Advanced Coverage Metrics
for Object-Oriented Software

Executive Summary

The use of structural coverage metrics to measure the thoroughness of a test set is a well-understood technique. However, the application of the technique to object-oriented software presents new challenges. This paper presents object-oriented context coverage, a new way to measure coverage for object-oriented systems.

IPL is an independent software house founded in 1979 and based in Bath. IPL has been accredited to ISO9001 since 1988, and TickIT accredited since 1991. IPL has developed and supplies the AdaTEST, Cantata and Cantata++ software verification products. AdaTEST, Cantata and Cantata++ have been produced to these standards.

IPL
Eveleigh House
Grove Street
Bath
BA1 5LR
UK
Phone: +44 (0) 1225 444888
Fax: +44 (0) 1225 444400
email ipl@iplbath.com

Certificate Number FM1589

Last Update:28/10/99 10:28
File: Advanced Coverage Metrics.doc

1. Introduction

The use of structural coverage metrics to measure the thoroughness of a test set is a well-understood technique. However, the application of the technique to object-oriented software presents new challenges.

Traditional structural coverage metrics such as statement coverage, branch coverage and condition coverage measure how well the bodies of each method have been tested. Unfortunately these traditional metrics do not take into account the object-oriented features of the software under test. In particular, the use of polymorphism and the encapsulation of state-dependent behaviour behind well-defined class interfaces are effectively ignored. Since polymorphism and encapsulation of behaviour are major features of any object-oriented design, metrics which ignore them are insufficient for determining whether the software under test has been thoroughly tested.

2. A New Way to Measure Coverage

To address this Cantata++ introduces an extension to traditional structural coverage – ***context coverage***. Context coverage is a way of gathering more data on how the software under test executes.

The context coverage approach can be applied to the testing of both polymorphic and state-dependent behaviour. The approach can also be extended to aid in the testing of multi-threaded applications.

By using these additional ***object-oriented context coverage*** metrics, in combination with traditional coverage metrics, we can ensure that the structure of the code has been fully exercised and thus have high confidence in the quality of our test set.

Three varieties of OO context coverage metrics are defined. ***Inheritance context coverage*** metrics are designed to help measure how well the polymorphic calls in the system have been tested. ***State-based context coverage*** metrics are designed to improve the testing of classes with state-dependent behaviour. ***User-defined context coverage*** metrics are also supported, allowing the application of the context coverage approach to other cases where traditional structural coverage metrics are inadequate, such as multi-threaded applications.

3. How Should We Test Polymorphism?

Traditional structural coverage metrics are inadequate as a measure of how well polymorphic[1] code has been tested. Consider the following fragment of code:

```
class Base {
public:
      void foo()                    { ... helper(); ... }
      void bar()                    { ... helper(); ... }
```

[1] Polymorphism is the ability to perform operations on an object without knowing exactly how the object will implement the operation. In C++, polymorphism can be achieved at run-time through the combination of several language features: (public) class inheritance, overriding virtual methods and using base class pointers or references to refer to derived class objects. Polymorphism can also be achieved at compile-time using templates.

```
private:
      virtual void helper()         { ... }
};
class Derived : public Base {
private:
      virtual void helper()         { ... }
};
void test_driver() {
      Base    base;
      Derived derived;
      base.foo();                  // Test Case A1
      derived.bar();               // Test Case A2
}
```

In this example, test case A1 invokes `Base::foo()` on the base object which in turn calls virtual function `helper()` on the base object, invoking `Base::helper()`.

In test case A2, note that `bar()` is not overridden in `Derived`, so the inherited method `Base::bar()` is invoked on the derived object, which in turn calls `helper()` on the derived object, invoking `Derived::helper()`.

Does the `test_driver()` function fully exercise the `Base` class? Does it fully exercise the `Derived` class?

Let's assume for now that the functions each contain linear code only – there are no decisions or loops at all.

Using a traditional structural coverage metric (such as statement coverage) as our guide we would answer yes to both questions since running `test_driver()` would achieve 100% coverage of `Base::foo()`, `Base::bar()`, `Base::helper()` and `Derived::helper()`.

3.1. Compile-Time Polymorphism With Templates

The C++ template feature provides a form of compile-time polymorphism. As with inheritance-based polymorphism, a routine from one class can be invoked on objects of different types. The template class corresponds to the base class where the routine is defined. The instantiations correspond to derived classes which inherit the routine.

The template routine can in turn call "helper" routines, where the particular helper routine invoked will depend on the type of the object. For example, a list template might invoke the contained object's copy constructor. The copy constructor invoked will be the one corresponding to the type of the contained object, which will be specific to a particular instantiation of the list template.

For the remainder of this paper, only inheritance-based (run-time) polymorphism will be discussed. However, the principles and techniques involved apply equally to template-based (compile-time) polymorphism.

3.2. What Did We Miss?

We have not fully tested the interactions between `Base` and `Derived`. Specifically, we have not tested the interactions between `Base::bar()` and `Base::helper()` or the interactions between `Base::foo()` and `Derived::helper()`.

Obviously, just because `Base::foo()` works with `Base::helper()` does not mean that it will automatically work when used with `Derived::helper()`. Although `Derived::helper()` has the same interface as `Base::helper()`, they have different implementations.

For our testing to be *really* thorough we should exercise both `foo()` and `bar()` for both the base class and the derived class – as we would if they were explicitly overridden in the derived class.

Figure 1 shows that the inherited methods are not fully covered by the original test cases. The diagram includes the “make-believe” versions of `foo()` and `bar()`(shown enclosed in {braces}) which represent their inheritance in `Derived`.

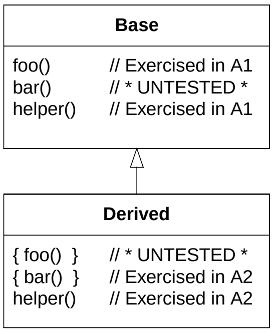
Base
foo() // Exercised in A1
bar() // * UNTESTED *
helper() // Exercised in A1
Derived
{ foo() } // * UNTESTED *
{ bar() } // Exercised in A2
helper() // Exercised in A2

Figure 1: The inherited methods have not been fully exercised

3.3. We Need Better Tests

To achieve 100% coverage an enhanced test set is required:

```
void better_test_driver() {
      Base    base;
      Derived derived;
      base.foo();                     // Test Case B1
      base.bar();                     // Test Case B2
      derived.foo();                  // Test Case B3
      derived.bar();                  // Test Case B4
}
```

The additional test cases help ensure that the interactions between the public methods and the private helper functions have been fully exercised, as shown in figure 2.

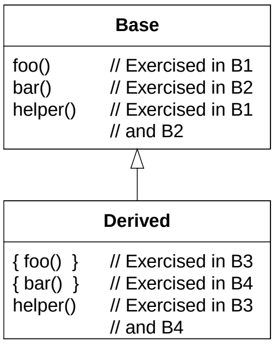
Base
foo() // Exercised in B1
bar() // Exercised in B2
helper() // Exercised in B1
// and B2
Derived
{ foo() } // Exercised in B3
{ bar() } // Exercised in B4
helper() // Exercised in B3
// and B4

Figure 2: Better test cases give 100% inheritance context coverage

3.4. Traditional Coverage Is Not Enough

It is clear that when methods are inherited by a derived class some further testing is necessary. This is true both for those methods which have been inherited unchanged and for those which have been overridden. Coverage measurement needs to be more context-dependent, recording the class of the specific object on which the method was executed (i.e. the context in which the coverage was achieved). Coverage achieved in the context of one derived class should not be taken as evidence that the method has been fully tested in the context of another derived class.

Unfortunately, traditional structural coverage metrics do just that. They treat any execution of a routine – whether it is in the context of the base class or a derived class – as equivalent. Thus `Base::bar()` is wrongly considered “100% covered” when it has been exercised in the context of class `Derived` only.

As figure 3 shows, OO code may be wrongly considered 100% covered by traditional metrics when in fact it has been 50% covered in the context of one derived class and 50% covered in the context of another derived class.

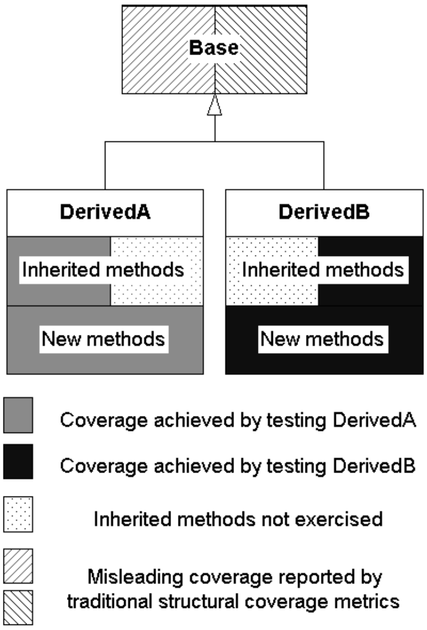
Base
DerivedA
DerivedB
Inherited methods
Inherited methods
New methods
New methods
Coverage achieved by testing DerivedA
Coverage achieved by testing DerivedB
Inherited methods not exercised
Misleading coverage reported by traditional structural coverage metrics

Figure 3: Only 50% coverage of inherited methods in each derived class yet Base appears fully tested using traditional metrics

This problem applies to *all* the traditional structural coverage metrics – none of them take into account the interactions between inherited base class methods and the overridden methods in each derived class.

3.5. What Is Inheritance Context Coverage?

Inheritance context coverage is not a single metric, but rather a way of extending the interpretation of (any of) the traditional structural coverage metrics to take into account the additional interactions which occur when methods are inherited.

Inheritance context coverage provides alternative metric definitions which consider the levels of coverage achieved in the context of each class as separate measurements. The inheritance context definitions regard execution of the routine in the context of the base class as separate from execution of the routine in the context of a derived class. Similarly, they regard execution of the routine in the context of one derived class as separate from execution in the context of another derived class.

To achieve 100% inheritance context coverage, the code must be fully exercised in each appropriate context.

3.6. Definition

To take a specific example, the inheritance context coverage variant of decision coverage for a routine in a particular context is simply the number of decision branches exercised in the context divided by the total number of decision branches in the routine.

The overall ***inheritance context decision coverage*** for the routine is then defined as the average of the inheritance context decision coverage in each context appropriate to the routine. For a routine defined in a base class the appropriate contexts are the context which corresponds to the base class along with those corresponding to each derived class which inherits the routine unchanged. Note that the routine need not (more accurately, *can* not) be tested in the context of derived classes which provide an overriding definition of the routine.

Thus the overall inheritance context decision coverage for a routine is given by:

$$InheritanceContextDecnCoverage = \frac{\sum_{i=1}^{NumberOfContexts} DecnBranchesExercisedInContext_i}{NumberOfDecnBranchesInRoutine \times NumberOfContexts} \times 100\%$$

As with the standard coverage metrics, system-wide averages of inheritance context coverage (across all routines in the system) can also be defined.

3.7. An Example Using Decision Coverage

Let’s assume that functions `foo()` and `bar()` shared significant amounts of common code. In an attempt to simplify the class, the functions are merged into one, with a boolean parameter determining the behaviour[2]:

```
class Base {
public:
      void foo_or_bar(bool flag) {
            ...
            if (flag) {
                  ...
                  helper();
                  ...
            } else {
                  ...
                  helper();
                  ...
            }
```

[2] This example is for the purposes of explanation only. In practice such a transformation would not normally be made – if “foo-ing” and “bar-ing” are separate enough concepts to have different names then it is unlikely that the merged method would be functionally cohesive. A better solution would be to extract the common code into separate private methods invoked by both `foo()` and `bar()`.

```
            ...
      }
private:
      virtual void helper()         { ... }
};
class Derived : public Base {
private:
      virtual void helper()         { ... }
};
void test_driver() {
      Base    base;
      Derived derived;
      base.foo_or_bar(false);       // Test Case A1
      derived.foo_or_bar(true);     // Test Case A1
}
```

The branches of `foo_or_bar()` have not been fully tested in the context of either `Base` or `Derived` – only 50% inheritance context decision coverage has been achieved in each case.

Note that in this example traditional decision coverage would misleadingly indicate 100% coverage of `Base::foo_or_bar()`.

3.8. What Are The Contexts?

In the examples above the valid contexts for the inherited methods (`Base::foo()` and `Base::bar()` or `Base::foo_or_bar()`) are simply "Base" and "Derived".

For the method `Base::helper()` the sole valid context is "Base", since the method is not inherited by `Derived` but is instead overridden.

For the overriding definition `Derived::helper()` the only valid context is "Derived".

3.9. Hierarchical Integration Testing

Let us consider the level of testing required for inherited methods which are inherited, using the example shown in figure 4.

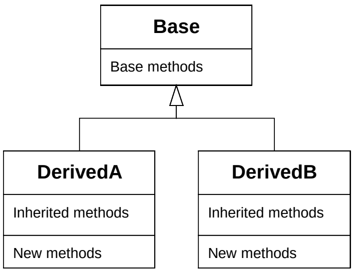

Figure 4: The software under test

The Hierarchical Integration Testing (HIT) approach to unit testing was proposed as a technique for ensuring thorough class testing (see [Harrold] for full details). HIT recommends that as a first step all methods be tested fully in the context of a particular class (the base class or, for abstract base classes, a particular derived class).

For a base class this would typically be interpreted as a coverage requirement of 100% decision coverage of each defined method in the context of the particular class. This criterion is also known as “Once-Full” coverage and is equivalent to the minimum traditional coverage requirement for thorough unit testing.

This recommendation applies to all classes, so that re-definitions of a method in a derived class (“overridden” methods) are tested with the same thoroughness as the original base class definition.

3.9.1. Interaction Coverage

The HIT approach further recommends that any methods which are inherited by a derived class and which interact with any re-defined methods should be re-tested in the context of the derived class. The focus of this re-testing is to exercise the interactions between the inherited methods and the re-defined method(s) – primarily the calls between the methods. This recommendation could be interpreted as a coverage requirement of 100% call-pair coverage of the inherited methods in the context of each derived class.

3.9.2. Strict Coverage

In practice the integration test cases which exercise the interactions between methods are often spread throughout the complete test suite for the class.

Rather than separating the integration test cases out, sometimes the easiest way to ensure that interactions are thoroughly re-tested is to re-run *all* of the base class test cases. This conservative approach can be enforced through the application of a stricter coverage requirement, namely that 100% decision coverage is achieved for each base class method in the context of *every* derived class by which it is inherited.

3.10. Achieving Inheritance Coverage Is Easy

During unit testing the effort required to achieve inheritance context coverage is not significantly greater than that required to achieve coverage according to traditional metrics.

Typically, no additional test cases are required. Instead, test cases already written to test the base class are used to re-test the inherited methods in the context of the derived class. The test cases form a ***parallel inheritance hierarchy*** which mirrors the inheritance structure of the software under test and enables base class test cases to be easily re-used to test derived classes (see Figure 5).

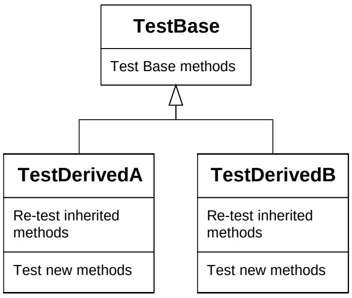

Figure 5: Test classes and software under test form parallel inheritance hierarchies

This re-use of base class test cases has the additional benefit of automatically testing the design for conformance to the Liskov Substitutability Principle (LSP). The LSP is an important object-oriented design principle (described in [Liskov]) which helps ensure that inheritance hierarchies are well-defined. The use of a parallel inheritance hierarchy also forms the basis of the PACT method described in [McGregor].

The only costs for this re-use are the (small) initial effort to ensure that the test cases are structured so as to be re-usable and the CPU time required to re-run the test cases.

4. State-Based Context Coverage

In most object-oriented systems there will exist a number of classes which can best be described as “state machines”. Objects of these classes can exist in any of a number of distinct states, and the behaviour of each class is qualitatively different in each possible state – the behaviour of the class is state-dependant. ***State-based context coverage*** is designed to measure how fully this behaviour has been tested.

Consider a typical class with state-dependent behaviour: a bounded stack. A bounded stack can be in one of three possible states: ‘empty’, ‘partially full’ or ‘full’. The behaviour of the stack is qualitatively different in each state. For example, the `pop()` operation removes and returns the top element from the stack in states ‘partially full’ or ‘full’ but throws an exception and leaves the stack unmodified in state ‘empty’.

The class interface for the bounded stack class is shown below:

```
class BoundedStack {
public:
      BoundedStack(size_t maxsize);
      ~BoundedStack();
      void push(int);
      int pop();
      struct underflow : std::exception { };
      struct overflow : std::exception { };
};
```

4.1. Black-Box Testing using Entry-Point Coverage

When testing a class, the first step is to write test cases to exercise the public interface to the class:

```
int test_driver() {
      BoundedStack stack(2);
      stack.push(1);
      stack.pop();
      // destructor called implicitly at end of block
};
```

We can use entry-point coverage to ensure that the test cases exercise each of the methods of the class. The test cases above achieve 100% entry-point coverage.

However, it is clear that this test set does not fully exercise the `BoundedStack` class; the stack never becomes full and an exception is never thrown.

4.2. Using White-Box Coverage Metrics

If entry-point coverage does not ensure thorough testing, perhaps we should use a stronger coverage requirement, say, 100% decision coverage. Unfortunately, there are disadvantages to the adoption of such a requirement.

One factor is that there may be decisions in the code which do not correspond to features of the public interface. Typical examples include error handling code and defensive programming idioms. In these cases, 100% decision coverage may be difficult to achieve.

A more significant problem is the inability of traditional structural coverage metrics to identify code which is missing altogether. For example, consider what would happen if the `BoundedStack` implementor forgot to check for the stack-empty condition in `pop()`. Requiring 100% decision coverage would not help find this fault – the missing condition is not there to be covered!

4.3. We Can Do Better

Actually, we can write a better test set than that required by any traditional structural coverage metric, and without any knowledge of the internal details of the class.

The UML state transition diagram (see figure 6) shows how the behaviour of the class changes, depending on the current state. This type of diagram is commonly used to describe the state-dependent behaviour of classes

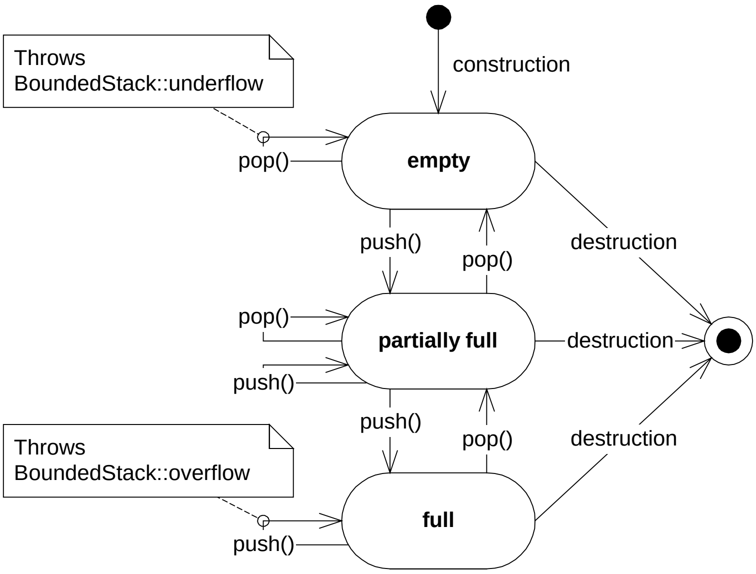

Figure 6: State transition diagram for BoundedStack

We can use the additional information provided in the state transition diagram to design a test set which thoroughly exercises the `BoundedStack` class.

4.4. Writing State-Driven Test Cases

The aim of our test design is to exercise every method in every possible state. In the improved test set which follows, the `push()` and `pop()` methods are invoked on an empty, partially full and full stack:

```
int better_test_driver() {
      BoundedStack stack(2);
      stack.push(3);                // push() when empty
      stack.push(1);                // push() when partially-full
      try { stack.push(9); }        // push() when full
      catch (BoundedStack::overflow) { }  // expected to throw
      stack.pop();                  // pop() when full
      stack.pop();                  // pop() when partially-full
      try { stack.pop(); }          // pop() when empty
      catch (BoundedStack::underflow) { } // expected to throw
      // destructor called implicitly at end of block
};
```

The crucial issue is: does this test set fully exercise the `BoundedStack` class?

State-based context coverage is designed to answer this question.

4.5. State-Based Context Coverage

State-based context coverage is similar to inheritance context coverage: it provides alternative definitions of the traditional structural coverage metrics. The alternative definitions differ in that they separately measure coverage in different contexts.

With inheritance context coverage the contexts depend on the inheritance structure of the software under test. Similarly, with state-based context coverage the contexts correspond to the potential states of objects of the class under test.

Thus the state-based context coverage metrics regard execution of a routine in the context of one state (that is, when invoked on an object in that state) as separate from execution of the same routine in another state. To achieve 100% state-based context coverage, the routine must be exercised in each appropriate context (state).

4.6. Definition

The state-based context coverage variant of entry-point coverage for a class (in the context corresponding to a particular state) is given by the number of methods which have been invoked on objects in that state divided by the total number of methods in the class.

The overall state-based context entry-point coverage is then defined as the average of the state-based context entry-point coverage in each state appropriate to the class.

Thus the overall state-based context entry-point coverage for a routine is given by:

$$StateBasedContextEntryPointCoverage = \frac{\sum_{i=1}^{NumberOfStates} MethodsExercisedInState_i}{NumberOfMethodsInClass \times NumberOfStates} \times 100\%$$

Note that, for the purposes of state-based context coverage, constructors are not considered to be methods of the class, in the sense that when they are invoked the object does not (yet) exist, and thus there is no state with which to associate the coverage.

Alternatives are similarly defined for each of the traditional structural coverage metrics.

As with the standard coverage metrics, system-wide averages of state-based context coverage (across all classes in the system) can also be defined.

4.7. What Are The Contexts?

The contexts for state-based context coverage are just the states appropriate to the class. Thus, in the bounded stack example the contexts are “empty”, “partially-full” and “full”.

Unfortunately, the presence of these states is a design feature which is not directly visible in the code. In order for a tool to automatically gather state-specific coverage information, the code must be changed so that it defines the possible states and so that the tool can determine the current state.

The simplest solution is to define an additional member function which returns a string containing the current state. For example:

```
class BoundedStack {
public:
      ...
private:
      const char* cppca_user_context_function() const {
            if (num_elements == 0) return "empty";
            else if (num_elements == max_elements) return "full";
            else return "partially-full";
      }
};
```

When provided by the user, this specially-named member function is used automatically by Cantata++ to determine the current state while gathering coverage data.

4.8. How Well Did We Do?

Using the test cases from `better_test_driver()` to test the `BoundedStack`, we find that we still haven’t achieved 100% state-based entry-point coverage. The destructor has not been exercised in the “partially-full” or “full” states.

Although state-based coverage does not apply to constructor functions, it does apply to destructors. Exercising destructors in all states helps ensure that all allocated resources are properly freed.

The following, further enhanced, test set does achieve 100% state-based entry-point coverage:

```
int even_better_test_driver() {
      BoundedStack stack(2);
      stack.push(3);                // push() when empty
      stack.push(1);                // push() when partially-full
      try { stack.push(9); }        // push() when full
      catch (BoundedStack::overflow) { }  // expected to throw
      stack.pop();                  // pop() when full
      stack.pop();                  // pop() when partially-full
      try { stack.pop(); }          // pop() when empty
      catch (BoundedStack::underflow) { } // expected to throw
      BoundedStack stack2(3);
      stack2.push(6);               // stack2 is partially-full
      BoundedStack stack3(1);
      stack3.push(6);               // stack3 is full
      // destructors called implicitly at end of block for
      // stack (empty), stack2 (partially-full) and stack3 (full)
};
```

4.9. Extra Effort Gives Extra Benefit

State-based context coverage is unusual because it requires the user to provide additional information (in the form of the additional member function to return the current state).

In contrast, traditional structural coverage metrics such as decision and condition coverage are based purely on the structure of the software under test and therefore require no additional information. As a result, traditional structure coverage metrics are typically weak at highlighting “faults of omission” – requirements which have simply not been implemented. Suppose, through an oversight, that the code to implement a particular requirement is missing. There is no hint in the code that the requirement exists, and no traditional structural coverage metric will force the testing of the missing requirement.

The use of an independent source of additional information means that state-based context coverage has the potential to go beyond the constraints of traditional structural coverage metrics and highlight these faults of omission.

For example, areas of unimplemented functionality can be identified by analysing those areas for which state-based entry-point context coverage has not been achieved while 100% traditional (non-context) decision coverage has been achieved.

4.10. Achieving State-Based Coverage

Achieving 100% state-based entry-point coverage typically requires more test cases than are required for traditional entry-point coverage and fewer than are required for traditional decision coverage. Moreover, because the test cases which are required to achieve state-based entry-point coverage are based on the design they are usually just those which would form a normal black-box test set, particularly if an approach such as [Binder]’s “FREE” is used. This reduces the need to write additional test cases purely to achieve a defined coverage target.

5. Thread-Based Context Coverage

The concept of context coverage which underpins both inheritance context coverage and state-based context coverage can also be used to assist with other coverage analysis problems.

For example, when testing a multithreaded application, traditional structural coverage metrics will “aggregate” the coverage achieved by all the threads into a single coverage value.

The context coverage approach can be applied here to maintain separate coverage information for each thread. As with state-based context coverage, a user-provided function is used automatically by Cantata++ to determine the current context:

```
const char* cppca_user_context_function() const {
      static char buffer[16];
      sprintf(buffer, "%x", OS::Threads::GetCurrentThreadID());
      return buffer;
}
```

The additional coverage information provided through context coverage can be used o ensure that each individual thread is thoroughly tested. Moreover, the raw coverage data (which statements and decisions are executed in which threads) can be used to analyse how the software under test actually behaves in the presence of complex thread-interaction issues.

6. Conclusion

Traditional structural coverage metrics are inadequate measures of test thoroughness for object-oriented software systems. New object-oriented context coverage metrics are required to ensure thorough testing. Inheritance, state-based and thread-based context coverage metrics are a useful and practical addition to the unit tester’s tool set.

Inheritance context coverage can be used, in conjunction with traditional structural coverage metrics and with little additional cost to ensure that polymorphic interactions between methods are fully tested in each derived class. Similar techniques can be applied to uses of template-based compile-time polymorphism.

State-based context coverage can be used, again in conjunction with traditional structural coverage metrics, to ensure that classes whose behaviour depends on an internal state have been thoroughly tested. In particular, state-based entry-point coverage is an appropriate metric for black-box unit-testing of classes with state-dependent behaviour.

The underlying techniques of context coverage can also be extended to apply to assist with other coverage problems, such as the testing of multi-threaded applications.

7. References

[Binder] Binder,B., *The FREE Approach to Testing Object-Oriented Software: An Overview*, http://www.rbsc.com/pages/FREE.html.

[Harrold] Harrold,M.J. and J.D.McGregor, *Incremental Testing of Object-Oriented Class Structures*, http://www.cs.clemson.edu/~johnmc/papers/TESTING/HIT/hit.ps.

[Liskov] Liskov, B. and J.Wing, *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, Vol 16, No 6, November, 1994, pages 1811-1841.

[McGregor] McGregor,J.D. and A.Kare, *PACT: An Architecture for Object-Oriented Component Testing*, Proceedings of the Ninth International Software Quality Week, May 1996.