

# Abstract Coverage Specification

(extended abstract)

I.S.W.B. Prasetya and M. Gerritsen  
Dept. of Inf. and Comp. Sciences, Utrecht Univ.  
wishnu@cs.uu.nl, mggerrit@cs.uu.nl  
URL: <http://www.cs.uu.nl/~wishnu>

September 12, 2008

At Utrecht University we are currently developing a coverage utility to support our automated unit testing tool for Java called T2. A rather unexpected result of this utility is that it gives us the ability to express more abstract notions of coverage than the traditional line or branch coverage.

The unit to test in T2 is a class as a whole. It does not test each method in isolation, as many other tools do; rather, it generates random sequences of field updates and method calls to test the target class. T2 supports model-based testing tool. For more on T2 see:

<http://www.cs.uu.nl/wiki/WP/T2Framework>

Our coverage utility measures so-called *prime paths* coverage [1]. Roughly, a prime path is either a maximal non-circular path or an elementary circuit in the control flow of a program. Furthermore the utility has some intelligence to generate more T2 tests to improve coverage. Now, it turns out that we can lean on this prime path coverage to express more abstract notions of coverage, namely abstract-state coverage and model coverage explained below.

Suppose we have a class `Queue`. When testing this class we may feel that it is useful to cover the cases where the queue is empty, almost empty, full, and almost full. Essentially, we would like to define an abstract domain  $D$  to which we map the concrete states of instances of `Queue`. Coverage over this abstract domain is called "*abstract-state coverage*". This is a very useful notion of coverage, because it is more abstract and is more semantic related. It is abstract because it is expressed in terms of high level concepts (e.g. the queue being empty or full) rather than e.g. line number. Although we can infer abstract-state coverage from traditional code coverage, the process can be quite hard and it has to be done manually.

In T2 specifications are written in plain Java. This mean a lot for developers, because they remain in a familiar ground. This also means that we can conveniently express our mapping to abstract domain (as meant above) as plain Java methods. Since it may not be relevant to cover the whole domain, we introduce the notion of *coverage specification* for specifying the relevant part of the abstract domain that should be covered. However, our coverage utility only measures prime path coverage. The trick is to encode our coverage specification as control paths in the class invariant of `Queue`. To show this, consider the code below. It shows the class `Queue`. We do not show its members, except for the field `n`. The boolean methods shown implement our mapping to abstract states as meant above. T2 can also

check class invariant, so we include a class invariant for `Queue` which simply requires `x` to be non-negative.

---

```
begin class Queue {  
  
    private int n ;  
  
    ... // other members  
  
    // These define mapping to abstract state:  
    public boolean isEmpty() { ... }  
    public boolean isFull() { ... }  
    public boolean isAlmostEmpty() { ... }  
    public boolean isAlmostFull() { ... }  
  
    // class invariant  
    private boolean classinv() {  
  
        return n>=0  
    }  
}
```

---

We get full abstract-state coverage if our tests can produce all combinations of the values of `isEmpty`, `isFull`, `isAlmostEmpty`, and `isAlmostFull`. Since T2 check the class invariant at the end of every test step in the sequences it generates, we can piggy back on the the class invariant to peek into the values of those abstract mapping:

---

```
private boolean classinv() {  
    // peeking:  
    if (isEmpty()) { }  
    if (isFull()) { }  
    if (isAlmostEmpty()) { }  
    if (isAlmostFull()) { }  
    // the actual expression to check as the class invariant:  
    return n>=0  
}
```

---

The code in the peeking part does not do anything. However notice that full path coverage over this method implies full abstract-state coverage. Note also that we really need path coverage. Branch coverage will not do.

Certain parts of the abstract domain may be absurd. E.g. the combination where `isEmpty` and `isFull` are both true is usually absurd, so we should not actually try to cover it. We can express this. We just write a different peeking code, e.g:

---

```

private boolean classinv() {
    // peeking:
    if (isEmpty()) { }
    else if (isFull()) { }
    else {
        if (isAlmostEmpty()) { }
        if (isAlmostFull()) { }
    }
    // the actual expression to check as the class invariant:
    return n>=0
}

```

---

Notice that peeking part now specifies which part of the abstract domain that we want to cover. That is why we call it coverage specification. To improve readability we can also write it separately:

---

```

private void CoverageSpec() {
    if (isEmpty()) { }
    else if (isFull()) { }
    else {
        if (isAlmostEmpty()) { }
        if (isAlmostFull()) { }
    }
}

private boolean classinv() {
    coverageSpec() ;
    return n>=0
}

```

---

The above encoding introduces indeed an indirection, but it also gives automation. The prime path coverage utility will now tell us exactly which control paths have been visited, and thus which part of the abstract domain are covered. Furthermore, the utility also has a built-in ability to generate more tests to improve coverage.

The same trick can be applied at the method level too. That is, we can augment the precondition of a method with its own abstract coverage specification. For example, suppose Queue has a method called `valOf(i)` that would return the  $i$ -th element of the queue. We may want to specifically cover the cases when  $i$  is zero, negative, and is equal to  $n - 1$ . We can express this as coverage specification:

---

```
public Item valOf(int i) {
    // Coverage specification:
    if (i==0) { }
    else if (i<0) { }
        else if (i==n-1) { }

    // the actual body of valOf:
    ...
}
```

---

In T2 you can also write the specification of a method *m* separately in a method called *m\_spec*. This avoids cluttering. So we can write e.g. :

---

```
public Item valOf(int i) { ... }

public Item valOf_spec(int i) {
    // Coverage specification:
    if (i==0) { }
    else if (i<0) { }
        else if (i==n-1) { }

    Item result = valOf(i) ;
    // You can specify post condition here, if you have one:
    assert ... ;
    return result ;
}
```

---

In T2 we can also specify an *application model*. The application model of e.g. the class `Queue` specifies a proper use of instances of this class by an environment (other class or an application). *Model coverage* is graph-based coverage (e.g. branch or prime path coverage) in terms of this model (rather than directly in terms of `Queue` itself). Because in T2 an application model is expressed as a plain Java method, it is also within the scope of our path coverage utility. So, through this utility we can also measure our model coverage. We get this feature for free.

Our research in coverage utility is still on-going, but we think it may be interesting to share our progress with the Dutch Test Day community.

## References

- [1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.