Testing with Concepts and Axioms in C++

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

Abstract

Unit testing is a popular way of increasing software reliability. *Axioms*, known from program specification, allow functionality to be described as rules or equations. We show a method and prototype tool for using the proposed `concept` and `axiom` features of the upcoming C++0x standard 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

Concepts, in the upcoming C++0x standard [5], allow the programmer to specify requirements for template parameters. Generic sorting, for example, might require an array of less-than comparable elements as argument.

A concept is an interface specification and contains a list of abstract types, constraints on those types, operations on those types and axioms for the operations. For example, we might have stack concept, with stack and element types, and push, pop, top and new operations. A concept map declares that one or more types model a particular concept – for example that `Stack<T>` models the stack concept with element type `T`. An example concept with an axiom is shown in Figure 1.

Axioms are simple conditional equations over the operations defined in the concept. The compiler is not obligated to act on the axioms in any way, but it is free to assume that they hold and use them for code transformations. With rewriting support in the compiler or an optimization tool, this opens possibilities for domain-specific optimizations [1, 9].

```
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. The concept *Indexable* has indexing operators and an axiom *ArrayEqual* that states that two Indexables are equal if and only if their elements are equal. `A` is an indexable type, `I` is the index type, and `E` is the element type. `A` and `I` are required to be of the same shape, i.e., the values of type `I` are the allowable indices for the type `A`.

Here, we focus on another use of axioms – automated axiom-based testing. The basic idea is to use 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. 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. At the top level we have a main test routine that calls the test code for the classes we want to test.

2. Axiom-Based Testing

The idea of axiom-based testing was introduced in the early eighties in the DAISTS [4] system which supported testing based on formal algebraic specifications. Experiences with DAISTS were positive, uncovering errors not found using traditional unit testing. More recent experiments with

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

JAX [8], an axiom-based testing system for Java, show similar results.

Axiom-based testing has three requirements, in addition to the implementation being tested:

- axioms, in the form of conditional equations
- an equality operation for evaluating the axioms
- a set of data points to exercise the implementation.

In the C++ proposal, axioms are part of concepts, and are thus separate from concrete class implementations. The same axioms – and axiom-based tests – can be used for all classes that model a given concept, allowing reuse of existing, well-thought out axioms. For example, libraries of concepts and axioms may be developed for standard algebraic classes (monoid, ring, field) or data structures (trees, sequential containers, stacks).

Evaluation of tests assumes an implementation of equality (or other relevant comparison) for the types involved. In the draft standard, it is legal to write axiom equalities where there is no equality implementation (the semantics of this is simply that it is legal to replace one side of the equality with the other) – in this case the axioms won't be directly usable for testing. Test results are of course only as reliable as the comparisons used – in practice, however, a varied selection of axioms using both equalities and inequalities is likely to uncover bugs hiding in the equality operator.

Generating test data is a difficult issue for any testing scheme. Our generated test code works by iterating over a sequence of test data points, using data generators that are part of the implementation classes. Each class is expected to have a data generator, which is an iterator supplying instances of the class. In the simplest case, this may simply 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. We have built a small library for simplifying data generation and combining different data sequences, inspired by the QuickCheck system for Haskell [2].

In order to gain more experience with axiom-based testing, we have built prototype tool which automatically generates test oracles and test code based on concepts and axioms. It contains a grammar for C++ with concepts [3], and works by reading an input program and generating code whenever it encounters a concept declaration or a concept map.

3. Conclusion

We have investigated the use of concepts and axioms in the draft C++0x proposal for generating automated unit tests. Our method is based on generating test oracles from axioms, and then generating test code whenever a class is declared as modeling some concept.

We have built a proof-of-concept tool to test our method. The standard proposal is still undergoing changes, and compiler support for concepts is still immature. Getting a larger body of concept-enabled code would be an important step in developing these ideas further. Integration with a testing framework like CppUnit would also make the tool more suitable for general use.

Almost thirty years after DAISTS, axiom-based testing may finally start to catch on. In addition to the work described here, there is also work being done for Java, such as JAX [8], JAxT [6], and *theories* [7] which are available in recent versions of JUnit. The abstraction and separation of concerns available with the concept feature makes axiom-based testing in C++0x particularly appealing.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] Magne Haveraaen and Karl Trygve Kalleberg. JAxT and JDI: The simplicity of JUnit applied to axioms and data invariants. In *OOPSLA '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, New York, NY, USA, 2008. ACM.

[7] 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.

[8] 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, *XP/Agile Universe*, volume 2418 of *Lecture Notes in Computer Science*, pages 131–143. Springer, 2002.

[9] 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.