.

Harvesting the data produced by an application program is related to the term generator 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 practise. One way of harvesting application data would be to insert the axiom tests directly as assertions into an application, using the available data values as parameter arguments to the axiom. This would only be safe for functional-style axioms, or with stateless data types or copy-assignable data types, otherwise we risk that the axiom itself modifies the state of the application.

Studies of testing efficiency seem to indicate that random testing often outperforms other test set designs [Gut99, HT90, Ham94]. 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 [Ham94]. Random data generation offers an easy route to expand the data set to any reasonable size.

Our data generator library offers support for user selected data sets, and a combination of the two random data generator methods: term generation with user-provided operations. Harvesting program data or instrumenting with axiom tests is difficult to achieve without using an external tool.[3]

For advice on selecting data for testing, we refer the reader to any good book on testing, such as Myers [Mye79].

[3]Though we may be able to do this for constrained template code that makes calls through wrappers.

```
1  auto generator =
2    choose
3    (list_data_generator<int>
4     ({-1, 0, 1, 2, 3,
5         std::numeric_limits<int>::min(),
6         std::numeric_limits<int>::max()}),
7     default_generator());
```

Listing 4 – Generating integers from a static set of values.

4.1 Data Generator Library

As we have seen in Section 3, we need to pass a data generator to the `test` or `test_all` test drivers. In our system, a *generator* produces, for some type `T`, a fresh copy of a container of data elements of that type. The library provides three basic generic generators:

`default_generator` generates singletons from the default constructor of the type. This is useful for wrapped state-less operations, for example.[4]

`list_data_generator<T...>` generates lists of values which do not change. This generator is initialised by the content of those lists. This is useful for testing boundary or extreme conditions, where the desired values must be determined by the programmer.

`term_generator<T...>` generates random values based random terms of a given signature. The generator is initialised with a C++ random generator (see header `<random>` in the standard) and a list of operations that makes the signature. We describe this generator in Section 4.2.

Listing 4 illustrates the use of the list and default generators. Generators are combined using the `choose` combinator, which implements a left choice: Data is generated by the leftmost generator that can handle the requested type. In the example, requests for integers are handled by the static list generator, while all other types are handled by the default generator. This generator is sufficient for testing concepts like *monoid* or *ring*; though we may wish to have more data points or use random data.

A useful approach is to combine static data with randomly generated data. Using our current generator library, generators cannot be combined in this way; only a single generator is used for each type. The alternative is to call the tests multiple times, with both static and random generators. We hope to remove this restriction in the future.

Programmers may also provide their own generators, based on the same interface. The generator should have a member function template `C generator.get( selector<type>)` where `C` is a container with element type `Type&`. The reference is needed to support axioms with side effects (returning a container by copy, and not by reference should take care of this requirement).

[4] Operations wrapped as types must be instantiated as objects before they can be used. They are then passed to the axiom tests as normal test data.

4.2 Random Terms

In our experience, generating random values is easier than attempting to pick values manually. There are however some difficulties in generating data randomly:

- We do not want to manually re-implement a generator for each type, nor do we want to require that types conform to some particular interface for generating random values.
- In some cases we need values that are related to each other in some way, in order to satisfy axiom conditions and ensure proper coverage of our axiom tests.

We supply a random term generator library to simplify random data generation. The term generator generates random values of one or more types from a set of operations. When a value of a particular type is requested, the generator will select an operation of the correct type at random, and generate an argument list for it recursively. It handles some cases of supplying related values by reusing old values when generating new values.

The example generator in Listing 5 can generate integers, integer sets, and iterators for integer sets. It uses the following operations (given as C++ lambda expressions in the code): `int()`, constructing a random integer, `set<int>()` constructing an empty set, `set<int>(set<int>, int)` building a set from a set and an integer, and `set<int>::iterator(set<int>)` building an iterator from a set.

The term generator tends to produce simple terms first, with more complicated (nested) terms produced later. This means that errors are likely to be uncovered by simple test values if possible; making the error reporting nicer and less overwhelming, without the need to actively search for a simpler test case.

4.2.1 Generating Related Values

Obtaining related values is important in many common situations. For example, we may have the following transitivity axiom:

```
static void transitivity(const Rel& rel,
                         const T& a, const T& b, const T& c) {
  if (rel(a, b) && rel(b, c))
    axiom_assert(rel(a, c));
}
```

If we pick random values for `a`, `b` and `c`, and use `==` for `rel`, we may end up never generating a case where `a == b` and `b == c`, and the body of the axiom is never tested. Ideally, we would like a mix of cases covering either, both and neither of `a == b`, `b == c`.

A similar situation occurs with C++ containers. Containers use iterators, and it is often important to keep a universal quantifier on both a container and an iterator so that we can test more than just the usual `begin` and `end` iterators in our axioms. For instance, a C++ `AssociativeContainer` is a container that supports removal of elements, which reduces the size by exactly 1. Listing 6 shows an axiom for `erase`. In order to test erasure of other elements than the first and the last, we need to receive an `Iterator` in addition to the set – but the given iterator must be an iterator from the container. To test the axiom properly, we need this precondition to hold on at least some values, *e.g.*, we need to make sure the axiom is called with arguments of form

```
auto int_set_generator =
  term_generator_builder
    <std::set<int>, // list of supported types
     std::set<int>::iterator,
     int>()
  (engine, // random generator engine

   // generate random integers
   std::function<int()>([&engine] () {
     return std::uniform_int_distribution<int>()(engine);
   }),

   // generate set: initial
   constructor<std::set<int>()>(),

   // generate set: insert
   std::function<std::set<int>(std::set<int>, int)>
   ([](std::set<int> in, int i) {
       in.insert(i);
       return std::move(in);
     }),

   // generate random iterator for a given set
   std::function<std::set<int>::iterator(std::set<int>&)>
   ([&engine] (std::set<int>& s) {
      auto n = std::uniform_int_distribution<decltype(s.size())>
        (0, s.size())(engine);
      auto i = s.begin();
      for (decltype(n) j = 0; j < n; ++j, ++i) ;
      return i; // will point to random element in s
    }));
```

Listing 5 – Generating random sets: The `term_generator` is initialised with a set of supported types (`std::set<int>`, `std::set<int>::iterator` and `int`), a random number engine and set of operations for generating values of the supported types. When the test driver requests a data value, it will select and call a random operation of the appropriate type, recursively generating arguments to it.

```
static void erasure(AssociativeContainer c, Iterator i) {
  if ((i == find(c, i)) && (i != end(c)))
    axiom_assert(size(erase(c, i)) == size(c) - 1);
}
```

Listing 6 – Erasure axiom on an associative container.

`(s, s.begin()+N)` with `N<s.size()`. We cannot expect a pure random generator of iterators to generate such iterators inside sets from another generator.

Our term generator library solves this by sometimes reusing old values. With some probability (currently 50% by default), it will reuse a previously generated value, if available.

For example, the integer set generator in Listing 5 will generate related sets and iterators. Iterators are constructed from sets; since the generator will reuse old values, chances are that we will at some point get both a set and an iterator generated from that set.

5 Discussion

5.1 Specification in Programming Languages

Floyd and Hoare [Flo67, Hoa69] did important early work on specification and verification of data abstractions. This axiomatic or assertion-based approach has been used in languages like Gypsy [AGB+77], Euclid [PHL+77], CLU [LAB+81, Lis93], Alphard [SWL77, WLS76], Eiffel [Mey92b] – and more recently – Spec# [BLS05] and JML [LBR06].

Early attempts at using specifications in programming languages focused on documentation and verification. Gypsy supported verification with the help of an interactive theorem prover. In the case of Euclid, verification was left to external tools, but assertions could be tested at runtime. Alphard integrated specifications in the language, but proofs had to be done by hand. In CLU, the `requires`, `ensures` and `modifies` clauses were used in an informal way.

Eiffel [Mey92b] is a more modern design, formalising the idea of `require` and `ensures` into *design by contract* [Mey92a]. Data structures are protected by a class `invariant` (known in Euclid as a *module invariant*), which must hold on entry and exit to any exported method. Eiffel also provides statement-level assertions and loop invariants, and the compiler can automatically insert assertion checks if desired.

As Eiffel is an object-oriented language, it also deals with contracts and assertions in the context of inheritance, polymorphism and dynamic binding. A subclass must always obey the contract of the class or classes it inherits from, since objects of a subclass can be assigned to and used as variables declared as the superclass (in accordance with the substitution principle). Preconditions must therefore not be stronger than those of the inherited classes, and postconditions may not be weaker. It is allowed, however, to have weaker preconditions and stronger postconditions in the subclass – this just means the subclass does a 'better job' at fulfilling the contract.

Class invariants are always at least as strong in subclasses – we may add additional clauses to the invariant, but the invariants of inherited classes must still hold. In contrast with C++, which uses static binding of methods by default, Eiffel always uses dynamic binding (`virtual` functions in C++ terminology). This follows naturally from the use of class invariants; with static binding, the method of a superclass may be called to process an object of a subclass – but the superclass method has no knowledge of, and cannot be expected to uphold the invariant of the subclass. The design of contract inheritance ensures that methods always act according to their defined contract, even in the face of inheritance and dynamic binding, thus making it possible to reason about code that employs inheritance.

Spec# [BLS05] brings design by contract to C#, as a language extension with

convenient syntax. It comes with a theorem-proving static checking tool that can verify assertions. A similar solution for Java is the Java Modeling Language [LBR06], where specifications are written as comments in the source code, allowing programs to be compiled with a normal Java compiler. The comments are processed by a wide range of tools, including the `jmlc` compiler which inserts runtime assertion checks, verification tools like the Extended Static Checker (ESC/Java), unit testing tools and integrated editors.

5.2 Algebraic Specification

Early work by Liskov and Zilles [LZ75] discuss techniques for formal specification of abstract data types. They point out that specification should be done by relating the operations of the abstract data type, rather than directly specifying the input / output of each operation. The latter leads to over-specification, providing 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.

Among the techniques discussed by Liskov and Zilles, they point out algebraic specification [GTW78, GH78, GHM78, LZ75] as showing 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 our idea of concepts, which provide axioms together with a syntax description in the form of associated types and operations.

The Tecton system [KMS81] pioneered concept-oriented programming, providing *structure types* (similar to concepts, with axioms) and a set of operations to manipulate them; including refinement (adding axioms), generalisation (*e.g.*, by removing axioms), adding new operations and providing implementations.

Extended ML [ST86] is an important early work on mixing programming languages with algebraic specification, which introduced algebraic specification into the Standard ML language – rather than using axiomatic specifications with pre- and postconditions as in Euclid or Eiffel. Extended ML separates signatures from implementations, and allows axioms to be given together with the signatures. It builds on the idea of *institutions* [GB84], in order to achieve independence from the underlying logic system.

Zalewski and Schupp [ZS07] discuss C++ concepts more closely from a specification and institution point-of-view. The algebraic approach has the added benefit of being directly usable for rewriting [TJ07, BH09] and as a basis for testing, as seen in this paper. On the other hand, assertions are immediately useful as checks during the runtime of a program.

As in Extended ML, C++ concepts separate the specification from the implementation. The same concept may apply to several different implementations, and one should avoid putting undue constraints on the implementation (*i.e.*, avoid over-specification). Hence, we should limit axiom expressions to the operations provided in the concept (together with C++'s primitive operations – on Boolean values, for example – these can be considered implicitly defined in every concept).

Compared to the pre- and postcondition approach used by the systems in the previous section, algebraic specification brings the focus to abstract or generic inter-

faces, rather than dealing with specific procedures on particular data types. Certain relationships that are easily specified using axioms, cannot be specified using pre- and postconditions; for example the relationship between equality and the hash function (all equal values should have the same hash value), as shown in Listing A.2.

However, axioms are not a replacement for pre- and postconditions and assertions. Preconditions are still needed for partial functions, where the axioms may not define the behaviour at all (unless exception behaviour is specified, as in Section 3.5); and class/data and loop invariants and code assertions apply at the implementation level, and have no counterpart in a specification of an interface.

5.3 Axiom Selection

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 – prime candidates for the term generator in Section 4), *transformers* (which can be defined in terms of constructors) and *observers* (which yield values of another type). We can then construct axioms from the combination of each constructor with every non-constructor. Guttag [Gut80] and Antoy [Ant89] discuss further guidelines for constructing specifications.

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 less likely 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 multiple axioms covering related behaviour will increase the chance of catching the bug.

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 all programmers 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.

5.4 Axiom-Based Testing

The DAISTS [GMH81] system introduced axiom-based testing in the early eighties, using formal algebraic specifications as a basis for unit testing. In DAISTS, a test consists of axioms in the form of conditional equations; an implementation with an equality operator; and a set of test data. DAISTS performs simple coverage analysis to ensure that all the axioms and program code are exercised by the tests.

ASTOOT [DF94] 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 [HS96] 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++.

QuickCheck [CH00] is a popular axiom-based testing (property-based testing in their terminology) system for Haskell, with spin-offs for several other languages.

QuickCheck properties somewhat like free-floating axioms in CATSFOOT (see Section 3.1). QuickCheck provides a library for generating random data, including a default generator (based on the algebraic structure of types), and also includes support for observing the distribution of test data and classifying test data, so that one can check that the properties have been thoroughly exercised.

From a generic programming point of view, *type classes* in Haskell correspond to C++ concepts. However, type classes do not have axioms, and hence QuickCheck is not integrated with type classes in the way our axiom-based testing is integrated with concepts.

(Lazy) SmallCheck [RNL08] provides exhaustive property-based testing for Haskell; instead of generating random data values, *all* possible values are generated, up to some depth. We have not pursued this data generation scheme, but it should be possible to support something similar in our library, with a specialised version of the term generator.

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 [Bec98], while much formal methods research has focused on formal specification and verification – which have been difficult to apply to mainstream languages and mainstream development.

Verification attempts to prove the correctness of a program with respect to the specification, or at least prove some properties of the program (such as the absence of null pointer references or runtime exceptions). Testing is more pragmatic in nature, easier to automate and closer to ordinary programming. Current verification tools typically require highly-trained human assistance, but can give definitive answers; testing can only give confidence in correctness, not certainty.

We have used axiom-based testing for the Sophus numerical C++ library [HB05], and with the JAxT [HK08] tool for Java. JAX [SLA02] (Java Axioms) is another approach to axiom-based testing in Java. Axiom-like features have also been added to recent versions of JUnit [Saf07]. Gaudel and Le Gall [GG08] provide a survey of the use of algebraic specification in testing.

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 gives extra properties on which to base tools. For example, the approach of Antoy and Hamlet [AH00] uses initial specifications, which are evaluated alongside the implementation, as a *direct implementation* [GHM78] 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.

Axioms written in C++ concepts will normally be loose and incomplete, mak-

ing such 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). Based on our experience developing and testing Sophus [HB05, HFMK05], we find such axioms useful.

5.5 Algebraic Axioms and Imperative Code

As discussed in Section 3.4, a particular problem occurs for code written in object-oriented or imperative styles, relying on side-effects on arguments. Although this is a poor fit for algebraic-style axioms, we can make a uniform algebraic-style interface to implementation code using wrappers.

In the ASTOOT [DF94] 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 Doong and Frankl [Doo93, DF91] describe a relaxation of this. 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.

Chen *et al.* have developed the ideas of ASTOOT further, and applied them to axiom-based testing of object-oriented code at the level of class clusters and components [CTCC98, CTC01].

5.6 Equality in Axioms and Equality Testing

The original C++ concept proposal used equational axioms, leading to the question of whether equivalence in the axioms were the same equivalence as that defined by the equals operator of the type. In the end, a separate symbol was introduced, indicating substitutability. Our situation is different, as all our axioms are basically C++ Boolean expressions – of any kind, not just equations – and if equivalence is used, it will be the C++ equivalence (if defined for the given type).

There are cases where equality can be expensive or even impossible to define so that it corresponds to the behaviour expected by the axioms. There are also types on which we want to have different equivalence relations depending on the behaviour.

We may handle this by parametrising the concept by the equality operator. It is then treated like any other parameter, and the test driver will instantiate the concept with the right equality for the behaviour intended.

The equality operator is an operation like any other, and should itself be tested. The traditional equivalence relation is specified by the *equivalence* concept, shown in Listing A.1, which states that it should be reflexive, transitive and symmetric. The interaction between equality any other given operation is specified by the *congruence* concept (also in Listing A.1). The congruence axiom states that any two calls to the same operation with arguments that are pairwise equal, will yield equal results (relative to the given equivalence relation). Ideally, we expect this to hold for all operations for types with an equality operator. If we demand this behaviour, the congruence concept should be required for all operations in a concept – unfortunately we must specify this manually, as there is no way to generate a requirement for each defined operation.

If equality is missing for a type, we can define a custom equivalence relation. This may be exact, optimistic or pessimistic, depending on the properties of the axioms. For the use in equational axioms we can automatically generate such a relation over a signature (a list of operations) [CTC01, CTCC98]. The axioms can then be tested correctly. This approach randomly calls operations on the two objects, and compares observable values of other types with a defined equality – *i.e.*, it tests whether the objects behave the same (behavioural equality). An optimistic equivalence will be able to test equational equations.

We have experimented with such an infrastructure in C++, though we have not pursued the approach very far. The implementation is similar to a random term generator since it gets initialised with a C++ random number generator and a list of operations. However, it also takes an extra data generator (for example to generate indexes for an array-like structure) and returns a relation (*i.e.* a function returning a Boolean and taking two parameters of the same type). This function can then wrap this relation into a type and give it as parameter to a concept requiring the equivalence relation.

5.7 Experiences with Axiom-Based Testing

We have experience with axiom-based testing from the Sophus numerical software library [HB05]. 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 [HK08] 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 – supervised by the last author – successfully wrote JAxT axioms for parts of the Java collection classes, discovering some weaknesses in the interface specifications in the process.

The JAX [SLA02] method of combining axioms with the JUnit [BG11, Lou05] testing framework has provided some valuable insight into the usefulness of axiom-based testing. The JAX developers conducted 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 [GMH81] were positive and indicated that it helped users to develop effective tests, avoid weak tests, and the use of insufficient test data. With Daistish [HS96], 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.

Gaudel and Le Gall [GG08] summarise further experiences and case studies on testing with algebraic specifications.

5.8 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 in C++ and its impact on development.
- Testing of multi-threaded applications is notoriously difficult [Sen07], and it would be interesting to see if axiom-based testing could be applied here.
- Refining or layering concepts quickly becomes cumbersome as the parameter lists grow. This problem is probably difficult to solve within the constraints of C++ 2011.

There is also 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 [Got06]. Such concepts should eventually make their way into the C++ standard, for consistency and interoperability.
- For truly effective testing, we will likely need more kinds of data generators, and a set of combinators for the generators.
- Our term generator is fairly simplistic in terms of control over distribution of the generated data. This is something we need to explore more.
- As we no longer rely on an external tool with limited C++ support, we can now look into testing existing software like the STL. The GCC version of STL (at least) already uses a simple form of concepts internally, which could be used as a starting point.

6 Conclusion

Axiom-based testing is a technique for testing generic code in a generic way, based on algebraic specification. In this paper, we have discussed how we can integrate algebraic specification in C++ 2011 in the form of concepts using template meta-programming; and how these concepts can be used in automated testing with different kinds of data generators.

Compared to traditional unit testing, axiom-based testing offers a less *ad hoc* way to specify and test behaviour, and we believe it is particularly well suited to test generic code. Compared to pre- and postconditions, axioms are again more suited for generic code, and allows for specification of relationships that cannot be covered by pre- and postconditions – though axioms are not a replacement for assertions and some forms of pre- and postconditions at the implementation level.

A Code Examples and Listings

A.1 Equivalence and Congruence

Specification of equivalence and congruence relations. The equivalence operator should be reflexive, symmetric and transitive.

```
template <typename T, typename Rel>
struct equivalence: public concept {
  typedef concept_list<
    is_callable<Rel(T, T)>,
    std::is_convertible<typename is_callable<Rel(T, T)>::result_type,
                        bool>
    > requirements;
  static void reflexivity(const Rel& rel, const T& a) {
    axiom_assert(rel(a,a));
  }
  static void symmetry(const Rel& rel, const T& a, const T& b) {
    if (rel(a, b))
      axiom_assert(rel(b, a));
  }
  static void transitivity(const Rel& rel,
                           const T& a, const T& b, const T& c) {
    if (rel(a, b) && rel(b, c))
      axiom_assert(rel(a, c));
  }
  AXIOMS(reflexivity, symmetry, transitivity);
};
```

Congruence is defined for a relation `Rel` over an operation `Op(Args...)`. Calls to `Op` with equivalent arguments should yield equivalent results. (This form of axiom does not work currently with GCC but should according to the standard. We have however a version working with GCC using tuples as workaround.)

```
template <typename Rel, typename Op, typename... Args>
struct congruence: public concept {
  typedef concept_list<
    equivalence<Args, Rel>...,
    is_callable<Op(Args...)>,
    equivalence<typename is_callable<Op(Args...)>::result_type,
                Rel>
    > requirements;
  static void congruence_axiom(const Args&... args1, const Args&... args2,
                               const Op& op, const Rel& rel) {
    if (rel(args1, args2)...)
      axiom_assert(rel(op(args1...), op(args2...)));
  }
  AXIOMS(congruence_axiom);
};
```

A.2 Hashing

```
template <typename T, typename Hash>
struct hash: public concept {
  typedef concept_list<
    congruence<op_eq, Hash, T>
    > requirements;
```

```
};
```

A.3 Testing Exceptions

This can test whether a function has thrown an exception or not.

```
template <typename T,
          typename Fun, typename... Args,
          ENABLE_IF(is_callable<Fun(Args...)>)>
bool throwing(Fun&& fun, Args&&... args) {
  try {
    std::forward<Fun>(fun)(std::forward<Args>(args)...);
  } catch (T) {
    return true;
  }
  return false;
}
```

A.4 AXIOMS Macro

The output of `AXIOMS(associativity, identity)`:

```
static auto get_axioms()
  -> decltype(::catsfoot::details::zip_vec_tuple
          (::catsfoot::details::split_identifiers("associativity,␣identity"),
           std::make_tuple(associativity, identity)))
{ return ::catsfoot::details::zip_vec_tuple
       (::catsfoot::details::split_identifiers("associativity,␣identity"),
        std::make_tuple(associativity, identity)); };
```

The function `get_axioms` returns a list of pairs of names (as strings) and the corresponding axioms.

A.5 The Printable Concept and Overloading

The concept *printable* is an auto concept. Any pair of $\langle T, U \rangle$ for which a left shift operator (`<<`) is defined will model this concept (no *verified* declaration necessary).

```
template <typename T, typename U>
  struct printable: public auto_concept {
  typedef concept_list<is_callable<op_lsh(T, U)>> requirements;
  };
```

This concept can then be used in concept-based overloading to select between functions based on properties of the argument types:

```
template<typename Stream, typename T,
         ENABLE_IF(printable<Stream&, T>)>
void print_if_printable(Stream& s, T t) {
  s << t;
}
template<typename Stream, typename T,
         ENABLE_IF_NOT(printable<Stream&, T>), typename = void>
```

```
12 void print_if_printable(Stream& s, T) {
13   s << "Values␣of␣type␣" << type_to_string<T>() << "␣are␣not␣printable";
14 }
```

The first function is used if a `<<` operator is available to print the value on the stream; otherwise the other function is used, which prints a representation of the value's type.

B Quick definitions

We define here some parts of the API that we have used in the examples.

identifier	type	description
`is_callable<T, Args...>`	predicate	whether `T` is callable with parameters `Args...`
`disamb<Args...>()(&f)`	function	Returns the address of overloaded `f` that satisfies call with `Args...` (eventually with implicit conversions)
`op_plus`	type	Is default constructible. Is callable with `T, U` only when `operator+` is.
`axiom_assert`	macro	See Section 3
`AXIOMS(A...)`	macro	Implements `get_axioms`
`get_axioms`	function	Returns list of axioms of a concept.
`DEF_WRAPPER(F...)`	macro	Wrap function
`DEF_WRAPPER_AS_FUNC(F...)`	macro	Wrap function, functionalised
`DEF_MEMBER_WRAPPER(F...)`	macro	Wrap member
`DEF_MEMBER_WRAPPER_AS_FUNC(F...)`		Wrap method, functionalised
`ENABLE_IF(M)`	macro	Used in a template parameter list to disqualify a function from overloading unless `M` holds
`ENABLE_IF_NOT(M)`	macro	Used in a template parameter list to disqualify a function from overloading if `M` holds
`default_generator`		See Section 4.
`list_data_generator`		See Section 4.
`term_generator`		See Section 4.
`tuple_generator`		See Section 4.
`choose`		See Section 4.
`constant<T,V>`	type	Constant as operation.
`concept_list<C...>`	type	List of requirements (Section 2)
`concept`	class	All concepts should inherit from concepts, to distinguish them from predicates.

B.1 New in C++ 2011

identifier	type	description
`decltype(E)`		Gives the type of `E`
`typename... T`		Variadic template parameter
`T...`		Variadic template argument
`std::is_convertible<T, U>`		`T` is convertible to `U`
`[...] (T a, U b,...) {...}`		Lambda expression. Closure defined in `[...]` (selected variables `[x,&y]`, or any variable by reference `[&]` or value `[=]`)

References

[AGB+77] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. Gypsy: A language for specification and implementation of verifiable programs. In *Proceedings of an ACM conference on Language design for reliable software*, pages 1–10, New York, NY, USA, 1977. ACM. doi:10.1145/800022.808306.

[AH00] Sergio Antoy and Richard G. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Trans. Software Eng.*, 26(1):55–69, 2000. doi:10.1109/32.825766.

[Ant89] Sergio Antoy. Systematic design of algebraic specifications. In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design*, pages 278–280, New York, NY, USA, 1989. ACM. doi:10.1145/75199.75241.

[B+11a] Pete Becker et al. ISO/IEC 14882:2011: Programming languages – C++ (final draft international standard). Technical Report N3290, JTC1/SC22/WG21 – The C++ Standards Committee, April 2011. Available from: http://www.open-std.org/jtc1/sc22/wg21/.

[B+11b] Pete Becker et al. Working draft, standard for programming language C++. Technical Report N3242=11-0012, JTC1/SC22/WG21 – The C++ Standards Committee, February 2011. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[BDH09] Anya Helene Bagge, Valentin David, and Magne Haveraaen. The axioms strike back: Testing with concepts and axioms in C++. In *GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering*, pages 15–24, New York, NY, USA, 2009. ACM. doi:10.1145/1621607.1621612.

[Bec98] Kent Beck. Extreme programming: A humanistic discipline of software development. In *Fundamental Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, pages 1–6. Springer-Verlag, 1998. doi:10.1007/BFb0053579.

[Bec02] Kent Beck. *Test-Driven Development: By Example.* Addison-Wesley, 2002. ISBN:978-0-321-14653-3.

[BG11] Kent Beck and Erich Gamma. JUnit – Java Unit testing. 2011. Available from: http://www.junit.org/.

[BH09] Anya Helene Bagge and Magne Haveraaen. Axiom-based transformations: Optimisation and testing. In Jurgen J. Vinju and Adrian Johnstone, editors, *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238 of *Electronic Notes in Theoretical Computer Science*, pages 17–33, Budapest, Hungary, 2009. Elsevier. doi:10.1016/j.entcs.2009.09.038.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Proceedings of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture*

Notes in Computer Science, pages 49–69. Springer-Verlag, 2005. doi:10.1007/978-3-540-30569-9_3.

[CH00] Koen Claessen and John 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, New York, NY, USA, 2000. ACM Press. doi:10.1145/351240.351266.

[CTC01] Huo Yan 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.*, 10:56–109, January 2001. doi:10.1145/366378.366380.

[CTCC98] Huo Yan 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.*, 7:250–295, July 1998. doi:10.1145/287000.287004.

[Dav11] Valentin David. CATSFOOT, 2011. http://catsfoot.sourceforge.net/.

[DF91] Roong-Ko Doong and Phyllis G. Frankl. Case studies on testing object-oriented programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 165–177, New York, NY, USA, 1991. ACM Press. doi:10.1145/120807.120822.

[DF94] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994. doi:10.1145/192218.192221.

[Doo93] Roong-Ko Doong. *An approach to testing object-oriented programs.* PhD thesis, Polytechnic University, Brooklyn, NY, USA, 1993.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[GB84] Joseph A. Goguen and Rod M. Burstall. Introducing institutions. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256, London, UK, 1984. Springer-Verlag. doi:10.1007/3-540-12896-4_366.

[GG08] Marie-Claude Gaudel and Pascale Le 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-Verlag, 2008. doi:10.1007/978-3-540-78917-8.

[GH78] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10:27–52, 1978. doi:10.1007/BF00260922.

[GHM78] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Commun. ACM*, 21(12):1048–1064, 1978. doi:10.1145/359657.359666.

[GMH81] John D. Gannon, Paul R. McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981. doi:10.1145/357139.357140.

[GMWL08] Douglas Gregor, Mat Marcus, Thoams Witt, and Andrew Lumsdaine. Foundational concepts for the C++0x standard library (revision 5). Technical Report N2774=08-0284, JTC1/SC22/WG21 – The C++ Standards Committee, September 2008. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2774.pdf.

[Got06] Peter Gottschling. Fundamental algebraic concepts in concept-enabled C++. Technical Report TR638, Department of Computer Science, Indiana University, 2006. Available from: https://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR638.

[GSSW08] Douglas Gregor, Bjarne Stroustrup, Jeremy Siek, and James Widman. Proposed wording for concepts (revision 9). Technical Report N2773=08-0283, JTC1/SC22/WG21 – The C++ Standards Committee, September 2008. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2773.pdf.

[GTW78] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978. ISBN:978-0-13-195735-0.

[Gut80] John V. Guttag. Notes on type abstraction (version 2). *IEEE Trans. Softw. Eng.*, 6(1):13–23, 1980. doi:10.1109/TSE.1980.230209.

[Gut99] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999. doi:10.1109/32.815325.

[Ham94] Richard Hamlet. Random testing. In J Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994. doi:10.1002/0471028959.sof268.

[HB05] Magne Haveraaen and Enida Brkic. Structured testing in Sophus. In Eivind Coward, editor, *Norsk informatikkonferanse NIK'2005*, pages 43–54. Tapir akademisk forlag, Trondheim, Norway, 2005. Available from: http://www.nik.no/2005/.

[HFMK05] Magne Haveraaen, Helmer André Friis, and Hans Munthe-Kaas. Computable scalar fields: a basis for PDE software. *Journal of Logic and Algebraic Programming*, 65(1):36–49, September-October 2005. doi:10.1016/j.jlap.2004.12.001.

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996. doi:10.1145/227699.227700.

[HK08] Magne Haveraaen and Karl Trygve 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, New York, NY, USA, 2008. ACM. doi:10.1145/1449814.1449834.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

[HS96] Merlin Hughes and David 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, New York, NY, USA, 1996. ACM Press. doi:10.1145/229000.226301.

[HT90] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, 1990. doi:10.1109/32.62448.

[JWL03] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 228–244, New York, NY, USA, 2003. Springer-Verlag. doi:10.1007/978-3-540-39815-8_14.

[KMS81] Deepak Kapur, David R. Musser, and Alexander A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Program Specification, Proceedings of a Workshop*, Lecture Notes in Computer Science, pages 402–414, Aarhus, Denmark, August 1981. Springer-Verlag. doi:10.1007/3-540-11490-4_24.

[LAB+81] Barbara Liskov, Russell R. Atkinson, Toby Bloom, J. Eliot B. Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981. doi:10.1007/BFb0035014.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006. doi:10.1145/1127878.1127884.

[Lis93] Barbara Liskov. A history of CLU. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147, New York, NY, USA, 1993. ACM Press. doi:10.1145/154766.155367.

[Lou05] Panagiotis Louridas. JUnit: Unit testing and coding in tandem. *IEEE Softw.*, 22(4):12–15, 2005. doi:10.1109/MS.2005.100.

[LZ75] Barbara Liskov and Stephen Zilles. Specification techniques for data abstractions. In *Proceedings of the international conference on Reliable software*, pages 72–87, New York, NY, USA, 1975. ACM. doi:10.1145/800027.808426.

[Mey92a] Bertrand Meyer. Applying "Design by contract". *Computer*, 25(10):40–51, 1992. doi:10.1109/2.161279.

[Mey92b] Bertrand Meyer. *Eiffel: The language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN:978-0-13-247925-7.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., 1st edition, February 1979. ISBN:978-0-471-04328-7.

[PHL+77] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of Euclid. In *Proceedings of an ACM*

conference on Language design for reliable software, pages 11–18, 1977. doi:10.1145/800022.808307.

[PVB08] Wishnu Prasetya, Tanya Vos, and Arthur Baars. Trace-based reflexive testing of OO programs with T2. In *International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 151–160, Los Alamitos, CA, USA, 2008. IEEE Computer Society. doi:10.1109/ICST.2008.12.

[RNL08] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 37–48, New York, NY, USA, 2008. ACM. doi:10.1145/1411286.1411292.

[Saf07] David 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, New York, NY, USA, 2007. ACM. doi:10.1145/1297846.1297919.

[Sen07] Koushik 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, New York, NY, USA, 2007. ACM. doi:10.1145/1321631.1321679.

[SL00] Jeremy G. Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[SLA02] P. David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. In Don Wells and Laurie A. Williams, editors, *Proceedings of XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002*, volume 2418 of *Lecture Notes in Computer Science*, pages 131–143. Springer-Verlag, 2002. doi:10.1007/3-540-45672-4_13.

[ST86] Donald Sannella and Andrzej Tarlecki. Extended ML: An institution-independent framework for formal program development. In *Proceedings of the Tutorial and Workshop on Category Theory and Computer Programming*, pages 364–389, London, UK, 1986. Springer-Verlag. doi:10.1007/3-540-17162-2_133.

[Str09] Bjarne Stroustrup. The C++0x "remove concepts" decision. *C/C++ Users Journal*, July 22 2009. Available from: http://www.drdobbs.com/cpp/218600111.

[SWL77] Mary Shaw, William A. Wulf, and Ralph L. London. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM*, 20(8):553–564, 1977. doi:10.1145/359763.359782.

[TJ07] Xiaolong Tang and Jaakko Järvi. Concept-based optimization. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 97–108, New York, NY, USA, 2007. ACM. doi:10.1145/1512762.1512772.

[WLS76] Wm. A. Wulf, Ralph L. London, and Mary Shaw. An introduction to the construction and verification of Alphard programs. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 390, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. doi:10.1109/TSE.1976.233830.

[ZS07] Marcin Zalewski and Sibylle Schupp. C++ concepts as institutions: a specification view on concepts. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 76–87, New York, NY, USA, 2007. ACM. doi:10.1145/1512762.1512770.

About the authors

Anya Helene Bagge is a post-doctor at the Bergen Language Design Laboratory, Department of Informatics, University of Bergen, Norway, and is currently a visiting researcher at Centrum Wiskunde & Informatica (CWI) in Amsterdam, The Netherlands. Her research interest is in tools for programming language support and the design of programming languages. http://www.ii.uib.no/~anya/

Valentin David is a researcher at Bergen Language Design Laboratory, University of Bergen, Norway. His research interest is in tools for programming language support and advanced C++ programming, and he is the main developer of the Catsfoot concept and testing library. http://www.ii.uib.no/~valentin/

Magne Haveraaen is a professor at the Department of Informatics and head of Bergen Language Design Laboratory, University of Bergen, Norway. His research interest is in the theory and pragmatics programming language design, high level programming for dependency and high performance computing and numerical software. Prof. Haveraaen was visiting Texas A&M University during the work on this paper. http://www.ii.uib.no/~magne/

Acknowledgments Andrew Sutton also has tools for dealing with concepts in C++2011, though he is emphasising the syntactic part over the axiomatic part. We discussed and exchanged ideas with him. Some of the ideas of our implementation of concepts are inspired by his work or our discussion.

Thanks to the reviewers and to Paul Griffioen for useful comments on drafts of this paper. Thanks to the GPCE'09 reviewers and attendees for valuable feedback on the previous version of this paper [BDH09].

This research is partially financed by the Research Council of Norway, under the DMPL (Design of a Mouldable Programming Language) project.