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].

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
JAX [8], an axiom-based testing system for Java, shows 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 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 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 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 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 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 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.