Axiom-Based Testing for C++

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

Abstract

Axioms, known from program specification, allow program functionality to be described as rules or equations. The draft C++0x standard introduces axioms as part of the new *concept* feature. We will demonstrate a tool that uses these features for automated unit testing.

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

General Terms Reliability, Languages, Design

Keywords Mouldable Programming, Generative Programming, Program Transformation, Unit Testing, Test Generation, Axioms, Specifications, Concepts, C++, C++0x

1. Introduction

The draft C++0x standard introduces *concepts* [6], a new language feature for constraining template parameters. For example, one may now specify that a generic sorting function, for example, requires an array of less-than comparable elements as argument. Previously such requirements would be checked at template instantiation time, giving hard-to-understand error messages.

A concept consists of one or more types, constraints on those types, operations on those types and axioms for the operations. For example, consider the concept shown in Figure 1. It lists two operations, requires that the array and element types support the equality operation, and that the array and index type model the *SameShape* concept. Axioms are simple conditional equations over the operations defined in the concept. The ArrayEqual axiom merely states that two indexables are equal if and only if all the elements are equal. The compiler is free to ignore the axioms, or it can make use of them for purpose, like optimization or testing [3, 10].

```
concept Indexable<typename A, typename I,
                  typename E> {
  requires std::EqualityComparable<A,A>,
           std::EqualityComparable<E,E>
           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];
    if (a[i] != b[i])
      a != b;
  }
}
```

Figure 1. An example concept *Indexable* with indexing operators and an axiom *ArrayEqual*.

2. Axiom-Based Testing

We have been exploring the use of axioms for testing, and have built a prototype tool to support this for C++, similar to JUnit theories for Java [9]. The tool uses the axioms of a concept as test oracles for testing classes that model the concept. For each axiom, we generate a template test oracle, which evaluates the condition and checks that the equation is true (see Figure 2, left). For each concept map (i.e., for each case of classes modeling a concept), we generate test code that calls the oracles with generated test data (Figure 2, right). At the top level we have a main test routine that calls the test code for the classes we want to test.

Test data generation is handled through iterators. Each class is expected to have a data generator, which is an iterator supplying instances of the class. This may be a predefined sequence of values (for example, -1, 0, 1, and 42 as integer test data), but typically it will be some combination of randomly generated test data and hand-picked data values. Testing with random (or combination) data seems the most promising approach [7], and we are basing our ideas on those explored by the QuickCheck system for Haskell [4].

```
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;
    if (a[i] != b[i])
      if (!(a != b))
        return false;
    return true;
  }
};
```

```
template <int size,typename E>
struct Indexable_testCase<ArrayFI<size, E>, FiniteInt<size>, E> {
 static void ArrayEqual() {
  for (typename ::TestUtils::Traits<ArrayFI<size, E> >::data_iterator a_0
           = ::TestUtils::Traits<ArrayFI<size, E> >::data_begin();
       a_0 != ::TestUtils::Traits<ArrayFI<size, E> >::data_end(); ++a_0)
   for (typename ::TestUtils::Traits<ArrayFI<size, E> >::data_iterator b_0
            = ::TestUtils::Traits<ArrayFI<size, E> >::data_begin();
        b_0 != ::TestUtils::Traits<ArrayFI<size, E> >::data_end(); ++b_0)
    for (typename ::TestUtils::Traits<FiniteInt<size> >::data_iterator c_0
             = ::TestUtils::Traits<FiniteInt<size> >::data_begin();
         c_0 != ::TestUtils::Traits<FiniteInt<size> >::data_end(); ++c_0)
     ::TestUtils::check(Indexable_oracle<ArrayFI<size, E>,
                          FiniteInt<size>, E>::ArrayEqual(*a_0, *b_0, *c_0),
                          "Indexable", "ArrayEqual");
}};
```

Figure 2. Oracle code from the ArrayEqual axiom, and concrete test code generated from a concept map. *Traits* are used to select an appropriate iterator for each data type. `::TestUtils::check` is a hook for reporting results to a testing framework.

3. Testing in Practice

Concept based testing involves a sequence of activities. First the concepts need to be developed. This will be an unfamiliar activity for most system developers, but we have already shown [8] that doing this may have a profound, positive effect on software development. Then concept maps need to be written for all classes that model the concepts. Using our tool then incurs the following.

1. Run the test tool on the program code containing the concepts and concept maps. The tool will generate a file containing test oracle and test code, and code stubs for calling the test code.
2. Ensure that types that should be tested have appropriate data generators.
3. Compile and run the tests.
4. In case of failure, check the implementation code, the axioms and the comparison operators (equalities) used in the axioms – bugs may be in any of these places.

The latter activity is not as dramatic as it may seem: developing axioms has the same difficulty as writing code, so axioms also need to be tested and validated, e.g., against code that is known to be (fairly) correct.

4. Conclusion

Our tool is based on the Transformers C++ transformation framework [1, 5] and is mainly intended as a proof-of-concept, intended to gain early experience with concepts and axioms [2] at a stage where the standards draft is still being refined. We hope this demonstration will generate some interest in C++ concepts and in axiom-based testing in general.

References

[1] Robert Anisko, Valentin David, and Clément Vasseur. Transformers: A C++ program transformation framework. Technical Report 0310, EPITA/LRDE, 2003.

[2] Anya Helene Bagge, Valentin David, and Magne Haveraaen. Testing with concepts and axioms in C++. In *OOPSLA '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, New York, NY, USA, 2008. ACM.

[3] Anya Helene Bagge and Magne Haveraaen. Axiom-based transformations: Optimisation and testing. In Jurgen Vinju and Adrian Johnstone, editors, *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, Electronic Notes in Theoretical Computer Science, Budapest, Hungary, 2008. Elsevier.

[4] 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.

[5] Valentin David. Preparing for C++0x. In *OOPSLA '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, New York, NY, USA, 2008. ACM.

[6] Douglas Gregor, Bjarne Stroustrup, Jeremy Siek, and James Widman. Proposed wording for concepts (revision 4). Technical Report N2501=08-0011, JTC1/SC22/WG21 – The C++ Standards Committee, February 2008.

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

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

[9] 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.

[10] Xiaolong Tang and Jaakko Järvi. Concept-based optimization. In *Proceedings of the ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07)*, New York, NY, USA, 2007. ACM.